



HOW COMPUTERS WORK

From a Grain of Sand to Artificial Intelligence



KEDBYTE

TECHNOLOGIES PRIVATE LIMITED

HOW COMPUTERS WORK

From a Grain of Sand to Artificial Intelligence

The complete plain-English guide to computing,
followed by the engineer's version of the same thing.

WRITTEN BY

Shikhar Singh

Founder & Chairman

KedByte Technologies Private Limited

Nine Parts • Fifty-Three Chapters • First Edition

HOW COMPUTERS WORK

From a Grain of Sand to Artificial Intelligence

Author	Shikhar Singh
Designation	Founder & Chairman, KedByte Technologies Private Limited
Published by	KedByte Technologies Private Limited
Imprint	KedByte Press
Edition	First Edition
Subject	Computer science, engineering, general reference
Extent	Nine parts, fifty-three chapters

Copyright © KedByte Technologies Private Limited. All rights reserved.

No part of this publication may be reproduced, distributed or transmitted in any form or by any means, including photocopying, recording or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other non-commercial uses permitted by copyright law.

Trademarks. All product names, company names, service marks and trademarks referred to in this book are the property of their respective owners. They are used in this book in an editorial and educational capacity only, with no intention of infringement of the trademark. No such use, or the use of any trade name, is intended to convey endorsement or any other affiliation with this book or its publisher.

Accuracy and currency. This book describes technology that changes. Specifications, prices, version numbers, product names and industry figures were checked at the time of writing and are dated in the text wherever they are liable to change. The chapters on artificial intelligence in particular describe a field that moves quickly; readers are told, in the text, where to re-check a figure before relying on it. Where experts disagree, the book says so rather than choosing a side.

Disclaimer. The information in this book is provided on an “as is” basis, for education. The security chapter is written for defence: it explains how classes of attack work so that systems can be built to resist them, and it deliberately contains no working exploit code. Neither the author nor KedByte Technologies Private Limited shall have any liability to any person or entity with respect to any loss or damage caused, or alleged to have been caused, directly or indirectly by the information contained in this book.

Worked examples. Parts F and G are built around one real network fault and one real version-control session, reproduced exactly as they were observed, including the addresses, the traceroute output and the error messages. Nothing in those traces has been altered or invented.

From the Founder & Chairman

Most people who use computers every day have never been shown how one works. That is not their failing. It is ours, collectively, as an industry. We have built something extraordinary and then wrapped it in language that keeps people out.

I have watched capable students give up on the subject, not because it is beyond them, but because every explanation they found started somewhere in the middle. Someone explains what RAM is without ever explaining what a bit physically is. Someone explains an API without ever saying what a function call was in the first place. The gaps compound, quietly, until a person decides they are simply not a technical person. Almost always, they are wrong about that.

This book exists to close those gaps in a single read. It starts with electricity in a wire. It ends with how a language model is trained, judged and served. In between it goes through silicon pulled from the ground, transistors, logic, memory, processors, storage, compilers, operating systems, the terminal, files, networks, the internet, the web, security, games and artificial intelligence. Nothing is assumed. Nothing important is skipped.

The method matters more than the coverage. Every idea in this book is explained twice. First in plain language, with an everyday comparison and a worked example, in short numbered lines, so that a twelve-year-old could follow it. Then again in the correct technical vocabulary, with the real units, the real specifications and the real numbers an engineer would quote in a design review. The two halves are labelled so you can read the plain pass end to end, understand the whole of computing, and then come back for the engineering.

There are two commitments in these pages that I care about particularly. The first is that where a simple explanation is not strictly true, the book says so and gives the accurate version underneath. A useful lie told without warning is how people end up confidently wrong. The second is that the book distinguishes what is a written standard, what is merely a common convention, and what is one product's implementation detail. Engineers who cannot tell those apart make expensive mistakes.

Parts F and G are unusual, and deliberately so. Rather than invented examples, they follow one real network failure and one real version-control session from beginning to end: a real router, a real traceroute across three Indian cities, a real fifteen-second silence, and two real failed pushes. Diagnosis taught against real evidence is worth far more than diagnosis taught against a di-

agram, because real evidence is ambiguous and you have to learn what it does and does not prove.

At KedByte Technologies Private Limited we take the view that understanding should be transferable and not gate-kept. If this book leaves one reader able to trace a keystroke from their finger to the light on the screen, and to say honestly which part of that chain they have verified and which part they have merely assumed, it has done its job.

Start at Chapter 1. Take your time. It is all connected.

Shikhar Singh

Founder & Chairman

KedByte Technologies Private Limited

How to read this book

The shape of every section

Every section of every chapter is built from the same six blocks, always in the same order. Once you know the shape, you can navigate two thousand pages without ever being lost.

Block	What it is for
PLAIN in simple words	The idea for a complete beginner. No jargon at all.
PLAIN a picture in your head	One everyday comparison, and then a line telling you exactly where that comparison stops being true.
PLAIN a worked example	Concrete numbers, worked step by step.
PLAIN what is really happening inside	The mechanism, still in easy words, but with nothing hand-waved.
TECHNICAL the engineer's version	The correct professional vocabulary, exact units, real specifications, real figures, standard numbers and the commands that observe it.
WORDS remember these	Each term given twice: the plain meaning, then the technical meaning.

Two ways to read it

1. **The single pass.** Read everything, in order, from Chapter 1. This is the intended route and it works.
2. **The double pass.** Read only the blocks marked PLAIN, front to back. You will understand the whole of computing. Then return to the start and read the blocks marked TECHNICAL to be able to hold your own in an engineering conversation.

Conventions used throughout

1. Almost everything is written as short numbered lines, one idea per line, so nothing hides inside a paragraph.
2. *The honest version* marks the places where a simple explanation is not strictly true. The accurate model follows immediately.

3. The book states whether a thing is a **standard** (written down in a specification), a **convention** (everybody just does it this way), or an **implementation detail** (true of one product, not of the idea).
4. In the artificial intelligence chapters, claims are separated into established fact, active research, and marketing claim.
5. Anything that goes stale is dated. Check it again before you rely on it.
6. Every chapter ends with *Common wrong ideas* and a twenty-line summary. If you are short of time, read those two first.

Where to find things

1. **Chapter 1** is the map of the whole book.
2. **Chapter 52** is the reference desk: the numbers, the command cheat sheets and the two diagnostic decision trees.
3. **Chapter 53** is the master timeline and a glossary of more than seven hundred terms, each given in plain words and then technically.

Contents

Part A — The Physical World

1

1 The Whole Machine in One Look

2

1.0 What this chapter gives you	2
1.1 What a computer actually is	2
1.2 The five parts of every computer ever built	4
1.3 The one big idea: instructions and data are the same stuff	5
1.4 The layer cake	7
1.5 A journey in one page: you press the letter A	9
1.6 What “fast” and “big” mean here	11
1.7 Units done properly	13
1.8 Hardware, software and firmware	14
1.9 How to read this book	16
1.98 Common wrong ideas	18
1.99 Chapter summary in 20 lines	19

2 Electricity — The Thing That Does All The Work

20

2.0 What this chapter gives you	20
2.1 What electricity really is	20
2.2 Voltage, current and resistance	22
2.3 A circuit: the loop that makes it work	24
2.4 Conductors, insulators and semiconductors	26
2.5 The resistor	28
2.6 The capacitor	30
2.7 The inductor and the coil	33
2.8 The diode	35
2.9 Signals: analogue and digital	37
2.10 Clocks: how a computer keeps time	39
2.11 How one bit is physically stored and moved	42
2.98 Common wrong ideas	45
2.99 Chapter summary in 20 lines	46

3 From Sand to Silicon — Rock, Furnace, Wafer, Fab

47

3.0 What this chapter gives you	47
3.1 Where silicon comes from	47
3.2 First refining: the submerged-arc furnace	49
3.3 Getting to nine nines	50
3.4 Growing a single crystal	52
3.5 From boule to wafer	54
3.6 The cleanroom	56
3.7 Photolithography, the heart of it	58
3.8 The light	60
3.9 The other steps	63
3.10 Testing, dicing, packaging	65
3.11 What “3 nm” actually means now	67
3.12 Who makes chips and why it matters	69
3.98 Common wrong ideas	71
3.99 Chapter summary in 20 lines	72

4	The Transistor — The Switch That Built The World	74
4.0	What this chapter gives you	74
4.1	Before the transistor: relays and glowing glass	74
4.2	Doping, properly	76
4.3	The PN junction and the diode	78
4.4	The first transistor, and the argument about it	80
4.5	The bipolar junction transistor	82
4.6	The MOSFET, the transistor inside your computer	84
4.7	CMOS: the pairing that made low power possible	87
4.8	The same device, two jobs	90
4.9	The changing shape of the transistor	91
4.10	Scale, Moore's Law, and the power wall	94
4.11	What a transistor costs	96
4.98	Common wrong ideas	98
4.99	Chapter summary in 20 lines	99
5	Logic Gates and Arithmetic — How Switches Learn to Add	101
5.0	What this chapter gives you	101
5.1	Boolean algebra: true and false as 1 and 0	101
5.2	Building gates from real MOSFETs	104
5.3	The seven gates	108
5.4	Universality: why NAND alone is enough	111
5.5	Writing and simplifying logic	114
5.6	Adding	118
5.7	Subtracting without a subtractor	122
5.8	More building blocks	124
5.9	The ALU	127
5.10	Multiply and divide in hardware	130
5.11	Propagation delay and the critical path	133
5.12	Why floating point maths is not exact	136
5.98	Common wrong ideas	139
5.99	Chapter summary in 20 lines	140
6	How a Machine Remembers — Latches, Clocks and Memory Cells	142
6.0	What this chapter gives you	142
6.1	The problem: a gate forgets the moment its input changes	142
6.2	The SR latch from cross-coupled NOR gates	144
6.3	The gated D latch	146
6.4	The D flip-flop: edge-triggered memory	147
6.5	The clock: why the whole chip marches in step	150
6.6	Registers, shift registers and counters	152
6.7	Finite state machines	154
6.8	SRAM: the six-transistor cell	157
6.9	DRAM: one transistor and one capacitor	159
6.10	Non-volatile cells: from mask ROM to flash	162
6.11	Memory as a grid: addresses, word lines and bit lines	165
6.12	Volatile, non-volatile, and memory versus storage	167
6.98	Common wrong ideas	169
6.99	Chapter summary in 20 lines	170
	Part B — Turning Bits Into Meaning	171
7	Numbers and Meaning — Binary, Text and Unicode	172
7.0	What this chapter gives you	172
7.1	Why base 2	172
7.2	Hexadecimal and octal	175
7.3	Bits, bytes and words	178

7.4 Negative numbers	181
7.5 Fractions: fixed point and floating point	184
7.6 Text before Unicode	188
7.7 Unicode and UTF-8	191
7.8 Endianness	195
7.9 Checking that data is intact	197
7.10 Compression	200
7.11 How a number becomes a thing	204
7.98 Common wrong ideas	206
7.99 Chapter summary in 20 lines	207

8 Pixels, Screens and How You See an Image 209

8.0 What this chapter gives you	209
8.1 Light and the eye	209
8.2 What a pixel actually is	211
8.3 Resolution and PPI	212
8.4 Colour	214
8.5 How a display makes light, part 1: LCD	216
8.6 How a display makes light, part 2: OLED	218
8.7 The older ways	220
8.8 The framebuffer	221
8.9 Getting the picture out	223
8.10 Refresh rate, frame rate and smoothness	225
8.11 Touchscreens	227
8.12 Scaling and sharpness	229
8.98 Common wrong ideas	231
8.99 Chapter summary in 20 lines	232

9 Sound, Magnets, Speakers and Phone Calls 233

9.0 What this chapter gives you	233
9.1 What sound actually is	233
9.2 The speaker, in full	235
9.3 The microphone as a speaker in reverse	239
9.4 From wave to numbers	241
9.5 From numbers back to wave	243
9.6 Storing audio	246
9.7 How recorded sound was stored before computers	249
9.8 Audio inside a computer	251
9.9 Phone calls, part 1 – the old way	253
9.10 Phone calls, part 2 – mobile and internet	256
9.11 Noise cancelling, spatial audio and the modern tricks	259
9.98 Common wrong ideas	262
9.99 Chapter summary in 20 lines	262

Part C — The Machine 264

10 Inside the CPU 265

10.0 What this chapter gives you	265
10.1 What a CPU is: a box that repeats one loop forever	265
10.2 The stored-program idea	268
10.3 The fetch-decode-execute cycle	270
10.4 Instruction set architecture	273
10.5 Registers in real CPUs	276
10.6 Microcode	278
10.7 Pipelining	281
10.8 Going wider and smarter	284
10.9 SIMD and accelerators	287

10.10 Many cores	290
10.11 Caches from the CPU side	294
10.12 Interrupts, exceptions and privilege	296
10.13 Reading a real spec sheet	300
10.14 How fast is a CPU really	303
10.98 Common wrong ideas	305
10.99 Chapter summary in 20 lines	306
11 The Memory Hierarchy – Cache, RAM and Virtual Memory	307
11.0 What this chapter gives you	307
11.1 Why a hierarchy exists at all	307
11.2 Locality: why the whole idea works	310
11.3 RAM modules in the real world	312
11.4 The memory controller	315
11.5 The address space	317
11.6 Virtual memory, explained slowly	319
11.11 Measuring memory for real	333
11.12 Cache-friendly programming	335
11.99 Chapter summary in 20 lines	338
12 Storage – Disks, SSDs, Flash and Filesystems	340
12.0 What this chapter gives you	340
12.1 Storage versus memory: the real difference	340
12.2 The old ways: cards, tape, drums, cores and floppies	342
12.3 The hard disk drive, in full	345
12.4 HDD geometry and performance	348
12.5 Flash memory, in full	350
12.6 The SSD controller and the flash translation layer	353
12.7 SSD endurance and failure	357
12.8 How storage connects: interfaces and buses	359
12.9 What a filesystem is	362
12.10 Real filesystems compared	364
12.11 Journaling, consistency and crashes	367
12.12 Partitions, formatting and booting	370
12.13 RAID and redundancy	372
12.14 Data recovery, deletion and secure erase	375
12.98 Common wrong ideas	378
12.99 Chapter summary in 20 lines	378
13 Buses, Ports, Drivers and Firmware	380
13.0 What this chapter gives you	380
13.1 The motherboard as a city	380
13.2 What a bus actually is	383
13.3 The history of expansion buses	385
13.4 How the processor talks to a device	388
13.5 Interrupts, end to end	391
13.6 USB, properly	394
13.7 The other ports on the machine	397
13.8 What a driver actually is	399
13.9 Driver mechanics	402
13.10 Firmware: code that lives on the device	405
13.11 BIOS and UEFI	407
13.12 Plug and play, resources and conflicts	410
13.13 System-on-chip: when the whole board fits on one die	412
13.98 Common wrong ideas	414
13.99 Chapter summary in 20 lines	415
14 The Long Road – A History of Computing Machines	416

14.0 What this chapter gives you	416
14.1 Counting before machines	416
14.2 Machines that follow instructions	419
14.3 Charles Babbage and Ada Lovelace	421
14.4 Hollerith, the 1890 census, and the birth of IBM	424
14.5 The theory arrives: Hilbert, Gödel, Church and Turing	426
14.6 War machines, 1939 to 1945	430
14.7 ENIAC and EDVAC	433
14.8 The first stored-program computers	436
14.9 Transistors, mainframes and minicomputers	439
14.10 The integrated circuit and Silicon Valley	441
14.11 The microprocessor	443
14.12 The personal computer revolution	446
14.13 The graphical interface	449
14.14 Networks, the web and mobile: the spine only	451
14.15 The last twenty years	453
14.16 One master timeline, 1801 to 2026	456
14.98 Common wrong ideas	459
14.99 Chapter summary in 20 lines	460

Part D — Software

461

15 The Birth of Software — Punch Cards to the First Compiler **462**

15.0 What this chapter gives you	462
15.1 What a program is at the bottom	462
15.2 Before any language existed: programming by rewiring	464
15.3 Programming in raw numbers: switches, tape and cards	466
15.4 The first step up: mnemonics and assembly language	469
15.5 Why a higher language was needed	471
15.6 Grace Hopper and the first compilers	473
15.7 FORTRAN: the compiler that had to be fast	476
15.8 COBOL, LISP and ALGOL: three different futures	479
15.9 The moth, and what a bug really is	482
15.10 The rise of the operating system as software	484
15.11 UNIX and C	486
15.12 How software was shared, and then sold	488
15.13 A short history of the idea of a “programmer”	491
15.98 Common wrong ideas	493
15.99 Chapter summary in 20 lines	494

16 How a Compiler Actually Works **495**

16.0 What this chapter gives you	495
16.1 The whole pipeline in one view	495
16.2 The preprocessor	499
16.3 Lexing: characters into tokens	501
16.4 Parsing: tokens into a tree	503
16.5 Semantic analysis: does it make sense	506
16.6 Intermediate representation	508
16.7 Optimization	511
16.8 Code generation	515
16.9 The assembler	518
16.10 The linker	521
16.11 The loader and dynamic linking	524
16.12 Interpreters and virtual machines	527
16.13 Just-in-time compilation	529
16.14 Modern build reality	531
16.15 The bootstrap problem and trusting trust	534
16.98 Common wrong ideas	537

16.99 Chapter summary in 20 lines	538
17 Programming Languages and How to Think in Them	539
17.0 What this chapter gives you	539
17.1 What a programming language is	539
17.2 The building blocks every language has	542
17.3 Types	545
17.4 Memory management	548
17.5 Paradigms	551
17.6 Data structures, properly	554
17.7 Algorithms and Big-O	557
17.8 Concurrency and parallelism	561
17.9 Errors and correctness	564
17.10 A tour of the major languages	568
17.11 How to choose a language for a job	572
17.12 Reading other people's code and writing readable code	574
17.13 The standard library, packages and dependencies	577
17.98 Common wrong ideas	580
17.99 Chapter summary in 20 lines	581
18 Operating Systems	583
18.0 What this chapter gives you	583
18.1 What an operating system is and why it must exist	583
18.2 The kernel: the part that is really the operating system	586
18.3 The boot sequence, step by step	589
18.4 Processes	593
18.5 Threads	597
18.6 Scheduling	600
18.7 System calls	605
18.8 Memory management from the operating system's side	609
18.9 The filesystem from the operating system's side	612
18.10 Inter-process communication	615
18.11 Device management and drivers	619
18.12 Protection and isolation	622
18.13 UNIX: the full story	625
18.14 Linux	627
18.15 Windows	631
18.16 macOS, iOS, Android and the rest	634
18.17 How you would actually write one	636
18.98 Common wrong ideas	640
18.99 Chapter summary in 20 lines	640
19 The Terminal and the Shell	642
19.0 What this chapter gives you	642
19.1 What a terminal actually is	642
19.2 Terminal, terminal emulator, shell, console and command line	645
19.3 The pseudo-terminal	648
19.4 What a shell is	652
19.5 The shells themselves	655
19.6 stdin, stdout and stderr	657
19.7 Pipes	661
19.8 Exit codes	664
19.9 Environment variables	667
19.10 Configuration on each system	671
19.11 The essential commands	674
19.12 Globbing, quoting and expansion	681
19.13 Shell scripting	684
19.14 Getting comfortable	688
19.15 Package managers	691

19.98 Common wrong ideas	694
19.99 Chapter summary in 20 lines	695

20 Files, Formats, Extensions and the .exe 696

20.0 What this chapter gives you	696
20.1 What a file actually is	696
20.2 File extensions: a convention, not a rule	699
20.3 Magic numbers: the real identity of a file	701
20.4 Text files versus binary files	704
20.5 Common formats and how they are built	707
20.6 What an executable file really is	711
20.7 Anatomy of a PE file	714
20.8 Anatomy of an ELF file	718
20.9 What happens when you double-click a program	722
20.10 Installers and packaging	727
20.11 Scripts versus binaries	731
20.12 Archives and compression in practice	734
20.13 File metadata beyond the basics	737
20.14 Reading a file you do not understand	740
20.98 Common wrong ideas	743
20.99 Chapter summary in 20 lines	744

Part E — Seeing and Showing 745

21 From Key Press to Pixel — The Complete Trace 746

21.0 What this chapter gives you	746
21.1 The finger and the switch	746
21.2 The key matrix	749
21.3 The keyboard's own computer	752
21.4 Getting the data out: USB HID	755
21.5 Into the machine: the host controller	759
21.6 The kernel takes over	762
21.7 From key code to character	765
21.8 The window system	768
21.9 The application reacts	771
21.10 Drawing	774
21.11 Out to the screen	777
21.12 The complete time budget	780
21.13 The same trace for a mouse and a touchscreen	784
21.14 Why this matters	787
21.98 Common wrong ideas	790
21.99 Chapter summary in 20 lines	791

22 Graphics and the GPU 792

22.0 What this chapter gives you	792
22.1 Why a separate chip	792
22.2 Two-dimensional graphics first	795
22.3 The three-dimensional problem	798
22.4 The mathematics, taught gently	801
22.5 The rasterisation pipeline, stage by stage	805
22.6 Shaders	808
22.7 Textures and surface detail	812
22.8 Making it look real	815
22.9 Ray tracing	817
22.10 Upscaling and frame generation	820
22.11 Inside a real graphics chip	823
22.12 The software stack	826

22.13 Graphics chips for everything else	828
22.14 Integrated, discrete and mobile chips	831
22.98 Common wrong ideas	833
22.99 Chapter summary in 20 lines	834

Part F — Networks

836

23 Networking From Zero — Signals, Ethernet, MAC, ARP and DHCP

837

23.0 What this chapter gives you	837
23.1 What a network is: two machines and a shared medium	838
23.2 Bits on a wire, physically	840
23.3 Bits on fibre and on air	844
23.4 Bandwidth, throughput, latency and jitter	847
23.5 Ethernet	850
23.6 The Ethernet frame, field by field	853
23.7 MAC addresses	857
23.8 Hubs, switches and how a switch learns	860
23.9 VLANs and spanning tree, briefly	863
23.10 ARP: from an IP address to a MAC address	867
23.11 DHCP: how the laptop got an address	871
23.12 The network interface card	875
23.13 Putting the local network together	879
23.98 Common wrong ideas	882
23.99 Chapter summary in 20 lines	883

24 IP Addresses, Private and Public, NAT and Carrier-Grade NAT

885

24.0 What this chapter gives you	885
24.1 What an IP address is	885
24.2 Networks and hosts: the split, the mask and CIDR	888
24.3 Public and private addresses	892
24.4 The other special ranges	894
24.5 Address exhaustion: why 4.29 billion was not enough	898
24.6 NAT, explained completely	901
24.7 Ports: the trick that makes NAT work	904
24.8 What NAT breaks	907
24.9 Carrier-grade NAT	911
24.10 Reading the reader's own first six hops as a story	914
24.11 Static and dynamic addresses, and how yours is chosen	917
24.12 IPv6 in brief	920
24.13 Practical: finding your own addresses on macOS	923
24.98 Common wrong ideas	926
24.99 Chapter summary in 20 lines	927

25 DNS – From a Name to an Address

928

25.0 What this chapter gives you	928
25.1 The problem DNS solves	928
25.2 The shape of the name space	931
25.3 The four kinds of participant	933
25.4 One full lookup, step by step	936
25.5 The root servers	939
25.6 Record types	941
25.7 Caching and TTL	945
25.8 Why the reader's machine uses 1.1.1.1	948
25.9 Why the same name gives different answers	950
25.10 The reader's own answer decoded	953
25.11 DNS on the wire	956
25.12 DNS security and privacy	959

25.13 Owning a name	962
25.14 Diagnosing DNS for real	965
25.98 Common wrong ideas	968
25.99 Chapter summary in 20 lines	969

26 Packets and Routing 970

26.0 What this chapter gives you	970
26.1 Why data is split up at all	970
26.2 What a packet actually is	973
26.3 The IPv4 header field by field	975
26.4 Fragmentation and MTU	978
26.5 What a router actually does, mechanically	980
26.6 The routing table	983
26.7 How routers learn routes	986
26.8 Autonomous systems	989
26.9 BGP, the protocol that glues the internet together	991
26.10 Why nobody knows the whole path in advance	995
26.11 Load balancing inside the network	997
26.12 Quality of service and the real world	999
26.13 Reading a packet capture	1001
26.98 Common wrong ideas	1003
26.99 Chapter summary in 20 lines	1003

27 Traceroute Decoded — Reading Your Own Path 1005

27.0 What this chapter gives you	1005
27.1 Traceroute is a clever trick, not a feature	1005
27.2 The time to live field, properly	1008
27.3 The mechanism, step by step	1011
27.4 The variants, and which one to use	1013
27.5 Why one hop shows several addresses	1016
27.6 Stars, and how to read them honestly	1018
27.7 Why the times look strange	1022
27.8 What a traceroute cannot tell you	1024
27.9 The reader's trace, decoded hop by hop	1027
27.10 Reading carrier router names	1029
27.11 The trace as a story	1032
27.12 What the silence after hop 12 means	1035
27.13 How to run a much better traceroute	1037
27.98 Common wrong ideas	1039
27.99 Chapter summary in 20 lines	1040

28 TCP, Ports, and Why a Timeout Is Evidence 1041

28.0 What this chapter gives you	1041
28.1 What IP alone gives you, and what it does not	1041
28.2 TCP and UDP: two ways to use IP	1043
28.3 Ports, sockets, and the four-tuple	1046
28.4 The three-way handshake	1048
28.5 The TCP state machine and the four-way close	1051
28.6 Making TCP reliable and fast	1054
28.7 Congestion control in plain words	1056
28.8 The three outcomes of a connection attempt	1059
28.9 What the reader's timeout actually proves	1062
28.10 Why "the server is down" was ruled out	1065
28.11 Reading curl -v line by line	1067
28.12 The other tools that answer this question	1070
28.13 Sockets from the programmer's side	1071
28.98 Common wrong ideas	1074
28.99 Chapter summary in 20 lines	1074

29 Why Mobile Data Worked, and the Diagnostic Method 1076

29.0 What this chapter gives you	1076
29.1 "The site is down" and "I cannot reach the site"	1076
29.2 Everything that changed when the phone used mobile data	1079
29.3 The logic of a controlled experiment on a network	1083
29.4 The diagnostic ladder, rung by rung	1086
29.5 The reader's own fault, mapped onto the ladder	1091
29.6 Cross-checks that multiply your certainty	1095
29.7 Ping, honestly	1098
29.8 The commands in full, for macOS	1101
29.9 The decision tree for "I cannot reach a site"	1105
29.10 Writing up a fault so someone else can act	1107
29.11 When it is not your problem	1109
29.12 Intermittent faults, the hardest kind	1111
29.13 What to do next in this specific situation	1114
29.98 Common wrong ideas	1116
29.99 Chapter summary in 20 lines	1117

30 How the Internet Was Built 1118

30.0 What this chapter gives you	1118
30.1 Before electricity: fire, flags and towers	1118
30.2 The electric telegraph	1120
30.3 The telephone network	1122
30.4 Packet switching, invented three times	1124
30.5 ARPANET	1126
30.6 Email and the applications that came first	1128
30.7 TCP/IP and the flag day	1131
30.8 Other networks and the standards war	1133
30.9 From research network to public internet	1135
30.10 The World Wide Web	1137
30.11 The browser wars	1139
30.12 The dot-com boom and bust	1141
30.13 Broadband and the physical internet	1143
30.14 Who actually runs the internet	1146
30.15 Web 1.0 to now	1148
30.16 A master internet timeline	1150
30.98 Common wrong ideas	1151
30.99 Chapter summary in 20 lines	1152

31 Wireless – WiFi, Bluetooth, Cellular and GPS 1154

31.0 What this chapter gives you	1154
31.1 Radio from first principles	1154
31.2 Carrying information on a wave	1156
31.3 The realities of radio	1159
31.4 Spectrum and regulation	1162
31.5 The WiFi hardware	1164
31.6 WiFi history and standards	1166
31.7 How WiFi actually works on the air	1169
31.8 WiFi security	1172
31.9 The WiFi software stack	1175
31.10 Bluetooth	1178
31.11 Bluetooth Low Energy	1180
31.12 Bluetooth audio	1182
31.13 Cellular	1184
31.14 GPS and positioning	1186
31.15 The other short-range radios	1188
31.98 Common wrong ideas	1190
31.99 Chapter summary in 20 lines	1190

32 TLS, HTTPS and the Web Stack 1192

32.0 What this chapter gives you	1192
32.1 HTTP first, because TLS wraps it	1192
32.2 The three problems TLS solves	1196
32.3 The crypto building blocks, taught gently	1197
32.4 The TLS handshake, in full	1201
32.5 Certificates: what is actually in one	1204
32.6 The chain of trust	1208
32.7 What the browser verifies, and what it does NOT	1211
32.8 Self-signed certificates and the reader's own router	1213
32.9 When TLS is deliberately intercepted	1216
32.10 HTTP versions: 1.1, 2 and 3	1218
32.11 Cookies, sessions, and CORS	1220
32.12 The whole journey, assembled	1223
32.13 CDNs, load balancers and caching in front	1226
32.98 Common wrong ideas	1228
32.99 Chapter summary in 20 lines	1229

33 The OSI and TCP/IP Models, Retrofitted to Your Own Session 1230

33.0 What this chapter gives you	1230
33.1 Why layers exist at all	1231
33.2 The OSI seven-layer model	1233
33.3 The TCP/IP four-layer model	1236
33.4 The reader's own session, layer by layer	1238
33.5 Encapsulation walked through with real numbers	1244
33.6 Where the reader's failure sits in the model	1247
33.7 A fault-to-layer table	1250
33.8 Where the model lies	1252
33.9 Why layering still earns its keep	1254
33.10 The same idea outside networking	1256
33.98 Common wrong ideas	1257
33.99 Chapter summary in 20 lines	1258

34 Firewalls, VPNs, CDNs, IPv6 and the Gaps 1260

34.0 What this chapter gives you	1260
34.1 localhost, 127.0.0.1 and 0.0.0.0	1260
34.2 Ports, properly this time	1264
34.3 Firewalls, properly	1267
34.4 DROP versus REJECT	1270
34.5 Middleboxes, and the end of end-to-end	1274
34.6 VPNs, what they do and what they do not	1277
34.7 Proxies, forward and reverse	1280
34.8 Tunnelling in general	1284
34.9 CDNs and anycast in depth	1287
34.10 Latency versus bandwidth	1293
34.11 IPv6 properly	1297
34.12 Multicast, broadcast and mDNS	1301
34.13 Network performance for developers	1305
34.14 Cloud networking preview	1309
34.98 Common wrong ideas	1312
34.99 Chapter summary in 20 lines	1312

Part G — Building and Shipping Software 1314

35 Web and App Development 1315

35.0 What this chapter gives you	1315
35.1 Frontend, backend and the line between them	1315

35.2 HTML: the structure of a page	1318
35.3 CSS: presentation, and the cascade that confuses everyone	1321
35.4 JavaScript: the language and its event loop	1325
35.5 The DOM and how a browser turns HTML into pixels	1329
35.6 The browser as a platform	1333
35.7 Frontend frameworks, explained rather than listed	1336
35.8 Build tooling: why frontend code needs a build step	1339
35.9 The backend: what a server program actually does	1342
35.10 Databases	1345
35.11 ORMs and data access	1349
35.12 Authentication and sessions	1351
35.13 Application architecture	1357
35.14 Mobile apps	1362
35.15 Testing, performance and accessibility for the web	1367
35.16 A realistic learning path	1372
35.98 Common wrong ideas	1377
35.99 Chapter summary in 20 lines	1378

36 APIs — What They Are and Where They Came From 1380

36.0 What this chapter gives you	1380
36.1 What an API actually is	1380
36.2 The history, properly	1383
36.3 Local APIs: signatures, libraries and versions	1387
36.4 System call APIs	1390
36.5 Remote APIs: what the network changes	1393
36.6 HTTP APIs in practice	1396
36.7 REST, properly	1399
36.8 The alternatives, compared honestly	1402
36.9 Authentication and authorization for APIs	1404
36.10 Making an API reliable in the real world	1408
36.11 Webhooks: the inversion where the server calls you	1414
36.12 Designing an API people can actually use	1419
36.13 Consuming an API well	1425
36.14 APIs as a business	1431
36.98 Common wrong ideas	1435
36.99 Chapter summary in 20 lines	1436

37 Git — Why It Works With No Network 1438

37.0 What this chapter gives you	1438
37.1 What version control is for	1438
37.2 Centralized versus distributed	1441
37.3 Why Git was made	1445
37.4 What is actually inside a .git directory	1448
37.5 What “the whole history” really means	1451
37.6 What “origin” actually is	1454
37.7 Remote-tracking refs and staleness	1458
37.8 Which git operations need the network	1461
37.9 The reader’s outage, worked through	1464
37.10 The trade-offs of the distributed model	1467
37.11 Git is not GitHub	1470
37.12 Getting started properly	1473
37.98 Common wrong ideas	1476
37.99 Chapter summary in 20 lines	1477

38 What a Commit Actually Is 1479

38.0 What this chapter gives you	1479
38.1 Content-addressed storage, the central idea	1479
38.2 Hashing, and the honest truth about SHA-1	1482
38.3 The four object types	1485

38.4 How the four types compose	1489
38.5 Why a commit points at its parent	1493
38.6 What a branch really is	1496
38.7 HEAD	1499
38.8 The rest of the refs	1502
38.9 The reflog	1505
38.10 The commands that reveal the machinery	1508
38.11 How git stores things efficiently	1511
38.12 Reading the reader's own situation	1515
38.13 Why this design is good, and what it costs	1518
38.98 Common wrong ideas	1521
38.99 Chapter summary in 20 lines	1522

39 The Index, Push, Fetch, Merge and Rebase 1523

39.0 What this chapter gives you	1523
39.1 The three states and three places	1523
39.2 What the index actually is	1527
39.3 What git add really does	1530
39.4 What git commit really does	1533
39.5 Undoing things, laid out clearly	1536
39.6 What actually crosses the wire on a push	1539
39.7 The reader's own failure, analysed properly	1543
39.8 Fetch versus pull	1547
39.9 Merge	1549
39.10 Conflicts at the file level	1552
39.11 Rebase	1556
39.12 The reader's rebase, explained	1559
39.13 When rebasing is safe and when it destroys work	1561
39.14 Merge versus rebase as a team policy	1564
39.15 Cherry-pick, revert, bisect and the rest	1566
39.98 Common wrong ideas	1568
39.99 Chapter summary in 20 lines	1569

40 Your Own Git Server, and What GitHub Adds 1571

40.0 What this chapter gives you	1571
40.1 A git server is just another repository	1571
40.2 Bare repositories	1574
40.3 Doing it end to end	1577
40.4 Remotes in depth	1582
40.5 Hooks: server-side and client-side	1586
40.6 What GitHub actually is	1592
40.7 What GitHub adds that plain git does not have	1594
40.8 The exact boundary, using the reader's own outage as evidence	1599
40.9 The self-hosted alternatives	1603
40.10 Migrating and mirroring	1607
40.11 Working with others, practically	1611
40.12 Monorepos and large repositories	1616
40.13 Backing up a git repository properly	1621
40.98 Common wrong ideas	1625
40.99 Chapter summary in 20 lines	1626

41 CI/CD and What a "Run" Really Is 1628

41.0 What this chapter gives you	1628
41.1 What continuous integration means and the problem it solves	1628
41.2 Continuous delivery and continuous deployment	1631
41.3 What a runner actually is	1633
41.4 The workflow file	1636
41.5 Triggers: what starts a run	1640
41.6 The most important idea: a run belongs to a commit, not a branch	1645

41.7 Why verifying by run ID against a head SHA is the stronger claim	1649
41.8 Statuses, checks, and the two APIs behind them	1653
41.9 Merge queues	1657
41.10 mergeStateStatus, value by value	1661
41.11 What a good pipeline actually does, in order	1665
41.12 Deployment strategies	1668
41.13 Secrets in CI	1672
41.14 Reading a failed run like an engineer	1676
41.15 The wider landscape	1679
41.98 Common wrong ideas	1683
41.99 Chapter summary in 20 lines	1683

42 SSH, HTTPS, Keys, Tokens and Scopes 1685

42.0 What this chapter gives you	1685
42.1 The two transports git uses, and why there are two	1685
42.2 What SSH actually is	1689
42.3 Public key cryptography for SSH	1692
42.4 authorized_keys and how it works	1697
42.5 known_hosts and host key verification	1701
42.6 The SSH agent	1705
42.7 Why SSH on port 22 can be blocked when 443 is not	1709
42.8 HTTPS authentication for git	1713
42.9 Personal access tokens	1717
42.10 Scopes, properly	1721
42.11 OAuth, tokens, SSH keys and GitHub Apps	1725
42.12 Practical hardening for a developer's own machine	1728
42.13 A diagnosis table for git authentication failures	1732
42.98 Common wrong ideas	1735
42.99 Chapter summary in 20 lines	1736

43 Cloud Computing, the Bezos Mandate, AWS and Azure 1738

43.0 What this chapter gives you	1738
43.1 What the cloud actually is	1738
43.2 Inside a real datacentre	1741
43.3 Regions, availability zones and edge locations	1745
43.4 Virtualization, the technology that made cloud possible	1748
43.5 Containers	1751
43.6 Kubernetes and orchestration	1755
43.7 The Bezos API mandate	1759
43.8 How AWS actually started	1762
43.9 The AWS services you actually need to know	1765
43.10 Azure, Google Cloud and the rest	1768
43.11 The service models: IaaS, PaaS, SaaS and FaaS	1771
43.12 Serverless, properly	1773
43.13 Storage in the cloud	1776
43.14 Cloud networking	1779
43.15 Identity and access	1783
43.16 What cloud costs and how bills explode	1786
43.17 DevOps, infrastructure as code and the modern workflow	1789
43.18 Microservices versus monolith, decided honestly	1793
43.98 Common wrong ideas	1796
43.99 Chapter summary in 20 lines	1797

44 Security 1799

44.0 What this chapter gives you	1799
44.1 What security actually means	1799
44.2 Hashing	1803
44.3 Password storage done properly	1806
44.4 Symmetric encryption	1810

44.5 Asymmetric encryption, signatures and the quantum question	1813
44.6 Authentication and identity	1817
44.7 Authorization and access control	1820
44.8 The web application vulnerability classes, defensively	1823
44.9 Memory safety	1826
44.10 Network attacks conceptually	1829
44.11 Malware and how infection actually happens	1832
44.12 Social engineering	1835
44.13 Supply chain security	1837
44.14 Defence in depth for a real system	1840
44.15 Detection and response	1842
44.16 Privacy and the law, briefly	1845
44.17 Security for the individual developer	1847
44.18 Famous breaches and what each teaches	1849
44.98 Common wrong ideas	1851
44.99 Chapter summary in 20 lines	1852

Part H — Games and Machine Intelligence

1854

45 Games — How They Are Made and How They Run

1855

45.0 What this chapter gives you	1855
45.1 What a game actually is as a program	1855
45.2 A history of video games as technology	1858
45.3 The game loop in detail	1862
45.4 How a game is built, as a project	1867
45.5 Game engines	1870
45.6 Assets and the content pipeline	1873
45.7 Physics and collision	1876
45.8 Game AI	1879
45.9 Rendering a game frame, specifically	1882
45.10 Audio in games	1885
45.11 Multiplayer and netcode	1887
45.12 Cheating and anti-cheat	1891
45.13 What actually happens when you double-click a game	1894
45.14 Consoles versus PC	1897
45.15 Emulation	1900
45.16 Making your first game, practically	1902
45.98 Common wrong ideas	1905
45.99 Chapter summary in 20 lines	1906

46 Machine Learning From Scratch — Fitting, Loss, Gradients and Neural Networks

1907

46.0 What this chapter gives you	1907
46.1 What machine learning actually is	1907
46.2 A short honest history	1911
46.3 Fitting a line, the foundation of everything	1914
46.4 Loss: how you measure being wrong	1918
46.5 Gradient descent: walking downhill in fog	1921
46.6 The vocabulary of training	1926
46.7 Classical machine learning before neural networks	1930
46.8 The neuron	1933
46.9 Activation functions	1936
46.10 Layers and depth	1940
46.11 The forward pass as matrix multiplication	1942
46.12 Backpropagation	1945
46.13 Training in practice	1949
46.14 The families of neural network	1953
46.15 What a trained model physically is	1956
46.98 Common wrong ideas	1959

46.99 Chapter summary in 20 lines	1959

47 Parameters, Tokens and Embeddings 1961

47.0 What this chapter gives you	1961
47.1 What a parameter actually IS in a language model	1961
47.2 What 7B, 70B and 405B actually count	1964
47.3 Parameters versus the four things they get confused with	1968
47.4 Parameters, memory and precision	1971
47.5 Quantization	1975
47.6 Why capability does not scale linearly with parameter count	1979
47.7 Tokens: why text is not fed in as words or characters	1982
47.8 What tokenization does in practice	1986
47.9 Token economics	1989
47.10 Embeddings: turning a token id into a vector	1992
47.11 What embeddings capture	1995
47.12 Embeddings as a tool in their own right	1999
47.13 The context window	2002
47.14 Putting it together: one prompt from text to numbers	2005
47.98 Common wrong ideas	2007
47.99 Chapter summary in 20 lines	2008

48 The Transformer 2010

48.0 What this chapter gives you	2010
48.1 What came before, and why it was not enough	2010
48.2 Attention in plain language, before any mathematics	2014
48.3 Query, key and value, taught slowly	2018
48.4 Scaled dot-product attention, with the actual mathematics	2021
48.5 Masking: stopping the model reading ahead	2026
48.6 Multi-head attention	2030
48.7 Position: attention has no idea what order things are in	2034
48.8 The rest of the block: feed-forward, residuals and normalization	2039
48.9 The whole model, assembled	2043
48.10 The 2017 paper	2046
48.11 The three architectures	2050
48.12 Generation: producing text one token at a time	2052
48.13 The cost of attention: the quadratic problem	2056
48.14 Mixture of experts: paying for a big model, running a small one	2060
48.15 Why one architecture does so many tasks	2067
48.98 Common wrong ideas	2073
48.99 Chapter summary in 20 lines	2074

49 How a Model Is Actually Made — Pretraining to Release 2076

49.0 What this chapter gives you	2076
49.1 The pipeline in one view	2076
49.2 The data: where it comes from and how it is cleaned	2079
49.3 Pretraining itself	2084
49.4 Scaling laws: how big, how much data	2087
49.5 What a base model actually is	2091
49.6 Supervised fine-tuning	2094
49.7 Learning from preferences	2097
49.8 What each stage actually changes	2101
49.9 Efficient adaptation: LoRA, QLoRA and distillation	2103
49.10 What a training run looks like physically	2106
49.11 What weights physically are	2110
49.12 Open weights versus open source	2114
49.13 The Open Source Initiative's position	2117
49.14 Model releases in practice	2120
49.98 Common wrong ideas	2123
49.99 Chapter summary in 20 lines	2124

50 Benchmarks, Evaluation, Versions and Dates 2126

50.0 What this chapter gives you	2126
50.1 What a benchmark is, mechanically	2126
50.2 The major benchmarks, one at a time	2130
50.3 Actually running an evaluation	2136
50.4 Why two labs report different numbers	2140
50.5 Contamination	2143
50.6 The other reasons to be sceptical	2146
50.7 The alternatives, and their own weaknesses	2150
50.8 What “eval” means inside a company	2153
50.9 Reading a model announcement critically	2156
50.10 Training cutoff versus release date	2158
50.11 Versions, snapshots and the moving-name problem	2160
50.12 Deprecation and retirement	2162
50.13 What you should actually do	2164
50.98 Common wrong ideas	2167
50.99 Chapter summary in 20 lines	2167

51 Serving, Running One Yourself, RAG, Agents and the Limits 2169

51.0 What this chapter gives you	2169
51.1 Inference versus training: what changes	2169
51.2 What an API call actually contains	2171
51.3 How generation actually performs	2174
51.4 The serving optimizations, each explained	2177
51.5 Cost and capacity	2179
51.6 Running one yourself, practically	2181
51.7 What hardware you actually need	2184
51.8 Getting weights and reading a model card	2186
51.9 Building something small from scratch, honestly	2189
51.10 Prompting as an engineering practice, not a trick	2191
51.11 RAG: giving a model your documents	2193
51.12 Tool use and function calling	2196
51.13 Agents: what the word actually means	2198
51.14 Memory, and why a model with none appears to have some	2200
51.15 The limits, stated plainly	2202
51.16 The gaps you did not ask about but need	2204
51.17 A practical decision guide	2206
51.98 Common wrong ideas	2208
51.99 Chapter summary in 20 lines	2209

Part I — The Reference Desk 2210

52 The Engineer’s Reference 2211

52.0 How to use this chapter	2211
52.1 Numbers every engineer should know	2212
52.2 Units and conversions	2217
52.3 Number bases and encoding	2220
52.4 The networking reference	2223
52.5 Networking command cheat sheet	2232
52.6 Decision tree: I cannot reach a site	2239
52.7 Git command cheat sheet	2243
52.8 Decision tree: my git operation failed	2247
52.9 Shell and system cheat sheet	2251
52.10 File format identification by magic number	2254
52.11 Complexity and performance reference	2255
52.12 Cloud and infrastructure reference	2257
52.13 Security reference	2260
52.14 AI and machine learning reference	2262

52.15 Regular expressions	2265
52.16 Keyboard shortcuts and editor basics	2266
52.17 What to do when you are stuck	2267
52.98 Common wrong ideas	2269
52.99 Chapter summary in 20 lines	2270

53 Master Timeline and Glossary 2272

53.0 How to use this chapter	2272
53.1 The master timeline of computing	2272
53.2 Timeline of programming languages	2284
53.3 Timeline of standards and protocols	2286
53.4 The glossary	2289
53.5 Where to go next	2321
53.6 The whole book in 100 lines	2322



PART A

The Physical World

Rock, electricity, silicon, transistors, logic and memory.
Everything a computer is made of, before any software
exists.

The Whole Machine in One Look

1.0 What this chapter gives you

1. You will be able to say what a computer actually is, with nothing magical left in it.
2. You will name the five parts every computer has ever had, and find them in a 1945 machine and in a watch.
3. You will explain the one idea that made computers general-purpose, and name the 1945 document that wrote it down.
4. You will draw the layer cake from rock to artificial intelligence, and say what each layer hides.
5. You will trace one key press to one letter on screen, through about thirty steps.
6. You will feel the real speed gaps inside a machine, because we will stretch nanoseconds into human time.
7. You will stop being confused by KB versus KiB, by MB/s versus Mb/s, and by a 1 TB drive that reports 931 GB.
8. You will know where hardware stops and software starts, and why that border keeps moving.
9. You will have a map of all 53 chapters and a method for reading them.

1.1 What a computer actually is

■ 1.1.1 PLAIN - in simple words

1. A computer is a machine that moves and changes patterns of on and off.
2. "On" and "off" mean a wire carries a higher or a lower voltage. Voltage is electrical push.
3. We write on as 1 and off as 0. One 1 or 0 is a bit.
4. The machine does not know whether a pattern is a photo, a song or a bank balance.
5. Meaning is given by us, not by the metal.

■ 1.1.2 PLAIN - a picture in your head

1. Picture a long corridor with a million light switches on the wall.
2. Now imagine a rule-follower walking that corridor at enormous speed.
3. He carries one card: "if switch 7 is up and switch 8 is up, put switch 9 up".
4. He never thinks and never decides. Run him fast enough and the pattern becomes a spreadsheet, a call, a game.
5. Where this comparison breaks: there is no walker. The rules are in the wiring, and huge numbers of switches settle at the same instant.

■ 1.1.3 PLAIN – a worked example

1. Take four switches: S3, S2, S1, S0. We agree S3 is worth 8, S2 is worth 4, S1 is worth 2, S0 is worth 1.

S3	S2	S1	S0	value
0	0	0	0	0
0	0	1	1	3
0	1	0	1	5
1	0	0	1	9
1	1	1	1	15

2. Now we want 5 plus 3. One group is set to 0101, another to 0011, and both feed a small adding circuit.
3. That circuit has no memory and no cleverness. Its output settles to 1000, which is 8.
4. Nobody told the circuit this was arithmetic. Its wiring forced the result.

■ 1.1.4 PLAIN – what is really happening inside

1. Inside a chip, a bit is a voltage held on a tiny wire by a silicon switch called a transistor.
2. A transistor is a tap: a small voltage on one leg controls current between the other two.
3. A few transistors make a gate, a circuit applying one logical rule. Millions of gates make an adder or a memory cell.
4. Voltages take time to settle, because every wire has capacitance, the tendency to store charge and resist sudden change.
5. So the machine has a clock, a square wave ticking billions of times a second. On each tick, settled results are captured.
6. The honest version: a big 2026 chip is not one clock. It has many clock domains at different rates, and some blocks are deliberately clockless. One global heartbeat is a teaching simplification.

■ 1.1.5 TECHNICAL – the engineer’s version

1. Mainstream digital logic is CMOS, complementary metal-oxide-semiconductor, using paired n-channel and p-channel field-effect transistors.
2. A logic level is a voltage band, not a point. Above V_{IH} is 1, below V_{IL} is 0, and the gap between them is forbidden in steady state.

Rail	Typical voltage	Where it appears
CPU core, 2026	0.6 to 1.2 V	logic on the die
DDR5 VDD	1.1 V	main memory
LVC MOS 3.3 V, V_{IH}	2.0 V minimum	input read as 1
LVC MOS 3.3 V, V_{IL}	0.8 V maximum	input read as 0

3. Claude Shannon’s 1937 master’s thesis at MIT showed that relay switching circuits implement Boolean algebra. That is the formal bridge from wires to logic.
4. Alan Turing’s 1936 paper “On Computable Numbers” defined what a general computing machine can compute, before any such machine existed.
5. Observation tools: `lscpu` and `dmidecode` report clocks and memory type; `turbostat` and `powermetrics` report frequency and power live.

■ 1.1.6 WORDS – remember these

1. Bit — one on-or-off — a binary digit, value 0 or 1, the smallest unit of information.
2. Voltage — electrical push — potential difference in volts, the physical carrier of a logic level.
3. Transistor — an electrical tap — a semiconductor switch, in CMOS a field-effect transistor.
4. Clock — the machine's heartbeat — a periodic square wave defining when state elements sample inputs.

1.2 The five parts of every computer ever built

■ 1.2.1 PLAIN – in simple words

1. Every computer, from 1945 to the one on your wrist, has the same five parts.
2. Input: how the outside world gets patterns into the machine.
3. Memory: fast temporary space holding what the machine is working on right now.
4. Storage: slow permanent space that keeps things when power is off.
5. Processing: the part that changes patterns by rules, called the processor or CPU.
6. Output: how patterns get back out where you can sense them.
7. There is no sixth part. Memory forgets when power stops; storage remembers. Confusing those two causes more beginner errors than anything else.

■ 1.2.2 PLAIN – a picture in your head

1. Think of a kitchen. Shopping bags coming through the door are input.
2. The countertop is memory: small, close to hand, cleared at the end of the night.
3. The fridge and cupboard are storage: bigger, slower to reach, still full tomorrow.
4. The cook is processing, the only part that changes anything. The plate carried to the table is output.
5. Where this comparison breaks: a cook holds an idea in his head while walking to the fridge. A processor cannot. It has a few tiny slots called registers, and everything else must come back through memory.

■ 1.2.3 PLAIN – a worked example

1. The five parts in a 1945 machine and in an ordinary desktop.

Part	ENIAC, 1945	Desktop, 2026
Input	punched cards, switches	keyboard, mouse
Memory	20 accumulators	16 to 64 GB of RAM
Storage	punched cards, paper	1 to 4 TB NVMe SSD
Processing	tube arithmetic units	multi-core CPU
Output	punched cards, lamps	monitor, speakers

2. The same five parts in a phone and in a smartwatch.

Part	Phone	Smartwatch
Input	touch, mics, cameras	touch, crown, sensors
Memory	6 to 16 GB RAM	small, not published
Storage	128 GB to 1 TB flash	64 GB flash
Processing	CPU, GPU, NPU on a die	low-power SiP
Output	OLED screen, speaker	OLED screen, buzzer

■ 1.2.4 PLAIN – what is really happening inside

1. The processor never works directly on storage. Storage is far too slow and is not wired for it.
2. So the life of any data is: storage to memory, memory to processor, processor to memory, memory to storage.
3. Output runs in reverse: the processor writes numbers into a region of memory, and a device reads that region and makes light or sound.
4. The processor never touches a pixel. It writes numbers. Something else turns numbers into light.

■ 1.2.5 TECHNICAL – the engineer's version

1. ENIAC was built at the Moore School of Electrical Engineering, University of Pennsylvania, by J. Presper Eckert and John Mauchly. It first ran productively on 10 December 1945 and was announced publicly on 14 February 1946.
2. ENIAC used about 18,000 vacuum tubes, weighed more than 30 short tons, drew about 150 kW, and did roughly 5,000 additions per second. It was programmed by plugboard wiring, which could take days.
3. The Apple Watch Series 10, announced September 2024, uses the S10 SiP with a 64-bit dual-core processor, 64 GB of storage, and a 416 by 496 pixel display in the 46 mm size.
4. Observation tools: `lsblk` and `diskutil list` show storage; `free -h` and `vm_stat` show memory; `lspci` and `lsusb` show what sits on the buses.

■ 1.2.6 WORDS – remember these

1. Input — how things get in — any device converting an external event into machine-readable data.
2. Memory — the countertop — volatile working store, today DRAM, addressed byte by byte.
3. Storage — the cupboard — non-volatile store, today NAND flash or magnetic disk, addressed in blocks.
4. Processing — the cook — the CPU and co-processors that fetch, decode and execute instructions.
5. Output — how things get out — any device turning data into a signal a person or machine can sense.
6. Bus — a bundle of shared wires — a defined interface carrying address, data and control signals.

1.3 The one big idea: instructions and data are the same stuff

■ 1.3.1 PLAIN – in simple words

1. The list of steps a machine should follow is itself just a pattern of bits.
2. So it can live in the same memory, in the same kind of slots, as the numbers being worked on.
3. Before this idea the steps lived in the wiring. To change the job you rewired the machine.
4. After it, you load a different pattern into memory. That takes a moment, not a week. This is the stored-program idea.
5. The same idea is why a bad pattern arriving from outside can sometimes be run as instructions.
6. One idea gives you both general-purpose computing and computer viruses. They are the same coin.

■ 1.3.2 PLAIN – a picture in your head

1. Imagine a kitchen where recipe cards sit in the same drawer as shopping lists, on identical cards in the same handwriting.
2. A card is a recipe only because the cook happened to pick it up and follow it.
3. Where this comparison breaks: a human cook would notice that a shopping list is not a recipe and stop.

4. A processor does not notice. It will read your holiday photo as instructions and execute it, producing noise or a crash.

■ 1.3.3 PLAIN – a worked example

1. Consider one byte with the pattern 01001000. In base 16 that is 48. In decimal it is 72.

```
byte 01001000 (hex 48, decimal 72)

read as a number      -> 72
read as ASCII text    -> the letter H
read as x86 machine   -> part of a REX prefix
read as a pixel value -> a mid-grey level
read as an audio byte -> one sample, fairly loud
```

2. The bits never change. Only the interpretation changes.
3. The program counter is the register holding “where I am reading instructions from”.
4. If it points at that spot, the processor will try to execute those bytes as an instruction, and usually crash within microseconds.
5. Nothing in the memory chip distinguishes the two cases. The only difference is which register pointed at the address.

■ 1.3.4 PLAIN – what is really happening inside

1. The processor keeps a program counter holding the address of the next instruction.
2. Each cycle it reads the bits there, works out what they mean, does it, and advances the counter. That loop is fetch, decode, execute.
3. Because instructions are ordinary bytes, an attacker who can write memory and redirect the counter runs code of their choosing.
4. So operating systems mark memory pages writable or executable and try never to allow both at once.
5. The honest version: “never both at once” is the goal, not the reality. Just-in-time compilers must briefly have both, and the whole field of exploit mitigation lives in that gap.

■ 1.3.5 TECHNICAL – the engineer’s version

1. The canonical statement is the “First Draft of a Report on the EDVAC”, circulated 30 June 1945 under John von Neumann’s name alone.
2. It drew on work by Eckert and Mauchly, and credit for it is still disputed by historians.
3. The structure it describes, one memory holding both instructions and data reached over one path, is the **von Neumann architecture**.

Machine	First stored program	People
ENIAC, converted	April 1948	Clippinger, von Neumann
Manchester Baby	21 June 1948	Kilburn, Williams
EDSAC, Cambridge	6 May 1949	Maurice Wilkes
EDVAC	1951 in service	Eckert, Mauchly

4. The hardware defence against executing data is the no-execute page bit: AMD shipped NX from 2003, Intel calls it XD, Arm calls it XN. The operating system policy built on it is W xor X.
5. Observation tools: `objdump -d` disassembles bytes as instructions; `xxd` shows the same bytes as hex and text together; `/proc/<pid>/maps` shows which regions are executable.

■ 1.3.6 WORDS – remember these

1. Stored program — the steps live in memory like any other data — instructions encoded in the same address space as operands.
2. Program counter — a bookmark — the register holding the address of the next instruction.
3. von Neumann architecture — one memory for everything — a single address space and data path for instructions and data.
4. Just-in-time compilation — building instructions while running — generating native code at run time and jumping into it.
5. NX bit — a “do not run this” mark — a page-table attribute preventing instruction fetch from a page.

1.4 The layer cake

■ 1.4.1 PLAIN – in simple words

1. At the bottom is rock: sand, which is silicon dioxide, purified into silicon crystal. Above it is electricity, charge pushed along wires.
2. Above electricity are transistors, tiny switches made in the silicon. Above them are gates, small groups of switches applying one rule each.
3. Above gates is the processor, millions of gates arranged to fetch and obey instructions, and above that is machine code, the bit patterns it obeys.
4. Above machine code is the operating system, which shares the machine among other programs, and then the programs you use.
5. Above the programs come networks, then the cloud, which is other people’s machines rented by the hour, then artificial intelligence services.
6. Hiding the layer below is called abstraction. Its promise is that you can use one layer without knowing the one under it.
7. The promise is never fully kept. Details leak upward, and most hard bugs live in a leak.

■ 1.4.2 PLAIN – a picture in your head

1. Think of a tall building where each floor was built by a different team.
2. The team on floor six is told only that the floor holds their weight and there is a lift.
3. But on a windy day floor six sways, and on a hot day the lift is slow. Nobody promised that. Lower floors are leaking through.
4. Where this comparison breaks: floors are physically separate; computer layers are not.
5. There is no wall between the operating system and the processor. The separation is an agreement in documents, enforced only partly by hardware.

■ 1.4.3 PLAIN – a worked example

AI services	<- text in, text out
cloud	<- machines by the hour
networks	<- messages to elsewhere
programs	<- what you actually use
operating system	<- sharing, files, safety
machine code	<- what the CPU obeys
CPU	<- fetch, decode, execute
logic gates	<- one rule per circuit
transistors	<- one switch each
electricity	<- charge that moves
silicon / rock	<- purified sand

1. Now the honest part: what each layer hides, and what leaks through anyway.

Layer	Hides	Leaks anyway
Transistors	atoms, physics	heat, wear, errors
Gates	voltages, timing	propagation delay
CPU	gate wiring	cache misses, branches
Machine code	circuits	which CPU family is needed
Operating system	hardware detail	file locks, page faults
Programs	the OS	memory limits, crashes
Networks	distance	latency, packet loss
Cloud	the building	region choice, outages
AI services	the model	token limits, wrong answers

2. When a web page in India loads slowly, the network layer is leaking the speed of light through five layers of abstraction into a person's experience.

■ 1.4.4 PLAIN – what is really happening inside

1. Each boundary between layers is a contract listing what you may ask for and what you get back.
2. A contract that says what happens and never how is a clean abstraction. A leak happens when the “how” shows through timing, limits or failure.
3. Timing leaks are the worst, because the contract is still honoured. You get the right answer, just late. A cache miss is exactly that.
4. The honest version: layers are not a neat pile. A graphics driver reaches almost straight to hardware.
5. A hypervisor sits under an operating system that believes it is on bare metal. The cake is a teaching device; the real thing is a tangle.

■ 1.4.5 TECHNICAL – the engineer's version

1. The processor-software contract is the instruction set architecture, ISA: x86-64 from Intel and AMD, Arm A-profile at Armv9, and RISC-V, an open specification maintained by RISC-V International since 2015.
2. Below the ISA sits microarchitecture: pipelines, caches, branch predictors, out-of-order execution. It is invisible in the contract and visible on the stopwatch.
3. Joel Spolsky named the general phenomenon in his 2002 essay “The Law of Leaky Abstractions”: all non-trivial abstractions, to some degree, are leaky.
4. The most expensive proof arrived in January 2018 with Spectre and Meltdown, published by teams including Google Project Zero and Graz University of Technology.
5. Speculative execution, purely a microarchitectural optimization, leaked memory across the security boundary the ISA had promised.
6. To keep the three kinds of claim apart: x86-64 instruction encoding and POSIX system call names are **standards**; the layer diagram above is a **convention**; the L3 cache size on one chip is an **implementation detail**.
7. Observation tools: `perf stat` counts cache misses and branch mispredictions; `strace` and `dttruss` log system calls; `mttr` shows where network latency accumulates.

■ 1.4.6 WORDS – remember these

1. Abstraction — hiding the mess below — an interface exposing behaviour while concealing implementation.
2. Leaky abstraction — the mess showing through — an interface whose implementation is observable via timing, limits or failure modes.
3. Instruction set architecture — the deal between chip and software — the specified instructions, registers and memory model.
4. Protocol — an agreed way to talk — a specification of message formats and exchange rules between systems.

1.5 A journey in one page: you press the letter A

■ 1.5.1 PLAIN – in simple words

1. You press A. The letter appears. It feels instant and it feels like one event.
2. It is not one event. It is about thirty handovers across nine layers.
3. A physical push becomes an electrical contact, becomes a number, becomes a message, becomes a shape, becomes light.
4. This section is the trailer. Chapter 21 does the whole journey properly, with real timings and real code paths.

■ 1.5.2 PLAIN – a picture in your head

1. Think of a relay race with thirty runners, each carrying a baton a short distance.
2. Runner one carries “a key moved”. Runner ten carries “key number 4 went down”. Runner twenty carries “the character A”. Runner thirty carries “these pixels are white”.
3. Where this comparison breaks: several runners are actually queued behind other work.
4. And some handovers are not a pass at all. They are a note left in a shared box that the next runner checks periodically.
5. That periodic checking, called polling, is where a surprising amount of the delay comes from.

■ 1.5.3 PLAIN – a worked example

1. You move your finger. Layer: physics.
2. The key cap pushes a switch and two metal contacts touch. Layer: mechanics.
3. The contacts bounce, opening and closing several times in a few milliseconds. Layer: physics.
4. A small processor in the keyboard ignores the bounce by waiting. This is debouncing. Layer: firmware.
5. That processor scans a grid of rows and columns thousands of times a second to find which key is down. Layer: firmware.
6. It works out a position number for the key, called a scan code. Layer: firmware.
7. It builds a small report listing which keys are held. Layer: USB HID protocol.
8. The report waits until the host asks for it, on a fixed schedule. Layer: USB bus.
9. The host controller chip receives the report as electrical pulses. Layer: hardware.
10. That chip writes the bytes into main memory itself, without the CPU. This is direct memory access. Layer: hardware.
11. The chip raises an interrupt, a signal meaning “stop and look”. Layer: hardware.
12. The CPU finishes its current instruction, saves its place, and jumps to a fixed address. Layer: CPU.
13. That address holds the interrupt handler inside the operating system kernel. Layer: kernel.
14. The USB driver reads the report and turns the scan code into a kernel key code. Layer: driver.

15. The input subsystem timestamps the event and puts it in a queue. Layer: kernel.
16. The window system wakes, reads the queue, and asks which window has focus. Layer: display server.
17. The keyboard layout is applied. The same key is A on a US layout and Q on a French AZERTY layout. Layer: keymap.
18. Modifier state is applied: shift gives capital A, control gives a control character. Layer: keymap.
19. The event is delivered to the focused application as a message. Layer: operating system.
20. The application's event loop, a loop that waits for messages forever, picks it up. Layer: program.
21. The text widget inserts the character into its internal buffer. Layer: program.
22. The character is stored as a number: Unicode code point 65, encoded in UTF-8 as the single byte 0x41. Layer: text encoding.
23. The program marks the text area as needing redraw. Layer: program.
24. The text shaping engine picks the font, finds the glyph for A, and works out its position. Layer: text stack.
25. The rasterizer turns the glyph outline, a set of curves, into a grid of coverage values. Layer: font engine.
26. Those values become pixel colours blended with the background. This is anti-aliasing. Layer: graphics.
27. The pixels are written into a block of memory called a framebuffer, often by the GPU. Layer: GPU.
28. The compositor combines every window's framebuffer into one final image. Layer: display server.
29. The display controller reads that image out line by line and sends it over the cable. Layer: display hardware.
30. The panel's driver chips set the voltage on each subpixel. Light changes. You see A. Layer: physics.

■ 1.5.4 PLAIN – what is really happening inside

1. Time budget matters more than step count. Most of those thirty steps cost far less than a microsecond.
2. The expensive parts are the waits, not the work.
3. Waiting for the keyboard to be polled: up to 8 milliseconds on an old 125 Hz USB device, about 1 millisecond on a 1000 Hz gaming keyboard.
4. So the honest total, finger to photon, is usually 20 to 60 milliseconds, of which perhaps 50 to 200 microseconds is CPU work.
5. That is a general truth worth carrying: the machine is mostly waiting, and speeding up the work rarely helps as much as removing a wait.

■ 1.5.5 TECHNICAL – the engineer's version

1. Keyboards speak USB HID, Human Interface Device, defined in the USB HID class specification version 1.11 from 2001. A boot-protocol keyboard report is 8 bytes: modifier byte, reserved byte, six key codes.
2. Scan code, key code, keysym and character are four distinct things, and confusing them is a classic bug: a hardware position, a kernel-normalized position, a layout-resolved symbol, and a Unicode value.

Stage	Typical time
Debounce in keyboard	1 to 5 ms
USB poll wait	1 to 8 ms
Kernel and driver	under 100 us

Stage	Typical time
App and compositor	1 to 10 ms
Wait for vsync	8 to 17 ms
Panel response	1 to 20 ms

3. Observation tools: `evtest` and `libinput debug-events` show raw input events; `xev` shows keysyms; `showkey -s` prints scan codes.

■ 1.5.6 WORDS - remember these

1. Debounce — ignoring the rattle of a switch — filtering multiple contact transitions from one mechanical actuation.
2. Polling — checking regularly instead of being told — the host querying a device on a fixed interval.
3. Interrupt — a tap on the shoulder — a hardware signal making the CPU suspend execution and run a handler.
4. Direct memory access — the device writes to memory itself — a controller moving data to RAM without CPU involvement.
5. Framebuffer — the picture waiting to be shown — a memory region holding pixel values for one display image.

1.6 What “fast” and “big” mean here

■ 1.6.1 PLAIN - in simple words

1. Inside a computer there is no single speed. There is a ladder, and the rungs are enormously far apart.
2. The processor is fast. Everything it talks to is slower, and each step outward is slower again.
3. So we do one trick. We pretend one nanosecond, a billionth of a second, lasts one whole second.
4. On that scale a coffee break becomes a lifetime, and the shape of the ladder becomes unforgettable.

■ 1.6.2 PLAIN - a picture in your head

1. You are at a desk and you need one fact. If it is already in your head, one second.
2. On the desk, four seconds. On the shelf behind you, half a minute.
3. In the filing room down the corridor, nearly two minutes. Ordered from a warehouse, the day after tomorrow.
4. In a paper archive across the country, four months. Written to someone abroad and waiting for a reply, six years.
5. Where this comparison breaks: a person waiting six years does nothing else. A processor does not wait; it switches to other work and comes back.

■ 1.6.3 PLAIN - a worked example

1. Here is the ladder, with real figures and the same figures stretched so 1 nanosecond becomes 1 second.

Step	Real time	If 1 ns were 1 second
CPU cycle at 4 GHz	0.25 ns	a quarter second
L1 cache hit	about 1 ns	1 second
L2 cache hit	about 4 ns	4 seconds

Step	Real time	If 1 ns were 1 second
L3 cache hit	about 30 ns	half a minute
Main memory, RAM	80 to 100 ns	1.5 minutes
NVMe SSD read	50 to 100 us	14 to 28 hours
Hard disk seek	8 to 10 ms	3 to 4 months
India to US, return	about 200 ms	6.3 years

```

cache      -> seconds
memory     -> a couple of minutes
flash SSD  -> a day
hard disk  -> a season
the world  -> most of a degree course

```

2. So one extra round trip to another country can waste more time than millions of memory accesses.

■ 1.6.4 PLAIN – what is really happening inside

1. Two limits build this ladder. The first is distance: signals cannot outrun light.
2. In copper or fibre they travel slower still, about two thirds of light speed.
3. In one nanosecond light covers about 30 centimetres in vacuum and about 20 centimetres in glass fibre.
4. The second limit is mechanism. A cache is a small grid of transistors that answers immediately.
5. Main memory is a huge grid of tiny capacitors that must be selected, sensed, amplified and re-freshed, which costs tens of nanoseconds however much you spend.
6. A hard disk must move an arm and wait for a platter to spin, which is a mechanical event.
7. The honest version: every number in that table is an average hiding a wide spread. An SSD read under heavy write load can be ten times its idle figure.
8. Treat latency as a distribution with a long tail, never as a single constant.

■ 1.6.5 TECHNICAL – the engineer’s version

1. Quoting a latency ladder is a **convention**, not a specification. Peter Norvig published an early version in his 2001 essay “Teach Yourself Programming in Ten Years”.
2. Jeff Dean’s version, circulated in Google talks from about 2009, made the practice standard.
3. Cache latencies are specified in cycles, not nanoseconds, because they scale with clock: L1 data 4 to 5 cycles, L2 12 to 20 cycles, L3 30 to 60 cycles.
4. L3 varies most, since it depends on core count and on-die topology. A large server part with a mesh interconnect can exceed 40 nanoseconds; a small desktop part can be near 10.
5. NVMe SSDs are quoted as 4 KiB random read latency: consumer TLC drives land near 50 to 100 microseconds, while Intel Optane, discontinued in 2022, reached about 10.
6. Round-trip time across an ocean has a hard floor set by fibre. Mumbai to northern Virginia is roughly 12,900 km great-circle, so about 25,800 km return.
7. At 200,000 km/s that is about 129 milliseconds before a single router touches the packet.
8. Observation tools: `perf stat -e cache-misses`, `lmbench` with its `lat_mem_rd` test, `fio` with `--iodepth=1` for true storage latency, and `ping` or `mtr` for network round trip.

■ 1.6.6 WORDS – remember these

1. Latency — how long before you get an answer — the delay between issuing a request and its first result.
2. Cache — a small fast copy kept close — an SRAM store holding recently or likely used lines from a slower level.

3. Round-trip time — go and come back — the time for a packet to reach a host and its reply to return, in milliseconds.
4. Tail latency — the slow few — high percentile response times such as p99, which dominate what users feel.

1.7 Units done properly

■ 1.7.1 PLAIN – in simple words

1. A bit is one on-or-off. A byte is eight bits stuck together.
2. Capital B means bytes. Lowercase b means bits. That one letter changes the number by eight.
3. Sellers of storage use round decimal thousands, so one kilobyte is 1,000 bytes.
4. Operating systems traditionally use binary thousands, so one unit is 1,024 bytes.
5. Both are defensible and neither is lying, but they disagree by a margin that grows with size. That is the whole reason a 1 TB drive appears as 931 GB.
6. Hertz means times per second. A 3 GHz processor ticks three billion times each second, but more ticks does not automatically mean more work done.

■ 1.7.2 PLAIN – a picture in your head

1. Think of buying rice. The shop sells a bag holding 1,000 grams. Your kitchen jar holds 1,024 grams.
2. You buy a thousand bags. The shop says 1,000 bags. Your jars say you filled 976.6 jars.
3. Where this comparison breaks: rice pours freely, and the difference there is a pure counting choice.
4. With computers the 1,024 is not arbitrary. Memory addressing is binary, so memory really does come in powers of two.
5. Storage has no such constraint, which is exactly why the two industries drifted apart.

■ 1.7.3 PLAIN – a worked example

```
1 TB drive as sold:
  1,000,000,000,000 bytes
/ 1024 ->      976,562,500 KiB
/ 1024 ->      953,674 MiB
/ 1024 ->      931.32 GiB
Windows prints "931 GB". Nothing is missing.
```

```
100 Mbps line:
  100,000,000 bits per second
/ 8      = 12,500,000 bytes per second
          = 12.5 MB/s on paper
x 0.949   Ethernet + IP + TCP overhead
          = about 11.9 MB/s of real file
```

1. Those two sums explain most unit confusion. A 100 Mbps line downloading a 1.2 GB file takes about 100 seconds, not 12 and not 1,200.
2. Clock speed works the same way. A 3 GHz processor has a cycle time of 1 divided by 3,000,000,000, which is 0.333 nanoseconds.

■ 1.7.4 PLAIN – what is really happening inside

1. Memory chips are addressed with binary numbers, so 30 address lines give exactly 2 to the power 30 locations, which is 1,073,741,824.
2. Not every byte on a network carries your file. Each packet carries headers and the link adds framing, so useful throughput is always below line rate.

3. The honest version: a byte has not always been eight bits. Early machines used 6, 7 and 9-bit bytes, which is why standards documents say **octet** when they mean exactly eight. Eight became universal by convention through the 1970s and only later entered standards.

■ 1.7.5 TECHNICAL – the engineer’s version

1. Decimal prefixes are SI, maintained by the BIPM: k is 10^3 , M is 10^6 , G is 10^9 , T is 10^{12} .
2. Binary prefixes are IEC: Ki is 2^{10} , Mi is 2^{20} , Gi is 2^{30} , Ti is 2^{40} . They were published as IEC 60027-2 Amendment 2 in January 1999 and adopted as IEEE 1541-2002, which the IEEE Standards Association raised to a full-use standard on 19 March 2005.

Unit	Bytes	Difference
1 kB / 1 KiB	1,000 / 1,024	2.4 percent
1 MB / 1 MiB	1e6 / 1,048,576	4.9 percent
1 GB / 1 GiB	1e9 / 1.0737e9	7.4 percent
1 TB / 1 TiB	1e12 / 1.0995e12	10.0 percent

3. Practice differs by product, and this is an **implementation detail** with a date: macOS has reported decimal GB since Mac OS X 10.6 Snow Leopard in 2009, while Windows File Explorer still shows binary values labelled GB as of 2026. On Linux `ls -h` is binary and `ls --si` is decimal.
4. Ethernet efficiency at a full 1500-byte MTU with IPv4 and TCP: 1460 bytes of payload inside 1538 bytes on the wire, counting the 14-byte header, 4-byte FCS, 8-byte preamble and 12-byte interframe gap. That is 94.9 percent, before any retransmission.
5. Observation tools: `lsblk -b` shows exact byte counts; `df -B1` avoids rounding; `iperf3` measures true throughput; `dd` with `oflag=direct` measures raw device rate.

■ 1.7.6 WORDS – remember these

1. Byte — eight bits — a group of 8 bits, symbol B; standards say octet when precision matters.
2. Kibibyte — the binary thousand — $2^{10} = 1,024$ bytes, symbol KiB, per IEC 60027-2.
3. Kilobyte — the decimal thousand — $10^3 = 1,000$ bytes, symbol kB, per SI.
4. Mbps — millions of bits per second — a line rate; divide by 8 for bytes, then subtract protocol overhead.
5. Hertz — times per second — cycles per second, symbol Hz; cycle time is the reciprocal.
6. IPC — how much work per tick — instructions retired per clock cycle, the other half of performance.

1.8 Hardware, software and firmware

■ 1.8.1 PLAIN – in simple words

1. Hardware is the part you could drop on your foot: metal, silicon, plastic, wires.
2. Software is the pattern of bits telling the hardware what to do. It weighs nothing.
3. Firmware is software that lives inside a piece of hardware and ships with it, so most people never see it.
4. The boundary sounds obvious and is not. It moves every few years, in both directions.
5. Jobs move into hardware when they are stable and popular, and out of hardware when they must change often.

■ 1.8.2 PLAIN – a picture in your head

1. Think of a road. The tarmac is hardware. The traffic rules are software.
2. Firmware is the painted lines and fixed signs: part of the road, really instructions, repainted occasionally.
3. Where this comparison breaks: firmware can be replaced remotely and instantly, and a bad replacement can make the road unusable.
4. That failure has a name, bricking: a failed firmware update leaving a device unable to start at all.

■ 1.8.3 PLAIN – a worked example

1. Modems were once a box of dedicated signal chips. From the late 1990s the winmodem moved that work onto the main processor, because it was cheaper.
2. Encryption moved the other way: AES was ordinary software until Intel added AES-NI instructions in 2010.

Job	Was	Became
Modem signal processing	hardware chip	host software
Video decode	software	fixed-function block
AES encryption	software	CPU instruction, 2010
Neural network math	software	NPU, from 2017

■ 1.8.4 PLAIN – what is really happening inside

1. Even the processor's own instruction list is not pure hardware on every design.
2. On x86, complicated instructions are broken into simpler internal steps by a small program stored on the chip. That program is microcode.
3. Microcode can be updated, which is how makers fix processor bugs without recalling chips.
4. Going the other way, a field-programmable gate array is a chip whose wiring you rewrite: you write something like a program and get real circuits.
5. The honest version: "hardware is fixed, software is changeable" is comfortable and wrong. The real question is how fast a thing can change and who is allowed to change it.

■ 1.8.5 TECHNICAL – the engineer's version

1. UEFI replaced the legacy PC BIOS, which dates from the IBM PC of 1981. The UEFI specification is maintained by the UEFI Forum and reached version 2.10 in 2022.
2. Microcode updates load at boot from firmware or from the operating system. On Linux they arrive as intel-ucode or amd-ucode packages and appear in dmesg.
3. FPGA designs are written in Verilog or VHDL and compiled to a bitstream. An ASIC is the same design frozen into silicon, cheaper per unit and impossible to change.
4. Observation tools: fwupdmgr get-devices lists updatable firmware on Linux; system_profiler SPHardwareDataType on macOS shows boot ROM version; nvme id-ctrl reports SSD firmware revision.

■ 1.8.6 WORDS – remember these

1. Firmware — software shipped inside a device — code in non-volatile memory on a peripheral or platform, updated rarely.
2. Microcode — the processor's inner script — on-die translation of architectural instructions into internal micro-operations.

3. FPGA — a chip you can rewire — a field-programmable gate array configured by a bitstream into arbitrary logic.
4. Bricking — an update that kills the device — a failed firmware write leaving hardware unable to boot or recover.

1.9 How to read this book

■ 1.9.1 PLAIN - in simple words

1. This book is written to be read twice, and the second reading is built into the structure.
2. Every section has six blocks: four marked PLAIN, one marked TECHNICAL, one word list.
3. On your first pass read only the PLAIN blocks, straight through, Chapter 1 to Chapter 53.
4. On your second pass go back for the TECHNICAL blocks. Now you have somewhere to put the numbers.
5. Do not read the technical half first. It is correct, it will not stick, and you will wrongly conclude you are bad at this.

■ 1.9.2 PLAIN - a picture in your head

1. Think of learning a city. First you walk the main roads and learn how the districts sit next to each other.
2. The PLAIN blocks are the main roads. The TECHNICAL blocks are the house numbers.
3. Where this comparison breaks: a city stays still, and some technical detail here has a shelf life.
4. When you see a year attached to a fact, that is deliberate. It tells you when to check whether it is still true.

■ 1.9.3 PLAIN - a worked example

1. Here is the whole book, one line per chapter.
2. PART A, THE PHYSICAL WORLD, chapters 1 to 6.
3. Chapter 1, The whole machine in one look.
4. Chapter 2, Electricity. Charge, voltage, current.
5. Chapter 3, From sand to silicon. Making a chip.
6. Chapter 4, The transistor. The switch under everything.
7. Chapter 5, Logic gates and arithmetic.
8. Chapter 6, How a machine remembers.
9. PART B, TURNING BITS INTO MEANING, chapters 7 to 9.
10. Chapter 7, Numbers and meaning. Binary, hex, text.
11. Chapter 8, Pixels and screens.
12. Chapter 9, Sound, magnets, speakers and calls.
13. PART C, THE MACHINE, chapters 10 to 14.
14. Chapter 10, Inside the CPU.
15. Chapter 11, The memory hierarchy.
16. Chapter 12, Storage. Disks, flash, wear.
17. Chapter 13, Buses, ports, drivers and firmware.
18. Chapter 14, A history of computing machines.
19. PART D, SOFTWARE, chapters 15 to 20.
20. Chapter 15, The birth of software.
21. Chapter 16, How a compiler works.
22. Chapter 17, Programming languages.
23. Chapter 18, Operating systems.

24. Chapter 19, The terminal and the shell.
25. Chapter 20, Files, extensions and the .exe.
26. PART E, SEEING AND SHOWING, chapters 21 to 22.
27. Chapter 21, From key press to pixel, in full.
28. Chapter 22, Graphics and the GPU.
29. PART F, NETWORKS, chapters 23 to 34.
30. Chapter 23, Networking from zero.
31. Chapter 24, IP, private vs public, NAT and CGNAT.
32. Chapter 25, DNS. Names into addresses.
33. Chapter 26, Packets and routing.
34. Chapter 27, Traceroute decoded, hop by hop.
35. Chapter 28, TCP and why a timeout is evidence.
36. Chapter 29, The diagnostic method.
37. Chapter 30, How the internet was built.
38. Chapter 31, Wireless. Wi-Fi, mobile data, radio.
39. Chapter 32, TLS and HTTPS.
40. Chapter 33, The OSI and TCP/IP models.
41. Chapter 34, Firewalls, VPNs, CDNs, IPv6 and localhost.
42. PART G, BUILDING AND SHIPPING, chapters 35 to 44.
43. Chapter 35, Web and app development.
44. Chapter 36, APIs. Machines calling machines.
45. Chapter 37, Git with no network.
46. Chapter 38, What a commit actually is.
47. Chapter 39, The index, push, merge and rebase.
48. Chapter 40, Your own git server, and what GitHub adds.
49. Chapter 41, CI/CD and what a run is.
50. Chapter 42, SSH, keys, tokens and scopes.
51. Chapter 43, Cloud, the Bezos mandate, AWS and Azure.
52. Chapter 44, Security. Threats and honest limits.
53. PART H, GAMES AND MACHINE INTELLIGENCE, chapters 45 to 51.
54. Chapter 45, Games. Loops, frames, input latency.
55. Chapter 46, Machine learning from scratch.
56. Chapter 47, Parameters, tokens and embeddings.
57. Chapter 48, The transformer.
58. Chapter 49, How a model is actually made.
59. Chapter 50, Benchmarks, evaluation, versions and dates.
60. Chapter 51, Serving, RAG, agents and limits.
61. PART I, REFERENCE, chapters 52 to 53.
62. Chapter 52, The engineer's reference.
63. Chapter 53, Master timeline and glossary.

■ 1.9.4 PLAIN – what is really happening inside

1. One more thing about Parts F and G, the networking part and the building-and-shipping part.
2. They do not use invented examples. They use one real fault and one real work session, from the reader this book was written for.
3. A real home broadband connection in India, on macOS, that reached some things and silently could not reach others.
4. A real traceroute crossing a home router, a provider's private core, then a large company's backbone through Delhi, Mumbai and Pune, and then answering nothing.
5. A real set of git problems in the same session: pushes that failed halfway, a token missing a permission, a rebase that changed a commit's identity.
6. In particular you will learn to separate what an observation proves from what it merely suggests. That distinction is the whole of diagnosis.

■ 1.9.5 TECHNICAL – the engineer's version

1. Where this book touches artificial intelligence, in Part H, it separates three kinds of statement, and you should demand the same separation elsewhere.
2. **Established fact:** the transformer architecture was published in the 2017 paper “Attention Is All You Need” by Vaswani and colleagues at Google. That is checkable.
3. **Active research:** how much of a model's ability comes from scale versus from data quality is still argued, with published results on both sides.
4. **Marketing claim:** any statement that a model reasons, understands, or is near human level is a claim about words with no agreed test behind it. A benchmark score is evidence about the benchmark first and about the world second.
5. Dates are attached to volatile facts deliberately. Anything about a current product, model or price should be re-checked; this chapter was written in 2026.

■ 1.9.6 WORDS – remember these

1. PLAIN block — the everyday half — beginner-level explanation with no jargon, safe to read in isolation.
2. TECHNICAL block — the engineer's half — precise terms, units, specifications and observation commands.
3. Standard — written down and agreed — behaviour defined in a published specification such as an RFC or an IEC document.
4. Convention — everyone just does it — widespread practice with no governing specification.
5. Implementation detail — true of this one product — behaviour of a specific version that may change without notice.

1.98 Common wrong ideas

1. Wrong: a computer understands what it is doing. Right: it combines bits by fixed rules, and all meaning is assigned by people outside the machine.
2. Wrong: memory and storage are two words for one thing. Right: memory is fast and forgets without power; storage is slow and remembers.
3. Wrong: more gigahertz always means a faster computer. Right: work done is instructions per cycle times frequency, and a stalled processor retires almost nothing at any clock speed.
4. Wrong: a 1 TB drive showing 931 GB is missing space. Right: a trillion divided by 1024 three times is 931.32, and the operating system is labelling binary units with a decimal name.

5. Wrong: a 100 Mbps line downloads at 100 MB/s. Right: divide by 8 for bytes and subtract protocol overhead, giving about 11.9 MB/s.
6. Wrong: hardware is fixed and software is changeable. Right: firmware and microcode are updatable and reconfigurable chips exist, so the real question is how fast a thing can change and who may change it.
7. Wrong: the delay you feel when typing is the computer thinking. Right: most of it is waiting for a polling interval and a screen refresh, not computation.
8. Wrong: a slow reply from a distant server means bad engineering. Right: a round trip from India to the United States has a physical floor near 130 milliseconds, because light in fibre is not instantaneous.

1.99 Chapter summary in 20 lines

1. A computer stores, moves and combines patterns of on and off, and does nothing else.
2. A bit is one on-or-off, and meaning is assigned by people, never by the metal.
3. Every computer has five parts: input, memory, storage, processing, output.
4. ENIAC in 1945, a desktop, a phone and a watch differ in size, not in those roles.
5. Memory is the countertop and forgets without power. Storage is the cupboard and remembers.
6. The one big idea is that instructions are data, held in the same memory as the numbers.
7. It was written down in the First Draft of a Report on the EDVAC, 30 June 1945.
8. That made machines general-purpose and made malicious code possible, at the same stroke.
9. The Manchester Baby ran the first stored program on 21 June 1948, and EDSAC followed on 6 May 1949.
10. The machine is built in layers, from silicon up to artificial intelligence services.
11. Each layer hides the mess below it, and that hiding is called abstraction.
12. No abstraction is watertight. Timing, limits and failures leak upward.
13. One key press becoming one letter takes about thirty handovers across nine layers.
14. Almost all of that delay is waiting for a poll and a screen refresh, not computing.
15. Speeds form a ladder: 1 ns for L1, 100 ns for RAM, 100 us for an SSD, 10 ms for a disk seek, 200 ms across the world.
16. Stretched so one nanosecond is one second, that ladder reads: a second, a minute and a half, a day, a season, six years.
17. A byte is eight bits. Capital B means bytes and lowercase b means bits.
18. Decimal kB is 1,000 and binary KiB is 1,024, so a 1 TB drive reports 931 GB.
19. Hardware, software and firmware differ by how easily a thing changes, not by material.
20. Read every PLAIN block first, end to end, then return for the TECHNICAL blocks.

Electricity — The Thing That Does All The Work

2.0 What this chapter gives you

1. You will be able to say what electricity is, in terms of atoms and the particles inside them.
2. You will be able to explain voltage, current and resistance, and do sums with them.
3. You will be able to work out how much heat a chip makes, and say where that heat comes from.
4. You will be able to trace energy from a wall socket to one transistor inside a processor.
5. You will be able to say what a resistor, capacitor, inductor and diode do, and read real part values.
6. You will be able to explain why computers went digital, using noise margin, not a slogan.
7. You will be able to say what “3.5 GHz” physically means, and why light speed limits chip size.
8. You will be able to explain how one bit is held as charge, and why it can be lost.

2.1 What electricity really is

■ 2.1.1 PLAIN – in simple words

1. Everything is made of atoms. An atom has a heavy centre, the nucleus, with light electrons around it.
2. The nucleus holds protons and neutrons. Each particle has a property called **charge**, a label for how it pushes and pulls on others.
3. A proton is positive, an electron is negative, and a neutron has none, so it is called neutral.
4. Same charges push apart, opposite charges pull together. A normal atom has as many electrons as protons, so it seems to have no charge.
5. In a metal such as copper, the outermost electron of each atom is held weakly and can wander between atoms.
6. Those wandering electrons are electricity. A battery pushes them one way, pulling electrons in at one terminal and shoving them out at the other.

■ 2.1.2 PLAIN – a picture in your head

1. Picture a pipe, already completely full of water, joined end to end in a ring, with a pump in it.
2. Because the pipe is already full, water moves at the far end almost the instant you start the pump.
3. The pump is the battery, its pressure is the voltage, the moving water is the current, and a narrow section is a resistor.
4. Put a water wheel in the ring and the flow turns it. That is a load. Not one drop is destroyed.

Where this comparison breaks

1. In a real wire the electrons crawl at about a tenth of a millimetre per second, far slower than water.
2. Water carries energy inside the pipe. In a circuit most energy travels in the field around the wire, not in the copper.
3. The honest version: the pipe picture models pressure and flow only. Use it for sums, never to decide what is physically true.

■ 2.1.3 PLAIN – a worked example

1. Take a copper wire one square millimetre in cross section carrying one ampere. Copper has about 8.5×10^{28} free electrons per cubic metre.

Drift velocity
 $v = I / (n * A * e)$
 $= 1 / (8.5e28 * 1e-6 * 1.602e-19)$
 $= 0.0000735 \text{ m/s} = 0.0735 \text{ mm per second}$
 $= \text{about } 26 \text{ centimetres per hour}$

2. So one electron takes over an hour to crawl the length of a desk lamp cable.
3. Yet the lamp lights instantly, because the push travels through the already-full wire near the speed of light.

■ 2.1.4 PLAIN – what is really happening inside

1. A battery's chemistry separates charge, leaving one terminal short of electrons and one with a surplus.
2. That separation creates an electric field, an invisible push filling the space around the terminals.
3. Connect a wire and the field spreads along it near light speed, so every free electron feels the push almost at once.
4. As the electrons drift they bump into vibrating copper atoms, handing over energy that appears as heat.
5. The electrons are not consumed; the same number leaves as enters. What is consumed is energy, delivered by the field to wherever the bumping happens.

■ 2.1.5 TECHNICAL – the engineer's version

1. Charge is quantized in units of the elementary charge e , fixed exactly at $1.602176634 \times 10^{-19} \text{ C}$ by the 2019 SI redefinition.
2. The ampere is now defined from that fixed e , rather than from a force between wires as it was from 1948 to 2019.
3. In a metal the valence and conduction bands overlap, so carriers just above the Fermi level respond to any applied field.
4. Copper gives about one conduction electron per atom, so n is roughly 8.5×10^{28} per cubic metre, and drift velocity is $v = J / (n e)$.
5. Signal propagation is a field phenomenon, travelling at c divided by the square root of the effective relative permittivity.
6. Poynting's theorem, published by John Henry Poynting in 1884, places the energy flux outside the conductor, not inside it.
7. History: von Kleist and van Musschenbroek made the Leyden jar in 1745 and 1746, J. J. Thomson identified the electron in 1897, and Ernest Rutherford proposed the nuclear atom in 1911.

■ 2.1.6 WORDS – remember these

1. Atom — the smallest normal piece of a material — a nucleus of protons and neutrons with bound electrons.
2. Charge — the property that makes things push or pull electrically — quantized in units of e , measured in coulombs.
3. Electron — a tiny negative particle that moves between atoms in a metal — the mobile charge carrier in a conductor.
4. Drift velocity — the slow crawl of the electron crowd — $v = J / (n e)$, typically under a millimetre per second.
5. Electric field — the invisible push that moves charges — force per unit charge, in volts per metre.

2.2 Voltage, current and resistance

■ 2.2.1 PLAIN – in simple words

1. **Voltage** is how hard the electricity is pushed, measured in volts. It is always a difference between two points.
2. **Current** is how much charge flows past a point each second, measured in amperes.
3. **Resistance** is how strongly a material fights the flow, measured in ohms.
4. One rule ties all three together: voltage equals current times resistance, written $V = I R$. Know any two and you can find the third.
5. **Power** is how fast energy is delivered. It is voltage times current, $P = V I$, measured in watts.
6. Nearly all the power that goes into a chip comes out as heat, which is why chips need fans and metal blocks.

■ 2.2.2 PLAIN – a picture in your head

1. Back to the water ring. Pressure is voltage, litres per second is current, a narrow section is resistance.
2. More pressure through the same pipe gives more flow, so more voltage across the same resistance gives more current.
3. Narrow the pipe further and less flows. More resistance means less current.
4. The pump's effort depends on both. High pressure with no flow does no work, and neither does flow with no pressure.

Where this comparison breaks

1. Water resistance changes messily with speed. A good resistor keeps almost the same ohms at any current.
2. The honest version: Ohm's law is not a law of nature like gravity. It describes materials that happen to behave in a straight line. Diodes, lamps and transistors do not.

■ 2.2.3 PLAIN – a worked example

Sum 1. A 5 V supply across a 220 ohm resistor.

$$I = V / R = 5 / 220 = 0.0227 \text{ A} = 22.7 \text{ mA}$$

Sum 2. A wire of 0.05 ohms carrying 2 A.

$$V = I * R = 2 * 0.05 = 0.1 \text{ V lost along it}$$

Sum 3. A part dropping 1.8 V at 15 mA.

$$R = V / I = 1.8 / 0.015 = 120 \text{ ohms}$$

Power. A core at 1.0 V burning 125 W.
 $I = P / V = 125 / 1.0 = 125 \text{ A}$

1. That last figure is real. A modern processor pulls over a hundred amps at about one volt.
2. One volt is a gentle push, but a hundred amps is the current of a small welding machine.

■ 2.2.4 PLAIN – what is really happening inside

1. Voltage is energy per unit charge. One volt means one joule given to every coulomb that passes.
2. Current is a counting rate. One ampere is about 6.24 million million million electrons per second.
3. Resistance comes from electrons colliding with warm, vibrating atoms, and each collision turns motion into more vibration, which is heat.
4. So heat is not a side effect bolted on. It is energy the electrons lose when they cannot travel freely.
5. In a chip most heat is not wire resistance. It is the charging and discharging of tiny capacitances inside every transistor.
6. Heat is the enemy because hot silicon leaks more current, which makes more heat, which makes more leaking. That loop can run away.

■ 2.2.5 TECHNICAL – the engineer’s version

1. Ohm’s law comes from Georg Simon Ohm, published in 1827 in *Die galvanische Kette, mathematisch bearbeitet*.
2. Power has three equivalent forms: $P = VI$, $P = I^2 R$, and $P = V^2 / R$.
3. Conductor loss is $I^2 R$, so doubling current quadruples loss. That is why power is transmitted at high voltage and low current.
4. Dynamic switching power in CMOS is about $P = \alpha C V^2 f$, with α the activity factor and C the switched capacitance.
5. The V^2 term is why lowering supply voltage is the strongest power lever, and why core voltages fell from 5 V to under 1 V.
6. Static power is leakage, from subthreshold conduction and gate-oxide tunnelling, and it rises sharply with temperature.
7. Thermal Design Power is a cooling specification, not a measured maximum. Real peaks routinely exceed it.

Processor	Base power	Max turbo power
Intel Core Ultra 9 285K	125 W	250 W
AMD Ryzen 9 9950X	170 W TDP	230 W PPT
Thin laptop chip	15 to 28 W	45 to 65 W
Phone application chip	2 to 5 W	8 to 12 W

8. These are published figures as of 2025. Tom’s Hardware measured a 285K peak near 325 W under Prime95 with AVX, above its 250 W rating.
9. Observe power live with sensors and turbostat on Linux, powermetrics on macOS, or the RAPL counters under `/sys/class/powercap`.

■ 2.2.6 WORDS – remember these

1. Volt — how hard electricity is pushed — joules per coulomb of potential difference.
2. Ampere — how much charge flows per second — one coulomb per second, defined from fixed e since 2019.

3. Ohm — how much a part fights the flow — one volt of drop per ampere of current.
4. Watt — how fast energy is used — one joule per second, equal to V times I .
5. TDP — the heat a cooler must remove — Thermal Design Power, a cooling target, not a measured peak.
6. Leakage — current that escapes when nothing is switching — subthreshold and tunnelling current, strongly temperature dependent.

2.3 A circuit: the loop that makes it work

■ 2.3.1 PLAIN – in simple words

1. A circuit needs four things: a source, a path out, something that does work, and a path back.
2. The source is a battery, adapter or generator, the paths are conductors, and the thing that does work is the load.
3. It must be a complete loop, because charge cannot pile up at the end of a wire and stop.
4. A **short circuit** is a return path that skips the load. With nothing to limit it, current becomes enormous and things melt.
5. **Ground** is the point we agree to call zero volts, and every other voltage is measured against it.
6. In **DC** the push is steady and one way. In **AC** it swaps direction many times a second.

■ 2.3.2 PLAIN – a picture in your head

1. Think of a one-way ring road with a roundabout where a machine pushes cars along.
2. Cars leave, drive through a town delivering goods, and come back to the roundabout.
3. Close the road beyond the town and no car can complete the trip, so none leave at all. That is an open circuit.
4. Build a slip road that skips the town and every car takes it, delivering nothing. That is a short circuit.

Where this comparison breaks

1. Cars choose routes. Charge does not choose; it splits between all available paths in inverse proportion to their resistance.
2. The honest version: ground is not one thing. Circuit common, chassis ground and protective earth are three different ideas sharing one word and one symbol.

■ 2.3.3 PLAIN – a worked example

1. Indian mains supplies about 230 V AC at 50 Hz, so the push reverses direction 100 times a second. Here is the journey to one transistor gate.

```

Wall 230 V AC, 50 Hz
|
v  bridge of 4 diodes: all half-cycles made positive
Pulsing DC, peaks near 325 V
|
v  large capacitor: fills the dips between peaks
Rough DC bus, about 324 V
|
v  transistor chops it at 60 kHz to 150 kHz
High-frequency square wave
|
v  small transformer: steps down and isolates
Low-voltage AC

```

```

|
v  fast diodes, capacitors, feedback control
Clean 20 V DC out of the brick
|
v  buck converters on the motherboard
12 V rail -> 5 V, 3.3 V, 1.8 V, about 1.0 V core
|
v
One transistor gate inside the CPU

```

2. A transformer shrinks as frequency rises. At 50 Hz it is a heavy iron block; at 100 kHz it fits on a fingertip.
3. That is why modern chargers are light. Old adapters worked directly at 50 Hz and were heavy.

■ 2.3.4 PLAIN – what is really happening inside

1. Four diodes act as one-way gates, so both halves of the AC wave come out the same way up.
2. The big capacitor stores charge on the peaks and gives it back in the gaps, smoothing the bumps.
3. A transistor switches that high DC on and off fast, making a square wave the transformer can use.
4. The transformer has two coils. Changing current in the first makes a changing field, which makes a voltage in the second.
5. The turns ratio sets the voltage ratio, and the coils never touch electrically. That isolation keeps mains voltage away from your hands.
6. A feedback circuit adjusts how long the switch stays on, holding the output steady as the load changes.

■ 2.3.5 TECHNICAL – the engineer’s version

1. Kirchhoff’s current law, from Gustav Kirchhoff in 1845, says currents into any node sum to zero. That is why the loop is mandatory.
2. Indian mains is nominally 230 V RMS at 50 Hz per IS 12360 and IEC 60038; North America uses 120 V RMS at 60 Hz.
3. RMS is the equivalent heating value. Peak is RMS times root two, so 230 V RMS peaks near 325 V, and a bridge leaves a bus near 324 V.
4. The ATX specification defines desktop rails. Legacy ATX 2.x supplies these, plus -12 V for legacy serial ports.

Rail	Nominal	Tolerance	Typical use
+12 V	12.0 V	plus or minus 5%	CPU VRM, GPU, fans
+5 V	5.0 V	plus or minus 5%	SATA, USB, logic
+3.3 V	3.3 V	plus or minus 5%	DIMM support, M.2
+5 VSB	5.0 V	plus or minus 5%	standby, wake on LAN

5. Intel’s ATX12VO, published around 2019 and revised to 2.0 in February 2022, keeps only 12 V and 12 V standby. The board then makes 5 V and 3.3 V locally.
6. ATX 3.0, February 2022, introduced the 12VHPWR connector rated to 600 W. ATX 3.1, 2024, replaced it with the 12V-2x6 variant after connector failures.

USB level	Voltage	Current	Power
USB 2.0 high power	5 V	500 mA	2.5 W

USB level	Voltage	Current	Power
USB 3.x	5 V	900 mA	4.5 W
USB Type-C 3 A	5 V	3 A	15 W
USB PD SPR max	20 V	5 A	100 W
USB PD 3.1 EPR max	48 V	5 A	240 W

- Power Delivery fixed voltages in the Standard Power Range are 5 V, 9 V, 15 V and 20 V.
- PD 3.1, announced in May 2021, added 28 V, 36 V and 48 V, giving 140 W, 180 W and 240 W.
- PD 3.0 also defines Programmable Power Supply, a continuously adjustable output from about 3.3 V to 21 V in 20 mV steps.

■ 2.3.6 WORDS - remember these

- Circuit — a complete loop for charge to travel round — a closed network obeying Kirchhoff's current and voltage laws.
- Load — the part that does the useful work — the element converting electrical energy into heat, light, motion or computation.
- Short circuit — an unwanted low-resistance path skipping the load — a fault path of near-zero resistance giving very high current.
- Ground — the point everyone agrees to call zero volts — the reference node, distinct from chassis and protective earth.
- AC and DC — a push that swaps direction, against a steady one-way push — alternating current at 50 Hz in India, against direct current.
- RMS — the equivalent steady value of a wobbling one — root mean square; peak is RMS times root two for a sine.
- Buck converter — a circuit that steps voltage down efficiently — a switching regulator using an inductor and a switch.

2.4 Conductors, insulators and semiconductors

■ 2.4.1 PLAIN - in simple words

- A **conductor** lets electricity pass easily: copper, silver, aluminium, gold. An **insulator** blocks it: plastic, rubber, glass, dry air.
- A **semiconductor** sits between them. Silicon is the famous one, and on its own it barely conducts.
- Copper conducts because each atom releases one outer electron, and those electrons roam free through the metal.
- Plastic does not conduct because every electron is locked into a bond between two atoms, with nothing free to move.
- Silicon locks its electrons too, but weakly, so a little heat or light can free a few of them.
- The magic is that we can change how well silicon conducts, on purpose, by mixing in traces of other elements. That is why computers are silicon.

■ 2.4.2 PLAIN - a picture in your head

- Imagine a car park. A conductor is one only half full, so any car can move at once.
- An insulator is packed solid, bumper to bumper, with a high wall and the next empty level far above. Nothing moves.
- A semiconductor is the same packed park, but the empty level is only a short ramp up, so a nudge of warmth or light lets a car reach it.

4. A neighbour shuffles into the space it left, so the space appears to move the other way. Engineers call that moving space a hole.

Where this comparison breaks

1. Real electrons are not objects in slots. They are quantum states shared across the whole crystal.
2. The honest version: band gap is the true idea. The levels are energy bands and the wall height is the gap in electron-volts. The picture gets the behaviour right and the physics only roughly right.

■ 2.4.3 PLAIN – a worked example

1. Resistivity says how strongly a material fights current for a standard sample size. The range spans over twenty powers of ten.

Material	Resistivity, ohm-metre	Band gap, eV
Silver	1.59×10^{-8}	none, metal
Copper	1.68×10^{-8}	none, metal
Aluminium	2.65×10^{-8}	none, metal
Pure silicon	about 2.3×10^3	1.12
Germanium	about 0.46	0.66
Glass	10^{11} to 10^{15}	wide
PTFE plastic	above 10^{23}	very wide

2. Copper conducts about 10^{11} times better than pure silicon, and silicon about 10^{20} times better than PTFE.
3. Now the trick. Add one phosphorus atom per ten million silicon atoms and the resistivity falls by a factor of thousands.
4. That is **doping**, and it is the step that turns sand into a switch.

■ 2.4.4 PLAIN – what is really happening inside

1. Silicon has four outer electrons and bonds with four neighbours, so every electron is paired and none is free.
2. Phosphorus has five outer electrons, so four bonds form and one electron is left loosely held. That is **n-type** silicon.
3. Boron has three, so one bond is incomplete, leaving a vacancy that behaves like a positive carrier. That is **p-type** silicon.
4. Put n-type and p-type next to each other and the junction passes current one way and blocks the other.
5. Stack those junctions in the right pattern and you get a transistor: a switch with no moving parts, controlled by a voltage, repeated billions of times in a chip.

■ 2.4.5 TECHNICAL – the engineer's version

1. Conduction follows band structure. Metals have a partly filled conduction band; insulators and semiconductors have a filled valence band and a forbidden gap.
2. Silicon has an indirect band gap of 1.12 eV at 300 K. Germanium is 0.66 eV, gallium arsenide 1.42 eV, gallium nitride 3.4 eV, diamond about 5.5 eV.
3. Silicon dioxide, the gate insulator, has a gap near 9 eV, which is why it works as a barrier only a few atomic layers thick.

4. Intrinsic carrier concentration in silicon at 300 K is about 1.0×10^{10} per cubic centimetre, against 5×10^{22} atoms per cubic centimetre.
5. Doping runs from about 10^{15} per cubic centimetre in lightly doped wells to above 10^{20} at source and drain contacts.
6. Copper resistivity has a temperature coefficient near 0.00393 per degree Celsius, so a trace is about 40% more resistive at 100 C than at 0 C.
7. Semiconductors move the opposite way: resistance falls as temperature rises, because thermal energy frees more carriers.
8. Aluminium was the standard on-chip interconnect until IBM shipped copper interconnect in 1997, cutting wire resistance by roughly 40%.

■ 2.4.6 WORDS – remember these

1. Conductor — a material electricity passes through easily — partly filled conduction band, high carrier mobility.
2. Insulator — a material electricity cannot pass through — a wide forbidden band gap, typically above 4 eV.
3. Semiconductor — a material in between, whose conducting can be controlled — a moderate band gap, roughly 0.5 to 3.5 eV.
4. Band gap — the energy step an electron must climb to move — the forbidden range between valence and conduction bands, in eV.
5. Doping — adding traces of another element to change conductivity — controlled introduction of donor or acceptor impurities.
6. n-type and p-type — silicon with spare electrons, against silicon with missing ones — donor-doped and acceptor-doped material.
7. Hole — the moving gap where an electron is missing — a quasiparticle carrying positive charge in the valence band.

2.5 The resistor

■ 2.5.1 PLAIN – in simple words

1. A resistor is a small part whose only job is to fight current by a fixed, known amount.
2. Inside is a ceramic rod coated with a thin film of carbon or metal, with a spiral groove cut into the film.
3. A longer, thinner spiral means a longer electrical path, so more ohms. That is how the value is set.
4. Resistors turn the energy they take into heat, so each has a power rating that must not be exceeded.
5. Two very common uses are the **pull-up** and the **pull-down**, which stop an input wire from having no defined voltage.
6. A wire connected to nothing is **floating**, and a floating input on a chip is a genuine bug, not a harmless idle state.
7. Another everyday use is limiting the current into an LED, which would otherwise destroy itself instantly.

■ 2.5.2 PLAIN – a picture in your head

1. A pull-up resistor is like a weak spring holding a door closed.
2. Anyone can push the door open easily, but if nobody touches it the spring decides where it sits.
3. Without the spring the door drifts, banging half open in any draught. That is a floating input.
4. The spring is deliberately weak so a real signal can override it without wasting effort.

Where this comparison breaks

1. A doorway caps how many people pass. A resistor does not cap current at all.
2. The honest version: a resistor develops a voltage proportional to whatever current passes through it. Force enough current through and it burns.

■ 2.5.3 PLAIN – a worked example

1. Light a red LED from a 5 V supply at about 15 mA. A red LED drops about 2.0 V, and that drop barely changes with current.

```
Voltage left for the resistor
= 5.0 - 2.0 = 3.0 V
```

```
R = V / I = 3.0 / 0.015 = 200 ohms
Nearest standard E12 value: 220 ohms
```

```
Actual current I = 3.0 / 220 = 13.6 mA
Heat in resistor
P = I * I * R = 0.01364 * 0.01364 * 220
= 0.041 W
```

```
A 0.25 W part is comfortable. Even an 0603 surface
mount part rated 0.1 W is fine.
```

2. Leave the resistor out and the LED tries to pass everything the supply can give, and fails within seconds.

■ 2.5.4 PLAIN – what is really happening inside

1. In the resistive film, electrons collide with the disordered atoms of the alloy, turning electrical energy into heat.
2. Because electrons lose energy along the way, the potential falls steadily from one end to the other. That fall is the voltage drop.
3. A pull-up connects an input pin to the supply, so a small current holds the pin high when nothing else drives it.
4. When a switch pulls the pin to ground it wins easily, because the pull-up is weak. A 10 kilohm pull-up on 3.3 V wastes only 0.33 mA.
5. Floating is dangerous because a MOS gate is an insulator, giving an input resistance above a million million ohms, so any stray charge drags the pin anywhere.
6. Worse, a pin resting halfway turns on both transistors of the input stage at once, drawing extra current and adding heat.

■ 2.5.5 TECHNICAL – the engineer's version

1. Types include carbon film, metal film, metal oxide, thick film on alumina for surface mount, wire-wound for power, and foil for precision.
2. Values follow the IEC 60063 preferred series. E12 gives twelve values per decade at 10% spacing, E24 gives 24, and E96 gives 96 for 1% parts.
3. The E12 sequence is 10, 12, 15, 18, 22, 27, 33, 39, 47, 56, 68, 82, and decade multiples of those.
4. Through-hole colour bands are read left to right, with the tolerance band set off by a wider gap.

Colour	Digit	Multiplier	Tolerance
Black	0	x1	none
Brown	1	x10	1%
Red	2	x100	2%
Orange	3	x1 k	none
Yellow	4	x10 k	none
Green	5	x100 k	0.5%
Blue	6	x1 M	0.25%
Violet	7	x10 M	0.1%
Grey	8	x100 M	0.05%
White	9	x1 G	none
Gold	none	x0.1	5%
Silver	none	x0.01	10%

5. Worked reading: red, red, brown, gold is 2, 2, times 10, at 5 percent, giving 220 ohms plus or minus 11 ohms.
6. Surface mount sizes and typical ratings: 0402 at 0.063 W, 0603 at 0.1 W, 0805 at 0.125 W, 1206 at 0.25 W, sized in hundredths of an inch.
7. Common pull-up values are 10 kilohm general purpose, and 4.7 or 2.2 kilohm for I2C depending on bus capacitance and speed.
8. The I2C specification, from Philips in 1982 and now maintained by NXP as UM10204, caps bus capacitance at 400 pF and constrains rise time per mode.
9. Microcontroller internal pull-ups are typically 20 to 50 kilohms and are too weak for I2C. A floating CMOS input shows a slow wandering trace on a scope, not a flat line.

■ 2.5.6 WORDS - remember these

1. Resistor — a part that fights current by a known amount — a two-terminal component with specified ohmic value and power rating.
2. Pull-up — a weak connection to the supply that holds a line high — a resistor from a node to VDD defining the idle logic level.
3. Pull-down — a weak connection to ground that holds a line low — a resistor from a node to VSS defining the idle logic level.
4. Floating — a wire connected to nothing, with undefined voltage — a high-impedance node with no DC path, open to noise coupling.
5. E12 series — the standard twelve resistor values per decade — an IEC 60063 preferred number series at 10% spacing.
6. Current limiting resistor — the resistor that stops an LED burning out — a series element sized as supply minus V_f , divided by target I_f .

2.6 The capacitor

■ 2.6.1 PLAIN - in simple words

1. A **capacitor** is two sheets of metal held very close, with a thin insulator between them so they never touch.
2. Apply a voltage and electrons pile onto one sheet while the same number are pulled off the other. No charge crosses the gap.
3. The sheets simply become oppositely charged, and the capacitor now holds energy.

- How much charge it holds per volt is its **capacitance**, in farads. A farad is huge, so real parts are millionths or million-millionths.
- Charging is not instant. Fed through a resistor, it fills quickly at first and then more slowly, and the time depends on resistance times capacitance. That product is the **time constant**.
- One bit of your computer's main memory is literally a capacitor holding charge, and it leaks, so it must be topped up.

■ 2.6.2 PLAIN – a picture in your head

- Think of a rubber sheet stretched tightly across a pipe, sealing it completely.
- Water cannot get past, but pushing against one side stretches the sheet, which pushes water out of the other side.
- Push harder and it stretches more. That stretch is stored energy, exactly like charge on the plates.
- Push and release repeatedly and water sloshes back and forth. That is why a capacitor passes AC and blocks DC.

Where this comparison breaks

- A rubber sheet has a breaking point. A capacitor has a voltage limit too, but past it the insulator punches through and the part usually shorts.
- The honest version: a capacitor does not pass AC. No charge ever crosses the insulator. Current flows in one wire and out the other because the charge on each plate is changing.

■ 2.6.3 PLAIN – a worked example

- Take a 10 kilohm resistor feeding a 100 nanofarad capacitor from a 3.3 V supply.

$$\begin{aligned}\tau &= R * C = 10000 * 0.0000001 \\ &= 0.001 \text{ s} = 1 \text{ millisecond}\end{aligned}$$

- The voltage climbs along a curve, reaching about 63.2% after one time constant, and never quite arriving.

Time	Multiples of tau	Percent charged	Volts
1 ms	1	63.2%	2.09
2 ms	2	86.5%	2.85
3 ms	3	95.0%	3.14
5 ms	5	99.3%	3.28

- Engineers treat five time constants as fully charged, because the last 0.7% is below normal measurement noise.
- This exact sum sets how long a reset pin stays low at power-on, how long a button debounce lasts, and how fast an edge can rise.

■ 2.6.4 PLAIN – what is really happening inside

- When a chip switches millions of transistors at once, it demands a burst of current within a few billionths of a second.
- The supply is centimetres away, and the copper path to it behaves like a small coil that cannot change current that fast.
- Without help the voltage at the chip would dip every time it worked hard, and it would misread its own signals.

4. A **decoupling capacitor** sitting millimetres from the power pins supplies that burst locally, then refills from the main supply.
5. Large ones hold plenty but respond slowly; small ones hold little but respond fast, so designers fit several values in parallel.
6. In main memory one bit is one capacitor plus one transistor, and charge leaks away within tens of milliseconds.
7. So the controller reads every row and writes it straight back, forever. That is **refresh**, and it never stops while power is on.

■ 2.6.5 TECHNICAL - the engineer's version

1. Capacitance is $C = Q / V$ in farads. For parallel plates, C equals epsilon-zero times epsilon-r times area, divided by separation.
2. Charging through a resistor gives $v(t) = V$ times one minus e to the power of minus t over RC , and discharging drops to 36.8% after one tau.
3. Energy stored is one half $C V$ squared, so a 470 microfarad bulk capacitor at 12 V holds 0.034 joules.
4. Real capacitors have equivalent series resistance and inductance, creating a self-resonant frequency above which they behave as inductors.
5. A typical 0402 100 nF ceramic self-resonates around 30 to 50 MHz, which is why power networks use several decades of value in parallel.
6. Dielectric class matters. C0G or NP0 is stable and low loss; X7R is denser but loses 30% to 60% of its capacitance under DC bias near its rating.
7. Robert Dennard invented the one-transistor, one-capacitor DRAM cell at IBM in 1966; United States patent 3,387,286 was granted in 1968.
8. Modern DRAM storage capacitance is roughly 6 to 10 femtofarads, held in a deep trench or tall pillar to gain area without floor space.
9. DDR4 requires every row refreshed within 64 ms, with an average interval t_{REFI} of 7.8 microseconds at or below 85 C, halved above it.
10. DDR5 tightens the window to 32 ms at normal temperature and 16 ms in the extended range, with on-die thermal sensing to request faster refresh.
11. Refresh costs real bandwidth; on large DDR4 modules the overhead can exceed 5% of available cycles. Read module timings on Linux with `decode-dimms`.

■ 2.6.6 WORDS - remember these

1. Capacitor — two plates with a gap, storing charge — a component storing energy in an electric field, $C = Q / V$.
2. Farad — the unit of charge stored per volt — one coulomb per volt; practical parts are pF, nF and uF.
3. Time constant — how long a capacitor takes to fill — $\tau = R C$; 63.2% after one tau, 99.3% after five.
4. Decoupling capacitor — a local energy store beside a chip — a capacitor supplying transient current and lowering power network impedance.
5. Dielectric — the insulating layer between the plates — the material whose permittivity sets capacitance and whose breakdown sets voltage rating.
6. ESR — the hidden resistance inside a capacitor — equivalent series resistance, which dominates ripple loss and heating.
7. Refresh — reading and rewriting memory so it does not forget — periodic restoration of DRAM cell charge within the retention window.

2.7 The inductor and the coil

■ 2.7.1 PLAIN – in simple words

1. Whenever current flows through a wire, an invisible **magnetic field** wraps around it in circles.
2. One straight wire makes a weak field. Wind it into a coil and every turn adds its field to the others.
3. A coil carrying current therefore behaves like a magnet. That is an **inductor**.
4. The second half of the rule matters more: a magnetic field that is changing pushes electrons in any nearby wire.
5. Current makes a field, and a changing field makes current. Those two sentences explain motors, generators, transformers and speakers.
6. An inductor resists changes in current, passing a steady current freely but fighting any sudden rise or fall. That makes it the opposite of a capacitor.
7. Coils are all over a computer: the blocks near the processor socket, the coil in every speaker, the aerial in every wireless chip.

■ 2.7.2 PLAIN – a picture in your head

1. Think of a heavy flywheel on a shaft. To get it spinning you must push for a while.
2. It refuses to speed up instantly however hard you shove, and once spinning it fights hard if you try to stop it.
3. Current in an inductor behaves exactly like that speed. It builds slowly and resists being stopped.
4. Cut the current to a coil abruptly and it produces a large voltage spike, which is why any circuit switching a relay or motor includes a diode.

Where this comparison breaks

1. A flywheel stores energy in motion. An inductor stores it in the magnetic field around the coil, not in the copper.
2. The honest version: nothing is spinning. The inertia is the field's resistance to change, described by Faraday's law and Lenz's law, acting at the speed of light.

■ 2.7.3 PLAIN – a worked example

1. A motherboard must make about 1.0 V for the processor core from the 12 V rail, at over 100 amps.
2. It cannot use a resistor, because 11 V times 100 A would waste 1100 watts as heat. It uses a **buck converter** instead.

Vout is about $D \cdot V_{in}$, where D is the duty cycle

To get 1.0 V from 12 V:

$$D = 1.0 / 12 = 0.083 = 8.3\%$$

At 1 MHz switching, one period is 1000 ns, so the switch is on 83 ns and off 917 ns, a million times every second.

Energy in one choke, $L = 0.15 \mu\text{H}$ at $I = 30 \text{ A}$:

$$\begin{aligned} E &= 0.5 \cdot L \cdot I \cdot I \\ &= 0.5 \cdot 0.00000015 \cdot 900 = 67.5 \text{ microjoules} \end{aligned}$$

3. Because 100 A is too much for one switch and one coil, the design splits into phases, often 8, 12 or 16.
4. The phases take turns, evenly spaced in time, so their ripple largely cancels. Those evenly spaced blocks around the socket are exactly this.

■ 2.7.4 PLAIN – what is really happening inside

1. When the switch turns on, 12 V appears across the inductor and current ramps up, building the magnetic field.
2. Current cannot jump, so it rises in a straight ramp while the output capacitor collects the charge.
3. When the switch turns off, the field collapses and drives the current onward through a second transistor to ground.
4. So current into the load is continuous, even though the switch is only connected 8.3% of the time.
5. A transformer uses the same physics differently: two coils share one field, and the voltage ratio equals the turns ratio.
6. In a loudspeaker a coil sits in a magnet's gap and is glued to a cone.
7. Current in the coil makes a force, so the cone moves. The speaker chapter builds on this.

■ 2.7.5 TECHNICAL – the engineer's version

1. Hans Christian Orsted found in 1820 that a current deflects a compass needle, and Andre-Marie Ampere formalized the force between conductors that same year.
2. Michael Faraday demonstrated electromagnetic induction in 1831 at the Royal Institution, showing a changing field induces an electromotive force.
3. Heinrich Lenz stated in 1834 that the induced current opposes the change producing it, which is why Faraday's law carries a minus sign.
4. The defining relation is $v = L \, di/dt$, with L in henries, named after Joseph Henry, who found induction independently around 1831.
5. Energy stored is one half $L I^2$. Real inductors also have DC resistance, core loss, and a saturation current.

Application	Typical inductance	Typical current
CPU VRM choke	0.1 to 1 μH	25 to 60 A
Buck for 5 V or 3.3 V	1 to 10 μH	1 to 5 A
Speaker voice coil	0.1 to 1 mH	0.1 to 3 A
Wireless charging coil	6 to 24 μH	under 2 A

6. Multiphase controllers interleave phases 360 degrees divided by N apart, cancelling much of the input and output ripple current.
7. Switching frequencies run 300 kHz to 1 MHz for desktop CPU VRMs, and 1 to 3 MHz for point-of-load converters.
8. Coils appear in antennas as matching elements, in relays and motors as the actuator, and formerly in disk drives as inductive read heads.
9. Inductive heads gave way to giant magnetoresistive heads, shipped by IBM in 1997, from work by Albert Fert and Peter Grunberg that won the 2007 Nobel Prize.
10. Qi wireless charging, from the Wireless Power Consortium, operates at 87 to 205 kHz in the base-line power profile.

■ 2.7.6 WORDS – remember these

1. Magnetic field — the invisible pull wrapping around a current — the vector field B, in tesla, produced by moving charge.
2. Inductor — a coil that resists changes in current — a component with defined inductance, following $v = L \, di/dt$.
3. Henry — the unit of inductance — one volt induced per ampere per second of current change.

4. Induction — a changing field creating a voltage in a nearby wire — Faraday's law, opposed in direction per Lenz's law.
5. Transformer — two coils sharing a field to change voltage — a coupled pair with output set by turns ratio, giving galvanic isolation.
6. VRM — the circuit making the processor's low voltage — voltage regulator module, a multiphase synchronous buck converter.
7. Saturation — the point where a coil's core gives up — the current above which permeability collapses and inductance falls sharply.

2.8 The diode

■ 2.8.1 PLAIN – in simple words

1. A **diode** lets current pass one way and blocks it the other. It is made by joining p-type silicon to n-type silicon.
2. Push current the right way and it flows, but the diode keeps about 0.7 volts as a toll, called the **forward voltage**.
3. Push the other way and almost nothing flows, until the voltage is so high that the diode breaks and usually dies.
4. Four diodes in a diamond form a **rectifier bridge**, turning alternating current into current that always flows one way.
5. An **LED** is a diode made from materials that give off light when current passes.
6. A **photodiode** is the same idea backwards. Light falling on the junction makes a small current, so it works as a light sensor.

■ 2.8.2 PLAIN – a picture in your head

1. Picture a swing door hinged to open only one way, with a weak spring holding it shut.
2. From the correct side you can push through, but you must lean on it a little. That lean is the forward voltage.
3. From the wrong side the door will not budge, and shoving hard enough breaks the hinges, which is exactly reverse breakdown.
4. Once open, extra people flow through very easily, which is why a diode cannot limit current by itself.

Where this comparison breaks

1. A door is open or shut. A diode conducts smoothly and exponentially: a trickle at 0.5 V, plenty at 0.7 V, far too much at 0.8 V.
2. The honest version: 0.7 V is not a fixed property. It is where a small silicon diode reaches a few milliamps at room temperature, and it shifts by about minus 2 mV per degree Celsius as it warms.

■ 2.8.3 PLAIN – a worked example

1. Feed 230 V AC into a bridge of four silicon rectifiers with a smoothing capacitor.

Peak = RMS * square root of 2 = 230 * 1.414 = 325 V

Two diodes are in the path at any moment, each dropping about 0.7 V:

325 - 1.4 = about 324 V DC

2. This is why the inside of a mains adapter is dangerous even after unplugging. The bulk capacitor can hold over 300 V for many seconds.
3. Now LEDs. Forward voltage depends on colour, because colour is set by the energy step each electron falls through.

LED colour	Forward voltage at 20 mA
Infrared	1.2 to 1.6 V
Red	1.8 to 2.2 V
Yellow or amber	2.0 to 2.2 V
Green	2.8 to 3.5 V
Blue	3.0 to 3.6 V
White	2.8 to 3.6 V

4. Redder light means less energy per photon and a lower forward voltage. Blue and white need about 3 V, so one 1.5 V cell cannot light a white LED.

■ 2.8.4 PLAIN – what is really happening inside

1. Where p-type and n-type meet, spare electrons from the n side drift across and fill vacancies on the p side.
2. That leaves a thin zone with no free carriers, called the depletion region, with a built-in voltage across it.
3. Connect positive to the p side and you push against that built-in voltage. Overcome it and carriers flood across.
4. Connect positive to the n side and the empty zone widens instead, so nothing can cross and the diode blocks.
5. In an LED, an electron crossing and dropping into a vacancy loses energy that leaves as a photon, and the band gap sets its colour exactly.
6. In a photodiode the reverse happens. An arriving photon knocks an electron free and the built-in field sweeps it out as current.

■ 2.8.5 TECHNICAL – the engineer's version

1. The current-voltage relation is the Shockley diode equation, with thermal voltage about 25.9 mV at 300 K, giving an exponential curve.
2. Forward voltage by technology: silicon p-n 0.6 to 0.7 V, germanium 0.25 to 0.35 V, Schottky 0.2 to 0.45 V, silicon carbide about 0.8 V.
3. Schottky diodes use a metal-semiconductor junction and have almost no reverse recovery time, which is why switching supplies use them.
4. Zener diodes run deliberately in reverse breakdown, at standard values from 2.4 V to 200 V, as simple references and clamps.

Part	Type	Rating
1N4148	small signal	100 V, 200 mA
1N4007	rectifier	1000 V, 1 A
1N5819	Schottky	40 V, 1 A
BAT54	surface mount Schottky	30 V, 200 mA

5. Ratings to check on any datasheet: peak inverse voltage, average forward current, forward voltage at the operating current, and reverse recovery time.

6. Oleg Losev reported light from silicon carbide junctions in 1927, but the effect was not developed at the time.
7. Nick Holonyak Jr. made the first practical visible-spectrum LED, a red one, at General Electric in 1962.
8. Shuji Nakamura produced high-brightness blue LEDs at Nichia in 1993, making white LEDs possible. He shared the 2014 Nobel Prize in Physics with Isamu Akasaki and Hiroshi Amano.
9. Photodiodes usually run reverse biased in photoconductive mode, giving faster response and a current linear in optical power.
10. Every CMOS input pin has ESD protection diodes to VDD and VSS, which is why driving a signal into an unpowered chip can back-feed its supply rail.

■ 2.8.6 WORDS – remember these

1. Diode — a one-way valve for current — a p-n junction device conducting only under forward bias.
2. Forward voltage — the toll for letting current through — V_f , about 0.7 V for silicon, set by the Shockley equation and temperature.
3. Reverse breakdown — where a diode gives up and conducts backwards — avalanche or Zener breakdown beyond the peak inverse voltage.
4. Rectifier — a circuit turning AC into one-way current — a half-wave or full-wave bridge arrangement of diodes.
5. LED — a diode that gives out light — a direct-band-gap junction emitting photons on carrier recombination.
6. Photodiode — a diode turning light into current — a reverse-biased junction producing photocurrent proportional to optical power.
7. Schottky diode — a fast diode with a low toll — a metal-semiconductor junction with low V_f and negligible reverse recovery.

2.9 Signals: analogue and digital

■ 2.9.1 PLAIN – in simple words

1. A **signal** is any physical thing that changes over time in order to carry information: a voltage on a wire, sound in air, light in a fibre.
2. An **analogue** signal carries meaning in its exact value. Half a volt more is a genuinely different message.
3. A **digital** signal carries meaning only in which side of a line the value falls on, so 2.9 V and 3.2 V mean exactly the same thing.
4. Every wire picks up unwanted wobble from nearby wires, motors and radio. That wobble is **noise**.
5. Noise damages an analogue signal a little at every stage, and the damage adds up and can never be removed.
6. A digital signal survives, because each stage rebuilds a clean, full-strength copy, as long as noise never pushes a value across the line.
7. The gap between the worst a sender may produce and the least a receiver demands is the **noise margin**.

■ 2.9.2 PLAIN – a picture in your head

1. Imagine passing a message down a line of a hundred people in a noisy room.
2. In the analogue version each person whispers the exact loudness they heard, so small errors creep in at every step and the end is mush.

3. In the digital version everyone agrees on only two messages, a nod or a shake, and each person makes a fresh, full-strength one.
4. Person one hundred receives exactly what person one sent, even though every step was imperfect.

Where this comparison breaks

1. If the room gets loud enough that a nod is mistaken for a shake, the error becomes permanent and is copied faithfully forever.
2. The honest version: digital is not immune to noise. Below the threshold errors vanish completely; above it they become total. That cliff edge is why systems add error-detecting codes on top.

■ 2.9.3 PLAIN – a worked example

1. A 3.3 V chip drives another. The sender promises a one will be at least 2.4 V, and the receiver reads anything above 2.0 V as a one.

$$\begin{aligned}\text{High noise margin} &= V_{OH \text{ min}} - V_{IH \text{ min}} \\ &= 2.4 - 2.0 = 0.4 \text{ V}\end{aligned}$$

$$\begin{aligned}\text{Low noise margin} &= V_{IL \text{ max}} - V_{OL \text{ max}} \\ &= 0.8 - 0.4 = 0.4 \text{ V}\end{aligned}$$

2. So any noise smaller than 0.4 V cannot corrupt a bit. Drop the system to 1.2 V logic and the cushion falls to roughly 0.15 V.
3. The same 0.2 V of noise that was harmless at 3.3 V now flips bits. Nothing about the noise changed, only the cushion.
4. That is the central trade of the last thirty years: lower voltage saves a great deal of power and costs you margin.

■ 2.9.4 PLAIN – what is really happening inside

1. A digital output is a pair of transistors, one connecting the wire to the supply and one to ground, with only one on at a time.
2. When the output flips, the wire's own capacitance must be charged or drained, and that takes time. That time is the **rise time**.
3. Rise time is measured from 10% to 90% of the swing, because the very start and end of the curve are slow and vague.
4. During the rise the voltage passes through the forbidden middle zone where the receiver's answer is undefined, which is safe only because the clock says when to read.
5. A faster edge spends less time undefined, but carries higher frequency content that reflects, radiates and rings, so designers sometimes slow edges on purpose.

■ 2.9.5 TECHNICAL – the engineer's version

1. The four defining parameters on every logic datasheet are V_{OH} minimum, V_{OL} maximum, V_{IH} minimum and V_{IL} maximum.
2. Noise margin high is $V_{OH \text{ min}}$ minus $V_{IH \text{ min}}$. Noise margin low is $V_{IL \text{ max}}$ minus $V_{OL \text{ max}}$.

Family	Supply	$V_{IL \text{ max}}$	$V_{IH \text{ min}}$
TTL 74xx	5.0 V	0.8 V	2.0 V
CMOS 74HC	5.0 V	1.35 V	3.15 V
LVTTL, LVCMOS	3.3 V	0.8 V	2.0 V

Family	Supply	VIL max	VIH min
LVC MOS	2.5 V	0.7 V	1.7 V
LVC MOS	1.8 V	0.63 V	1.17 V

Family	VOL max	VOH min	Margins, low and high
TTL 74xx	0.4 V	2.4 V	0.4 V and 0.4 V
CMOS 74HC	0.1 V	4.4 V	1.25 V and 1.25 V
LV TTL 3.3 V	0.4 V	2.4 V	0.4 V and 0.4 V
LVC MOS 1.8 V	0.45 V	1.35 V	0.18 V and 0.18 V

- These are typical family values from standard datasheets and the JEDEC JESD8 series. Always confirm against the exact part and its test conditions.
- TTL, transistor-transistor logic, came from the Texas Instruments 7400 series introduced in 1964 and dominated for two decades.
- TTL thresholds are asymmetric because bipolar inputs sink current when low. CMOS inputs draw almost none, so their thresholds sit near half the supply.
- Supply voltages fell in steps as features shrank: 5 V through the 1980s, 3.3 V from the mid 1990s, then 2.5 V, 1.8 V, 1.5 V and 1.2 V.
- Processor core voltage today runs roughly 0.7 V to 1.35 V, adjusted dynamically thousands of times a second by the power management unit.
- DDR4 uses a 1.2 V supply, DDR5 uses 1.1 V, and LPDDR5 uses 1.05 V with 0.5 V on the data lines.
- Rise time relates to bandwidth by the approximation: bandwidth in gigahertz equals 0.35 divided by rise time in nanoseconds.
- Modern high-speed links drop single-ended thresholds for differential pairs. LVDS swings about 350 mV around a common mode near 1.2 V, and the receiver measures only the difference, rejecting common-mode noise.

■ 2.9.6 WORDS - remember these

- Signal — something that changes over time to carry information — a time-varying physical quantity conveying data.
- Analogue — meaning carried by the exact value — a continuous-amplitude representation where every level is distinct information.
- Digital — meaning carried by which side of a line — a discrete representation quantized to defined logic levels.
- Noise — unwanted wobble picked up along the way — random or coupled interference added to a signal.
- Noise margin — the cushion before noise breaks a bit — VOH min minus VIH min, and VIL max minus VOL max.
- Rise time — how long an edge takes to go up — the 10% to 90% transition time, which sets the bandwidth required.
- Differential pair — two wires carrying opposite copies of one signal — a balanced scheme rejecting common-mode noise.

2.10 Clocks: how a computer keeps time

■ 2.10.1 PLAIN – in simple words

1. A computer needs a steady heartbeat so every part agrees when a step begins and ends. That is the **clock**.
2. The clock is a square wave: a voltage going high, low, high, low, forever, at an exactly even rate.
3. The steadiness comes from a small piece of quartz crystal, usually sealed in a tiny metal can on the board.
4. Quartz has an unusual property. Squeeze it and it makes a small voltage; apply a voltage and it physically bends. That is the **piezoelectric effect**.
5. A quartz bar of a given size wants to vibrate at one particular rate, the way a tuning fork wants to sound one note.
6. Put it in an amplifier that feeds its own output back into it and it settles at exactly that rate and stays there.
7. The crystal is slow compared with a processor, so a circuit called a multiplier turns a low rate into a very high one.

■ 2.10.2 PLAIN – a picture in your head

1. Think of a pendulum in a grandfather clock. Its swing rate is set by its length, not by how hard you push it.
2. A tiny nudge each swing keeps it going, and the swings stay even for years. The nudge does not set the rate; the pendulum does.
3. A quartz crystal is the same idea, but the pendulum is a sliver of stone the size of a grain of rice, swinging thirty-two thousand times a second.

Where this comparison breaks

1. A pendulum drifts with gravity and temperature. A crystal drifts with its cut, its temperature and its age.
2. The honest version: a crystal is not perfect. A typical watch crystal is specified at plus or minus 20 parts per million, which is about 1.7 seconds a day, or roughly ten minutes a year.

■ 2.10.3 PLAIN – a worked example

1. Nearly every real-time clock chip and quartz wristwatch uses exactly 32,768 hertz, which is two multiplied by itself fifteen times.

```
32768 -> 16384 -> 8192 -> 4096 -> 2048 -> 1024
-> 512 -> 256 -> 128 -> 64 -> 32 -> 16
-> 8 -> 4 -> 2 -> 1 Hz
```

Fifteen halvings. A chain of 15 flip-flops does it with no arithmetic, no adder and almost no power.

2. It is also a good compromise. A lower frequency needs a physically larger crystal, and a higher one burns more power in the divider.

```
Base clock (BCLK) = 100 MHz
Multiplier        = 35
Core clock         = 100 * 35 = 3.5 GHz
```

```
One clock period at 3.5 GHz:
T = 1 / 3.5e9 = 0.2857 ns = 285.7 picoseconds
```

3. So 3.5 GHz physically means the core's heartbeat completes 3,500,000,000 full cycles a second, each lasting 286 trillionths of a second.

■ 2.10.4 PLAIN – what is really happening inside

1. In one clock period a signal can only travel so far, because nothing beats light and copper is slower than that.
2. Light in empty space covers about 30 centimetres in one nanosecond. On a circuit board a signal covers roughly half that.

Medium	Speed	In 1 ns	In one 3.5 GHz tick
Vacuum, light	30 cm/ns	30 cm	8.6 cm
Coax at 66%	20 cm/ns	20 cm	5.7 cm
FR-4 microstrip	17.5 cm/ns	17.5 cm	5.0 cm
FR-4 stripline	15.0 cm/ns	15.0 cm	4.3 cm

3. The honest version: the often-repeated figure of about 8.5 centimetres per tick at 3.5 GHz is the vacuum figure. In copper on a board it is roughly half that, and inside a chip it is worse again.
4. That is a hard physical ceiling. A processor die is one to three centimetres across, so a signal cannot cross a large die and settle within one cycle.
5. This is why big chips are divided into regions, why data takes several cycles to cross a die, and why making a chip bigger does not make it faster.
6. The clock itself must reach every corner at almost the same moment, so designers build a balanced tree of buffers with equal path lengths.
7. Any remaining difference in arrival time is clock skew, and it eats directly into the time available for real work.

■ 2.10.5 TECHNICAL – the engineer's version

1. Jacques and Pierre Curie discovered piezoelectricity in 1880, working with quartz, tourmaline and Rochelle salt.
2. Warren Marrison and J. W. Horton built the first quartz clock at Bell Telephone Laboratories in 1927.
3. Watch crystals use a tuning-fork cut in a small cylindrical can, commonly specified at 32.768 kHz plus or minus 20 ppm at 25 C.
4. Frequency drifts with temperature along a parabola for tuning-fork cuts, peaking near 25 C. TCXO and OCXO parts add compensation or an oven, reaching 0.01 ppm.
5. The oscillator is typically a Pierce circuit: one inverting amplifier, the crystal, two load capacitors and a feedback resistor.
6. A phase-locked loop multiplies the reference, using a phase detector, a loop filter, a voltage-controlled oscillator and a feedback divider.
7. A 100 MHz base clock has been the convention on Intel and AMD desktop platforms since around 2011, replacing earlier 133 MHz and 200 MHz references.
8. That is a convention, not a written standard, and board makers may shift it slightly.
9. Advertised frequency is several numbers, not one: a base clock, a single-core boost and an all-core boost, all limited by temperature and current.
10. Clock jitter, the cycle-to-cycle variation, matters as much as average frequency for serial links. PCI Express 5.0 at 32 GT/s has a unit interval of 31.25 ps.
11. On-die clock distribution uses H-trees or grids with skew budgets of tens of picoseconds, and can consume 20% to 30% of dynamic power.

```
lscpu | grep -i mhz
grep MHz /proc/cpuinfo
sudo dmidecode -t processor
cpupower frequency-info
```

■ 2.10.6 WORDS – remember these

1. Clock — the steady heartbeat every part follows — a periodic square wave defining the timing reference for synchronous logic.
2. Piezoelectric effect — squeeze makes voltage, voltage makes bend — reversible electromechanical coupling, found by the Curies in 1880.
3. Crystal — the quartz sliver that sets the rate — a mechanical resonator of very high Q used as a frequency reference.
4. 32.768 kHz — the standard watch frequency — two to the power of fifteen hertz, divisible to exactly 1 Hz by fifteen flip-flops.
5. PLL — the circuit that multiplies a slow clock into a fast one — phase-locked loop with phase detector, loop filter, VCO and divider.
6. Base clock — the low reference frequency on the board — BCLK, conventionally 100 MHz on modern desktop platforms.
7. Clock skew — the small difference in when the beat arrives — the spread in clock edge arrival times across a die, in picoseconds.

2.11 How one bit is physically stored and moved

■ 2.11.1 PLAIN – in simple words

1. A bit is not a thing you can hold. It is an agreement about what a physical state means.
2. On a wire in transit, a bit is a voltage: above an agreed level means one, below another agreed level means zero.
3. In main memory, a bit is an amount of electric charge sitting on a tiny capacitor.
4. In a solid-state drive, a bit is charge trapped inside an insulated island, where it stays for years without power.
5. On a spinning hard disk, a bit is the direction a microscopic patch of magnetic material points.
6. In every case the physical thing is continuous and messy, and the digital reading is made by comparing it against a threshold.
7. The cushion between what a sender guarantees and what a receiver requires is the **noise margin**, and it keeps the reading correct.

■ 2.11.2 PLAIN – a picture in your head

1. Picture a water tank with a line painted on the side, and a rule: above the line means yes, below means no.
2. Anyone filling it is told to fill well above the line, and anyone reading it answers yes for anything above the line.
3. The gap between fill-to-here and read-as-yes-from-here is the safety cushion, and ripples smaller than that cushion do not matter.
4. Shrink the tank to save water and the cushion shrinks with it, while the ripples stay the same size.

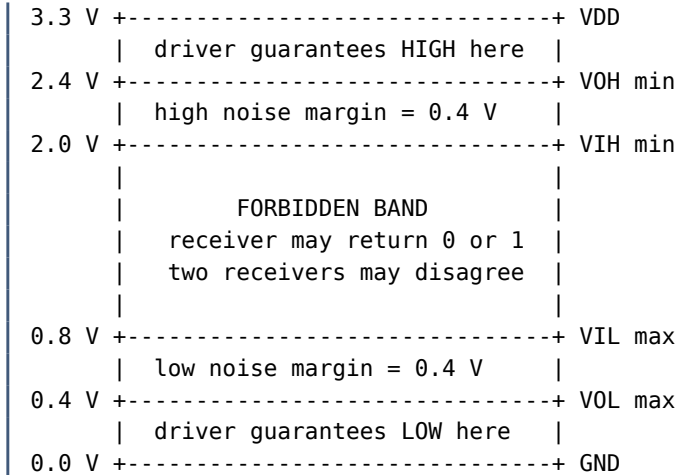
Where this comparison breaks

1. A tank has one line. Real logic has two, one for each direction, with a forbidden band between them.

2. The honest version: in that forbidden band a real receiver does not report an error. It returns a one or a zero effectively at random, and two receivers on the same wire may disagree.

■ 2.11.3 PLAIN – a worked example

1. Here is a 3.3 V logic signal drawn to scale in volts.



2. Now one bit of DRAM, using real numbers.

Cell capacitance	$C = 10 \text{ femtofarads} = 1\text{e-}14 \text{ F}$
Cell voltage	$V = 1.1 \text{ V}$
Charge stored	$Q = C * V = 1.1\text{e-}14 \text{ coulombs}$
Electrons	$N = Q / 1.602\text{e-}19$ = about 69,000 electrons

3. Sixty-nine thousand electrons is the entire physical existence of one bit of memory, and it leaks away within tens of milliseconds.

■ 2.11.4 PLAIN – what is really happening inside

1. To write a DRAM bit, the controller raises the word line, which switches on the cell's single transistor.
2. That connects the tiny capacitor to a long wire called the bit line, and the capacitor charges or discharges to match.
3. Lower the word line, the transistor turns off, and the charge is sealed in. The bit is stored.
4. To read it, the bit line is first set exactly halfway between one and zero, then the word line is raised.
5. The cell's small charge nudges the bit line up or down by only tens of millivolts, and a sense amplifier detects which way it moved.
6. Reading destroys the stored charge, so the sense amplifier immediately writes the value back. Every read is also a rewrite.
7. Nothing here is truly digital. It is careful analogue engineering, with the decision made at the end, on purpose, at a chosen moment.

■ 2.11.5 TECHNICAL – the engineer's version

1. Storage mechanisms and their physical carriers, with real figures.

Technology	Physical carrier	Typical retention
SRAM cell	6 cross-coupled MOSFETs	while powered
DRAM cell	6 to 10 fF capacitor	32 to 64 ms
NAND flash	trapped gate charge	about 1 to 10 years
Hard disk	magnetic domain	decades

2. DRAM sense amplifier input is on the order of 50 to 150 mV of bit line swing, against a precharge reference of half the supply.
3. NAND flash stores several bits per cell using more threshold levels: SLC 2 levels, MLC 4, TLC 8, QLC 16, and PLC 32 in research.
4. Each extra level halves the voltage window per state, which is exactly why QLC has lower endurance and needs stronger error correction.
5. Modern NAND relies on LDPC error-correcting codes in the controller, because the raw bit error rate is far too high to use directly.
6. Row hammer, published by Yoongu Kim and colleagues in 2014, showed that repeatedly activating one DRAM row flips bits in neighbouring rows.
7. That is a direct consequence of everything in this chapter: bits are charge, charge leaks, and neighbours are very close together.
8. Mitigations include Target Row Refresh, higher refresh rates and on-die ECC in DDR5. Research since 2020 has defeated several of them, so this is active research, not a solved problem.
9. Cosmic-ray neutrons and alpha particles also flip DRAM bits. Server memory uses ECC, usually single-error-correcting and double-error-detecting.
10. Inspect memory error events on Linux with `edac-util -v`, or with `rasdaemon` and `ras-mc-ctl --summary`.

■ 2.11.6 WORDS – remember these

1. Bit — the smallest piece of information, a yes or a no — a binary digit, physically encoded as a distinguishable state.
2. Threshold — the level that decides one from zero — VIH and VIL, the guaranteed input decision points.
3. Sense amplifier — the circuit that reads a very faint memory bit — a differential amplifier resolving tens of millivolts of bit line swing.
4. Word line — the wire selecting one row of memory cells — the gate control line activating access transistors across a row.
5. Bit line — the wire carrying the value in or out — the shared column line joining cells to sense amplifiers.
6. ECC — extra bits that catch and fix memory errors — error-correcting code, commonly SECDED for DRAM and LDPC for NAND.
7. Row hammer — poking memory until neighbours change — a disturbance fault from repeated row activation, first published in 2014.

2.98 Common wrong ideas

1. Wrong: Electrons travel through wires at the speed of light. Right: Individual electrons crawl, usually under a tenth of a millimetre per second. The field that pushes them travels near light speed, and that is what makes the lamp light instantly.
2. Wrong: The bulb uses up the current, so less comes back than went in. Right: Exactly the same current returns, because charge is conserved. What the bulb uses is energy, which the charge carries and gives up as heat and light.
3. Wrong: Higher voltage always means more power. Right: Power is voltage times current. A 10,000 V static spark carries microjoules and is harmless, while a 12 V car battery at 400 A can melt a spanner.
4. Wrong: Static electricity is a different kind of electricity from current in a wire. Right: It is the same electrons and the same charge. The only difference is that static charge sits still on an insulator instead of flowing through a conductor.
5. Wrong: An input pin with nothing connected reads zero. Right: It floats. Its voltage is undefined and wanders with nearby noise, so it can read one, zero, or both within a microsecond, and it makes the chip draw extra current.
6. Wrong: Ground is a real place where electricity is disposed of. Right: Ground is whichever node we choose to call zero volts. A phone on battery has a ground that touches nothing in the earth at all.
7. Wrong: A capacitor lets direct current pass through it. Right: No charge ever crosses the insulating gap. Current appears to flow because charge accumulates on one plate and leaves the other.
8. Wrong: Digital circuits are immune to noise. Right: They tolerate noise up to the noise margin and then fail completely. Below the margin errors vanish; above it bits flip, which is why error-correcting codes exist.
9. Wrong: Ohm's law applies to everything electrical. Right: It applies to ohmic materials over a limited range. Diodes are exponential, lamps change resistance as they heat, and transistors are controlled devices, not resistors.
10. Wrong: Chips get hot because of poor design, and better engineering could remove the heat. Right: Computing physically consists of charging and discharging capacitance. The energy has to go somewhere, and it goes into heat. It can be reduced, never abolished.

2.99 Chapter summary in 20 lines

1. Atoms have a nucleus of protons and neutrons surrounded by electrons, and charge is the property that makes them push and pull.
2. In a metal one outer electron per atom is loosely held, and that crowd of free electrons is what carries current.
3. Voltage is energy per unit charge, current is charge per second, and resistance is how strongly a material fights the flow.
4. Ohm's law, from Georg Ohm in 1827, ties them together as V equals I times R , and holds only for ohmic materials.
5. Power is V times I , also I squared R , and almost all power delivered to a chip leaves it again as heat.
6. Heat comes from electrons colliding with the lattice, and inside chips mainly from charging capacitance billions of times a second.
7. A circuit must be a closed loop because charge cannot pile up at the end of a wire, and ground is simply the node chosen as zero volts.
8. Indian mains is 230 V RMS at 50 Hz, peaking near 325 V, and an adapter rectifies, smooths, chops, transforms and regulates it down.
9. ATX supplies 12 V, 5 V, 3.3 V and 5 V standby, while USB Power Delivery 3.1 has reached 48 V and 240 W since 2021.
10. Conductors have free carriers, insulators have a wide band gap, and semiconductors sit between, with silicon at 1.12 eV.
11. Doping silicon with phosphorus or boron makes n-type or p-type material, and the junction between them is the basis of every chip.
12. A resistor sets a known voltage drop for a given current, provides pull-ups and pull-downs, and limits LED current.
13. A floating input is a genuine bug, because a MOS gate has near-infinite input resistance and follows any stray noise.
14. A capacitor stores charge on two plates, charges along an exponential curve with time constant RC , and decouples power locally.
15. One DRAM bit is about 69,000 electrons on a few femtofarads, which leaks and must be refreshed within 32 ms on DDR5.
16. An inductor stores energy in a magnetic field and resists changes in current, which is what makes buck converters and VRMs possible.
17. A diode conducts one way only, costing about 0.7 V in silicon; LEDs emit photons on recombination and photodiodes do the reverse.
18. Digital beat analogue because every stage regenerates a clean level, so errors stop accumulating while noise stays inside the margin.
19. A quartz crystal, using the piezoelectric effect found by the Curie brothers in 1880, gives the stable reference a PLL multiplies to gigahertz.
20. At 3.5 GHz one clock period is 286 picoseconds, in which a board signal travels four to five centimetres, a hard limit on chip size.

From Sand to Silicon — Rock, Furnace, Wafer, Fab

3.0 What this chapter gives you

1. You will be able to say where the silicon in a chip comes from, naming the rock and the furnace.
2. You will be able to explain why 99 percent pure is useless here.
3. You will be able to describe how a single crystal is grown, and why the whole ingot must be one unbroken lattice.
4. You will be able to walk from a 265 kilogram crystal to a 775 micrometer polished wafer, naming every cut.
5. You will be able to explain photolithography, and why shorter wavelength light gives smaller features.
6. You will be able to explain how extreme ultraviolet light is made from a falling tin droplet.
7. You will be able to explain doping, deposition, the copper wiring stack, testing, dicing and packaging.
8. You will be able to say honestly what “3 nm” means in 2026.
9. You will be able to explain why leading-edge chip making sits in a handful of buildings on Earth.

3.1 Where silicon comes from

■ 3.1.1 PLAIN – in simple words

1. Silicon is a chemical element, a basic building block of matter, like iron or oxygen.
2. About 28 percent of the mass of the Earth’s rocky shell is silicon. Only oxygen is more common.
3. But you will never dig up a lump of pure silicon anywhere on Earth.
4. Every silicon atom in the ground is already bound to oxygen, tightly.
5. Silicon plus oxygen makes silica. Silica plus metals makes the silicate minerals, which is most rock.
6. So the job is not to find silicon. The job is to tear the oxygen off it.
7. The purest common form of silica is the mineral quartz. That is what we mine.

■ 3.1.2 PLAIN – a picture in your head

1. Think of silicon as a person already holding hands with two oxygen partners, and refusing to let go.
2. Rock is that crowd, frozen solid. No silicon atom stands alone.
3. Quartz is the tidiest part of the crowd. Beach sand is a messier part, with shell and iron mixed through.

- Where this comparison breaks: atoms do not want anything. The real fact is that the silicon-oxygen bond is strong, near 800 kilojoules per mole.

■ 3.1.3 PLAIN – a worked example

- Take a kilogram of average continental crust and sort it by mass.

Element	Share of crust by mass
Oxygen	about 46 percent
Silicon	about 27.7 percent
Aluminum	about 8 percent
Iron	about 5 percent

- So that kilogram holds roughly 277 grams of silicon, and none of it is loose.
- Typical beach sand runs 90 to 99 percent silica. Chip-grade quartzite runs above 99.5 percent.
- That gap of a few percent is why we mine specific rock instead of a beach.

■ 3.1.4 PLAIN – what is really happening inside

- Silica is silicon dioxide: one silicon atom joined to two oxygen atoms, in an endless solid network.
- In quartz that network is a regular crystal with little else mixed in.
- In granite and clay the same units combine with sodium, potassium, aluminum, iron and magnesium. Those are silicates.
- Iron is the enemy. Iron atoms inside silicon later act as traps that ruin the electrical behaviour we want.
- So mining is really sorting. We look for rock where nature already sorted for us.
- The ore must also be lumpy, because the furnace needs gas to flow up through the charge. Powder would choke it.

■ 3.1.5 TECHNICAL – the engineer's version

- Silicon, atomic number 14, is the second most abundant crustal element at about 27.7 percent by mass, after oxygen.
- It does not occur natively, only as silicon dioxide and as the silicate mineral families.
- Feedstock is lump quartz or quartzite, sized roughly 10 to 100 millimetres, specified above 99 percent silicon dioxide.
- Limits are set on iron, aluminum, calcium, titanium, boron and phosphorus.
- Boron and phosphorus matter most. They are dopants, and they resist later chemical removal.
- Producing regions include Norway, Spain, Turkey, Brazil, China, Russia and the United States.

Material	What it is	Typical purity
Beach sand	Mixed sediment	90 to 99 pct silica
Silica sand	Washed and graded	98 to 99.5 pct
Quartzite lump ore	Metamorphic rock	above 99.5 pct
Spruce Pine quartz	Ultra-high purity	above 99.99 pct

- The honest version: Spruce Pine quartz, in North Carolina, is famous, but it is not usually the feedstock that becomes the chip.
- It is mainly melted into the fused-quartz crucibles used later in crystal growth. Do not repeat the claim that chips start as Spruce Pine sand.

■ 3.1.6 WORDS – remember these

1. Silica — silicon joined to oxygen — silicon dioxide, SiO₂, the network solid forming quartz.
2. Silicate — silica with metals mixed in — a mineral family built on silicon-oxygen tetrahedra.
3. Quartz — the clean crystal form of silica — crystalline SiO₂, the preferred carbothermic feedstock.
4. Quartzite — hard rock made mostly of quartz — metamorphosed sandstone supplied as graded lump ore.

3.2 First refining: the submerged-arc furnace

■ 3.2.1 PLAIN – in simple words

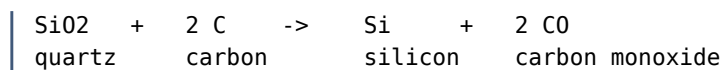
1. We have crushed quartz. Now we pull the oxygen off, using carbon and a very large amount of heat.
2. Carbon takes oxygen readily. Hot enough, it strips oxygen from silicon and leaves as carbon monoxide gas.
3. The machine is a submerged-arc furnace: a huge lined bowl with thick carbon rods pushed into the charge.
4. Electricity arcs through the material and heats it to around 1,900 degrees Celsius, hotter than lava.
5. Molten silicon collects at the bottom and is drained through a tap hole.
6. What comes out is metallurgical-grade silicon, about 98 to 99.5 percent pure.
7. That sounds excellent. For a chip it is hopelessly dirty.

■ 3.2.2 PLAIN – a picture in your head

1. Picture a stone pot several metres across, filled with gravel, coal and wood chips.
2. Three carbon rods, each as thick as a tree trunk, are pushed down into the mixture.
3. Where the current jumps it makes an arc, and that arc sits buried inside the charge. That is “submerged arc”.
4. Where this comparison breaks: nothing is cooking. This is a chemical reduction reactor, with reactions stacked in temperature zones.

■ 3.2.3 PLAIN – a worked example

1. Here is the overall reaction, written the simple way.



2. Silicon dioxide is near 60 grams per mole; silicon is near 28. So 60 kilograms of quartz gives at most 28 kilograms of silicon.
3. Real furnaces recover roughly 80 to 90 percent. Some silicon escapes as silicon monoxide gas.
4. Energy cost is roughly 11 to 13 kilowatt hours per kilogram of product.
5. A large furnace runs at 20 to 40 megawatts, continuously, for years. It is never switched off if that can be avoided.

■ 3.2.4 PLAIN – what is really happening inside

1. The charge is quartz plus carbon sources: coal, charcoal, petroleum coke and wood chips.
2. Wood chips are added for a physical reason. They keep the charge porous so gas can escape.
3. In the hot lower zone, around the arc, silicon dioxide is reduced to liquid silicon.
4. In the cooler upper zone, escaping silicon monoxide reacts with carbon to form silicon carbide, which sinks and reacts further.
5. That recycling is why the process works. Without it, far too much silicon would leave as gas.

6. The honest version: the tidy equation is a summary, not a description. It is a coupled set of reactions across a temperature gradient.
7. Impurities stay with the metal. Iron, aluminum and calcium dissolve in the liquid silicon.

■ 3.2.5 TECHNICAL – the engineer’s version

1. This is carbothermic reduction of silicon dioxide in a three-phase submerged-arc furnace, with Soderberg or prebaked graphite electrodes.
2. Arc temperature is around 1,900 to 2,000 degrees Celsius. Silicon melts at 1,414 degrees Celsius, so the product is liquid.
3. The dominant intermediate is silicon carbide.
4. Output is metallurgical-grade silicon, written MG-Si, at 98 to 99.5 percent.
5. Ladle refining with oxygen and slag additions removes some aluminum and calcium. It does not remove boron or phosphorus.

Grade	Silicon purity	Main use
Ferrosilicon	alloy, not pure	Steel making
MG-Si	98 to 99.5 pct	Alloys, silicones
UMG-Si	99.9 to 99.999 pct	Some solar cells
Polysilicon EG	99.9999999 pct up	Chips

6. World output is millions of tonnes per year. Only a small share reaches semiconductor purity.
7. Silica fume, the dust caught from the offgas, is sold as a concrete additive. It is a revenue stream, not waste.

■ 3.2.6 WORDS – remember these

1. Carbothermic reduction — using carbon and heat to strip oxygen — carbon as the reducing agent at high temperature.
2. Submerged-arc furnace — a furnace whose arc is buried in the material — a three-phase electrode furnace.
3. Metallurgical-grade silicon — first-pass silicon metal — MG-Si at 98 to 99.5 percent, the input to chemical purification.
4. Silica fume — fine dust caught from the exhaust — amorphous silicon dioxide sold as a concrete admixture.

3.3 Getting to nine nines

■ 3.3.1 PLAIN – in simple words

1. Silicon at 99 percent has one foreign atom in every hundred.
2. For a chip we need roughly one foreign atom in every billion, and for some impurities far fewer.
3. You cannot get there by melting and skimming. You need chemistry.
4. The trick is to turn solid silicon into a liquid chemical, clean it, then turn it back into solid silicon.
5. The chemical is trichlorosilane. It boils just under 32 degrees Celsius, so we can distil it.
6. Then we pipe the cleaned vapour over hot silicon rods, and silicon grows back onto them, atom by atom.
7. The result is polysilicon, at nine nines purity or better: 99.999999 percent.

■ 3.3.2 PLAIN – a picture in your head

1. Imagine a bucket of mixed salt and sand, and you want pure salt.
2. You dissolve what you can, filter out what will not dissolve, then boil the water away. The salt comes back clean.
3. Making trichlorosilane is the dissolving step. Distillation is the filtering step. Growing rods is the boiling-dry step.
4. Where this comparison breaks: salt water separates in one pass. Nine nines needs columns in series, some tens of metres tall, plus days of deposition.

■ 3.3.3 PLAIN – a worked example

1. Purity numbers are hard to feel, so convert them into countable things.

Purity	Foreign atoms	In everyday terms
99 pct (2N)	1 in 100	1 bad brick in a wall
99.9999 pct (6N)	1 in 1 million	1 person in a city
99.9999999 pct (9N)	1 in 1 billion	1 second in 32 years
11N	1 in 100 billion	1 card in a stack

2. That stack of 100 billion cards would stand about 15,000 kilometres tall, with exactly one wrong card in it.
3. Now the humbling part. A cubic centimetre of silicon holds about 5 times 10 to the power 22 atoms.
4. So even at nine nines, that cubic centimetre still holds around 50 thousand billion foreign atoms.
5. Nine nines is not “no impurities”. It is “impurities below the level where they change the behaviour we care about”.

■ 3.3.4 PLAIN – what is really happening inside

1. Step one. Ground metallurgical silicon meets hydrogen chloride gas at around 300 degrees Celsius in a fluidized bed.
2. Out comes trichlorosilane. Impurities come along as their own chlorides, with their own boiling points.
3. Step two. The mixture goes through distillation columns. Boron and phosphorus compounds are hardest to strip and set the final purity.
4. Step three. The cleaned liquid is vaporized, mixed with hydrogen, and fed into a sealed bell-jar reactor.
5. Inside stand thin silicon filaments heated to about 1,100 degrees Celsius. The gas breaks down there, silicon sticks, hydrogen chloride leaves.
6. Over two to four days the filaments grow into grey rods 15 to 20 centimetres across, which are broken into chunks.
7. This is polysilicon: extremely pure, but made of many small crystals pointing in random directions. We fix that next.

■ 3.3.5 TECHNICAL – the engineer’s version

1. The dominant route is the Siemens process, developed by Siemens in Germany in the late 1950s and still the workhorse in 2026.
2. Hydrochlorination runs at roughly 300 to 350 degrees Celsius. Trichlorosilane boils at 31.8 degrees Celsius, making distillation practical.

3. Deposition is chemical vapour deposition onto filaments at about 1,100 to 1,150 degrees Celsius inside a water-cooled bell jar.
4. Conversion per pass is low, near 20 percent, so unreacted gas and silicon tetrachloride are recycled.

Grade	Purity	Typical use
Solar, multicrystalline	7N to 8N	Standard PV cells
Solar, monocrystalline	9N to 10N	Higher-eff PV
Electronic grade	10N to 11N	Integrated circuits

5. The main alternative is the fluidized bed reactor route, fed with monosilane, giving granular polysilicon at roughly one tenth of the heating electricity.
6. The honest version: “eleven nines” is a bulk metallic-impurity figure, not a statement about every element.
7. Dopants such as boron and phosphorus are specified separately, in parts per billion atomic or as resistivity. Carbon and oxygen are separate again.
8. Metrology includes glow discharge mass spectrometry, inductively coupled plasma mass spectrometry, and photoluminescence.

■ 3.3.6 WORDS – remember these

1. Trichlorosilane — the liquid we clean instead of the metal — SiHCl_3 , boiling point 31.8 degrees Celsius.
2. Siemens process — growing pure silicon back onto hot rods — CVD from trichlorosilane and hydrogen at about 1,100 degrees Celsius.
3. Polysilicon — very pure silicon made of many small crystals — electronic-grade polycrystalline silicon, the feedstock for crystal growth.
4. Nine nines — 99.999999 percent pure — 9N, one foreign atom per billion by the stated metric.

3.4 Growing a single crystal

■ 3.4.1 PLAIN – in simple words

1. Our polysilicon chunks are pure, but their internal arrangement is a mess.
2. They are many small crystals at random angles. Where two meet there is a seam called a grain boundary.
3. Electrons behave badly at those seams. A chip needs one repeating pattern across the whole piece, with no seams.
4. So we melt the polysilicon completely and grow it back as one single crystal.
5. We dip a small perfect seed crystal into the melt and slowly pull it upward while turning it.
6. Liquid silicon freezes onto the seed and copies its arrangement exactly.
7. Keep pulling for a day or more and you get a cylinder of one crystal, called a boule or ingot.
8. A 300 millimetre boule is about 2 metres long and weighs about 265 kilograms.

■ 3.4.2 PLAIN – a picture in your head

1. Think of pulling hot toffee slowly out of a pan so it stretches into a smooth rod instead of breaking.
2. Now add one rule: it can only harden by copying the pattern already above it. The seed is that template.
3. Pull too fast and the pattern breaks. Pull too slow and the rod grows fat. Spinning keeps the heat even all round.

- Where this comparison breaks: toffee just cools. Each silicon atom must find one site in a repeating three-dimensional lattice. And the crucible slowly dissolves into the melt.

■ 3.4.3 PLAIN – a worked example

- Here is a full run for a 300 millimetre crystal.

Charge polysilicon into crucible	about 250 to 450 kg
Melt down under argon	melts at 1414 deg C
Dip the seed crystal	seed a few mm across
Neck: pull thin and fast	neck about 3 mm wide
Grow the crown, widen out	out to 300 mm across
Grow the body	about 2 m of cylinder
Taper the tail and lift clear	whole run 1 to 3 days

- Pull rate through the body is around 0.5 to 1.5 millimetres per minute.
- Crucible and crystal rotate in opposite directions, a few turns per minute each.
- The neck stage looks wasteful but is essential. Pulling thin and fast forces line defects out to the surface, where they vanish.
- That trick came from William Dash at General Electric around 1958. It is why modern crystals can be dislocation-free.

■ 3.4.4 PLAIN – what is really happening inside

- The polysilicon sits in a fused-quartz crucible inside a graphite support, under argon so no air gets in.
- Heaters bring everything above 1,414 degrees Celsius, the melting point of silicon.
- A measured amount of boron or phosphorus goes into the melt now. This is the first doping step.
- As the seed rises, liquid at the interface freezes, and each atom locks into the position dictated by the seed.
- The seed's lattice direction therefore sets the direction of the whole boule. That is crystal orientation.
- Impurities prefer to stay liquid, so the crystal purifies itself as it grows. That is segregation, and it leaves the tail dirtier than the head.
- The quartz crucible slowly dissolves, so oxygen enters the melt and ends up in the crystal. This is unavoidable here.

■ 3.4.5 TECHNICAL – the engineer's version

- The method is the Czochralski process, named for Jan Czochralski, the Polish chemist who described the pulling technique in 1916.
- It was adapted to semiconductors at Bell Telephone Laboratories by Gordon Teal and John Little around 1950.
- Orientation is given by Miller indices. The two common wafer orientations are (100) and (111).
- (100) dominates for modern CMOS logic, because the silicon to silicon-dioxide interface has lower trapped charge on that plane.
- Resistivity is set by melt doping: boron gives p-type; phosphorus, arsenic or antimony give n-type.
- Interstitial oxygen lands near 10 to 20 parts per million atomic. It gives internal gettering, which traps metals away from the device region.

Diameter	Typical body length	Typical mass
150 mm	about 1 to 1.5 m	about 30 to 50 kg
200 mm	about 1.5 to 2 m	about 100 kg
300 mm	about 2 m	about 265 kg

7. The alternative is float-zone growth. A radio-frequency coil melts a narrow band passed along a rod, with no crucible touching the melt.
8. Float-zone silicon has far lower oxygen and much higher resistivity, so it suits power devices, radiation detectors and some radio-frequency parts.
9. It is limited to about 200 millimetres and costs more, so almost all logic and memory wafers are Czochralski.
10. The honest version: a Czochralski crystal is dislocation-free, not defect free. Point defects, vacancy clusters and dissolved oxygen remain.

■ 3.4.6 WORDS - remember these

1. Single crystal — one unbroken repeating pattern of atoms — a monocrystalline solid with no grain boundaries.
2. Grain boundary — the seam where two crystals meet — a planar defect that scatters carriers and traps charge.
3. Boule — the big grown cylinder — the as-grown single-crystal ingot before machining.
4. Czochralski — the dip-and-pull growth method — crucible-based melt pulling, the dominant industrial route.
5. Float zone — crucible-free growth by a moving molten band — FZ silicon, low oxygen, high resistivity, limited diameter.

3.5 From boule to wafer

■ 3.5.1 PLAIN - in simple words

1. We have a two-metre crystal cylinder. Chips are built on thin flat slices of it.
2. First the cone-shaped ends are cut off and thrown back into the melt pot.
3. Then the cylinder is ground on the outside to exactly the right diameter all the way along.
4. A small notch is ground into the side. It tells every later machine which way the crystal pattern faces.
5. Then the cylinder is sliced, using one very long wire coated with cutting grit.
6. Sawing leaves slices rough and damaged, so they are ground flat, etched in acid, and polished.
7. The result is a wafer: a disc about three quarters of a millimetre thick, with a mirror surface.

■ 3.5.2 PLAIN - a picture in your head

1. Think of a very long, expensive salami sliced into perfect discs by a machine that never wobbles.
2. Except the slicer is not a blade. It is one wire, hundreds of kilometres long, wound across grooved rollers.
3. The wire runs at speed carrying hard particles, cutting the whole ingot into hundreds of slices in one pass.
4. Where this comparison breaks: salami only needs to look right. A wafer must be flat to a fraction of a wavelength of light, or the printing step goes out of focus.

■ 3.5.3 PLAIN – a worked example

1. Follow one 300 millimetre boule through the shop.

Boule 2 m, 265 kg	
-> crop the ends	lose about 10 to 20 pct
-> grind to 300.0 mm	lose a few mm of radius
-> grind the notch	one small notch on the edge
-> wire saw	about 900 to 1200 wafers
-> lap and edge-round	remove the saw damage
-> acid etch	remove more damage
-> CMP one side	mirror finish
-> clean and pack	sealed cassettes

2. Slicing loses material as kerf, the width of the cut. Diamond wire kerf is roughly 100 to 150 micrometers.
3. A finished 300 millimetre wafer is 775 micrometers thick, plus or minus about 25. That is standardized, not a guess.
4. So each wafer plus kerf uses close to 0.9 millimetres of boule, giving around one thousand wafers from 2 metres.
5. A blank polished 300 millimetre prime wafer sells for very roughly 100 to 150 US dollars in volume.

■ 3.5.4 PLAIN – what is really happening inside

1. Cropping removes the crown and tail, where diameter is wrong and impurity levels are off.
2. The notch is ground after an X-ray measurement finds the true crystal direction.
3. Older small wafers used a straight flat on the edge. Modern 200 and 300 millimetre wafers use a notch, which wastes less area.
4. In wire sawing the wire does not cut. The abrasive does. Sawing leaves a damaged layer tens of micrometers deep, full of microcracks.
5. Lapping presses wafers between plates with slurry to make both faces parallel and flat. Edge rounding stops chipping and particle shedding.
6. Acid etching removes the crushed layer. Chemical-mechanical polishing then finishes one face with a soft pad and an alkaline silica slurry.
7. Final cleaning ends with drying in filtered air. From here the wafer never touches ordinary air again.

■ 3.5.5 TECHNICAL – the engineer's version

1. Wafer geometry is set by SEMI standards: diameter, thickness, flatness, edge profile and notch dimensions.

Diameter	Thickness	Area	Notes
150 mm (6 in)	675 um	177 sq cm	Legacy, power, MEMS
200 mm (8 in)	725 um	314 sq cm	Analog, auto, RF
300 mm (12 in)	775 um	707 sq cm	All leading-edge
450 mm (18 in)	925 um	1590 sq cm	Never commercialized

2. Thickness rises with diameter for stiffness. A thin 300 millimetre wafer would sag and crack under its own weight.
3. Going from 200 to 300 millimetres multiplies usable area by about 2.25 for less than 2.25 times the cost. That is the case for larger wafers.

4. The 450 millimetre transition stalled. The Global 450 Consortium, based in Albany, New York, effectively collapsed around 2016 and 2017.
5. Reported reasons: huge tool redevelopment cost, EUV absorbing the industry's capital, and reluctance to fund a change helping mainly the largest players.
6. As of 2026 there is no credible 450 millimetre roadmap.
7. Processed wafer prices at leading nodes, reported by analysts in early 2026. These are negotiated and vary by customer.

Node	Approx price per wafer
28 nm	about 3,000 USD
7 nm	about 9,500 USD
5 nm class	about 18,500 USD
2 nm class	about 30,000 USD

8. The honest version: the raw silicon disc is a rounding error. At the 2 nanometre class it is well under 1 percent of the finished wafer price.
9. When people say “sand is cheap, so chips should be cheap”, this table is the answer. The material is cheap. The processing is not.

■ 3.5.6 WORDS – remember these

1. Kerf — the material lost to the cut — the saw slot width, roughly 100 to 150 micrometers with diamond wire.
2. Notch — the alignment mark on the wafer edge — a SEMI-specified notch replacing the older primary flat.
3. CMP — polishing with chemistry and rubbing together — chemical-mechanical planarization, using a pad and reactive slurry.
4. Prime wafer — the best grade sold — a polished wafer meeting full flatness, particle and defect specification.

3.6 The cleanroom

■ 3.6.1 PLAIN – in simple words

1. The features on a modern chip are far smaller than a speck of dust.
2. One particle in the wrong place can short two wires or block a printing step, and kill that chip.
3. So chip factories are built inside rooms far cleaner than an operating theatre.
4. Air is pushed down through ceiling filters, constantly, and pulled out through the floor.
5. The dirtiest thing in the room is the human. People wear sealed suits, called bunny suits, that keep skin flakes and breath in.
6. The building must be still, because even small vibrations blur the printing step.
7. The rinse water is purer than drinking water by an enormous margin.
8. All of this is why a leading-edge fab costs billions before it makes a single chip.

■ 3.6.2 PLAIN – a picture in your head

1. Picture an operating theatre, made a hundred times cleaner, inside a concrete box the size of several football pitches.
2. The whole ceiling is a filter. Air falls straight down like an invisible waterfall, sweeping particles to the floor.

3. Wafers do not ride through the open room. They travel in sealed pods on overhead rails and dock onto machines.
4. Where this comparison breaks: a theatre worries about living germs. A fab does not care whether a particle is alive. It cares about size.

■ 3.6.3 PLAIN – a worked example

1. Suppose one chip is 1 square centimetre, and the process leaves 0.1 killer defects per square centimetre.
2. A simple model gives a good fraction of e to the power of minus 0.1, about 0.90. So 90 percent survive.
3. Now make the chip ten times bigger, 10 square centimetres, like a large graphics processor.
4. The same defect density gives e to the power of minus 1, about 0.37. Only 37 percent survive.
5. Same factory, same wafer. Bigger chips are punished brutally, which drives the move to chiplets in section 3.10.

■ 3.6.4 PLAIN – what is really happening inside

1. Cleanliness is measured by counting particles in a fixed volume of air, at a fixed particle size.
2. The cleanest areas allow about ten particles of 0.1 micrometer or larger in a whole cubic metre of air.
3. To hold that, room air is replaced hundreds of times per hour, needing enormous fan power running permanently.
4. Not all of the fab is equally clean. The tightest classes apply where wafers are exposed, especially at the lithography tools.
5. Water is purified until it barely conducts, then deoxygenated and filtered again just before use. A big fab uses millions of litres a day.
6. Vibration matters because the printing tool must hold alignment to a few nanometres while a heavy stage moves fast.
7. So the lithography bay sits on an isolated slab with active dampers, and the fab is not built next to a railway line.

■ 3.6.5 TECHNICAL – the engineer's version

1. Air cleanliness is classified by ISO 14644-1. The older United States standard, Federal Standard 209E, was withdrawn in 2001.

ISO class	Max particles per cu m at 0.1 μ m	Air changes per hour
ISO 1	10	500 to 750
ISO 2	100	500 to 750
ISO 3	1,000	500 to 750
ISO 5	100,000	250 to 300

2. ISO 3 corresponds roughly to the old Class 1, and ISO 5 to the old Class 100.
3. Modern practice is a mini-environment strategy. The bay runs at ISO 5 or 6 while the wafer sees ISO 1 inside a sealed pod and inside the tool.
4. Wafers travel in a Front Opening Unified Pod, a SEMI-standard carrier, moved by an overhead hoist transport system.
5. Ultrapure water is specified at 18.2 megaohm-centimetre at 25 degrees Celsius, the theoretical maximum for pure water.

6. Molecular contamination is controlled too. Ammonia poisons chemically amplified photoresist, so lithography bays use chemical filtration.
7. Vibration is specified against generic vibration criteria curves. Advanced lithography demands VC-E or better, near 3 micrometres per second RMS.
8. TSMC stated in 2025 and 2026 that a 2 nanometre fab module of about 20,000 wafer starts per month costs roughly 25 to 35 billion US dollars.
9. TSMC guided 2026 capital spending to roughly 60 to 64 billion US dollars. Most of that is tools; a single EUV scanner can exceed 200 million dollars.
10. Standard, convention or implementation detail: ISO 14644-1 is a standard, the bunny suit is a convention, and each bay's class is an implementation detail of that fab.

■ 3.6.6 WORDS – remember these

1. Cleanroom — a room with almost no dust — a controlled environment classified by ISO 14644-1 particle limits.
2. Bunny suit — the full-body coverall — a cleanroom garment containing human-generated particles and fibres.
3. FOUP — the sealed wafer box — Front Opening Unified Pod, the standard 300 millimetre carrier.
4. Ultrapure water — water with nothing else in it — UPW at 18.2 megaohm-centimetre at 25 degrees Celsius.
5. Defect density — how many killer flaws per area — D0 in defects per square centimetre, the main input to yield models.

3.7 Photolithography, the heart of it

■ 3.7.1 PLAIN – in simple words

1. A chip is a stack of patterned layers. Photolithography puts the pattern on each layer.
2. It is printing with light, and it is the most important step in the factory.
3. First we coat the wafer with a liquid that hardens into a thin film and reacts to light. That is photoresist.
4. We spin the wafer very fast so the liquid spreads into an even layer, much thinner than a hair.
5. Then we shine light through a patterned plate, called a mask or reticle, onto the resist.
6. Where light lands, the resist changes chemically. A developer washes away either the lit parts or the unlit parts.
7. We etch through the holes left behind, then strip the remaining resist off.
8. That is one layer. A modern chip repeats this cycle, with different masks, many dozens of times.

■ 3.7.2 PLAIN – a picture in your head

1. Think of spray painting a wall through a cardboard stencil. The mask is the stencil; the resist records where the spray hit.
2. Now change two things. The stencil is four times bigger than the pattern you want, and a lens shrinks the image on the way down.
3. And the stencil covers only one small rectangle. The machine prints it, steps sideways, prints again, and repeats across the wafer.
4. That is why the tool is a stepper, or a scanner when it sweeps rather than flashes.
5. Where this comparison breaks: paint stays where it lands. Photoresist never becomes part of the chip. It is destroyed and removed on every layer.

■ 3.7.3 PLAIN – a worked example

1. Here is one complete layer cycle, in order.

1	clean and prime	make surface accept resist
2	spin coat resist	3000 rpm, film 30 to 200 nm
3	soft bake	drive off solvent
4	align to previous	match marks already on wafer
5	expose	light through the reticle
6	post-exposure bake	drive the resist chemistry
7	develop	wash away the soluble parts
8	inspect and measure	check width and overlay
9	etch	cut into the layer below
10	strip resist	remove the mask and clean

2. Now the print-and-repeat step. A standard exposure field is 26 by 33 millimetres, which is 858 square millimetres.
3. A 300 millimetre wafer is about 70,700 square millimetres, so the machine prints roughly 80 fields to cover one wafer.
4. If each field holds four dies of about 200 square millimetres, one wafer carries about 320 dies for that layer.
5. Then it does the whole thing again for the next layer. And again.

■ 3.7.4 PLAIN – what is really happening inside

1. The resist is not a simple dye. In modern deep ultraviolet resists, light creates a small amount of acid inside the film.
2. The bake after exposure lets that acid move and trigger a chain reaction, changing the solubility of the polymer around it.
3. That amplification is why modest light flips a whole film. It is also why stray ammonia in the air ruins the pattern.
4. Alignment is separate and just as hard. Each new layer must line up with the layers already there, to a few nanometres.
5. The machine reads marks already on the wafer and adjusts stage position, rotation and scale before printing. The error left over is overlay.
6. Etching transfers the pattern downward. Wet etching uses liquids and eats sideways as well as down.
7. Dry etching uses a plasma, an electrically excited gas, and cuts almost straight down. That is why small features need it.

■ 3.7.5 TECHNICAL – the engineer's version

1. Resolution follows the Rayleigh criterion.

```

CD = k1 * wavelength / NA
DOF = k2 * wavelength / (NA * NA)

CD = smallest printable half-pitch
NA = numerical aperture of the lens
k1 = process factor, hard physical floor at 0.25
DOF = depth of focus, the focus error allowed

```

2. Three levers exist: shorter wavelength, higher numerical aperture, lower k1.
3. Single exposure cannot go below k1 of 0.25. That physical limit is the reason multi-patterning exists.

4. Note the second formula. Numerical aperture costs depth of focus quadratically, which is why wafer flatness is specified so tightly.
5. Reticles are typically 4x reduction, so 60 nanometres on the mask prints at 15 nanometres on the wafer.
6. Optical proximity correction distorts the mask shapes so the printed result is correct. Sub-resolution assist features help their neighbours print.
7. The honest version: the mask is not a picture of the chip. It is the computed input that yields the chip after diffraction.
8. History: the first commercial wafer stepper was the GCA DSW4800, introduced in 1978. Step-and-scan replaced steppers at advanced nodes in the 1990s.
9. A leading-node mask set is commonly quoted at 10 to 30 million US dollars. That is the main barrier to low-volume advanced chips.

Term	Meaning	Typical value
Field size	One printed rectangle	26 by 33 mm
Reduction	Mask to wafer scale	4x
Overlay	Layer to layer error	1 to 3 nm
Throughput	Wafers per hour	100 to 220

■ 3.7.6 WORDS – remember these

1. Photoresist — light-sensitive coating — a polymer film whose solubility changes on exposure, via a photoacid generator.
2. Reticle — the patterned plate — a quartz plate with an absorber pattern, imaged at 4x reduction onto the wafer.
3. Stepper and scanner — print-and-repeat machines — step-and-repeat or step-and-scan exposure tools working field by field.
4. Numerical aperture — how widely the lens gathers light — NA, appearing in the Rayleigh resolution and depth of focus equations.
5. Overlay — how well layers line up — registration error between a printed layer and the layers beneath it.
6. OPC — deliberately distorting the mask — optical proximity correction, a computational step applied before mask writing.

3.8 The light

■ 3.8.1 PLAIN – in simple words

1. The smallest thing you can print depends on the wavelength of your light. Shorter waves draw finer lines.
2. Chip making has therefore been a long march toward shorter wavelengths.
3. It started with mercury lamps giving ultraviolet light at 436 and then 365 nanometres.
4. Then came gas lasers at 248 nanometres, then 193 nanometres. These are deep ultraviolet.
5. Progress stalled at 193, so engineers filled the gap between lens and wafer with water.
6. Water bends light more than air, which shrinks the effective wavelength in the gap. That is immersion lithography.
7. When that ran out, they printed one layer more than once with offset patterns. That is multi-patterning.
8. Finally came extreme ultraviolet at 13.5 nanometres, fourteen times shorter than 193.

■ 3.8.2 PLAIN – a picture in your head

1. Imagine drawing with a marker pen. A thick pen cannot draw thin lines, no matter how steady your hand.
2. Wavelength is the thickness of the pen tip. To draw finer, you need a finer tip.
3. Immersion is like drawing underwater, where the ink spreads less. You get a finer line from the same pen.
4. Multi-patterning is drawing every second line, letting it dry, then going back for the ones in between.
5. Extreme ultraviolet is not a finer pen. It is a different tool entirely, which no glass lens can focus.
6. Where this comparison breaks: a pen deposits ink. Light deposits nothing. The limit comes from diffraction, how waves spread past an edge.

■ 3.8.3 PLAIN – a worked example

1. Put real numbers into the Rayleigh formula from section 3.7: half-pitch equals k_1 times wavelength divided by numerical aperture.
2. Dry 193 nanometre tool, numerical aperture 0.93, k_1 of 0.30: about 62 nanometres.
3. Immersion 193 nanometre tool in water, numerical aperture 1.35, since water has refractive index near 1.44 here: about 43 nanometres.
4. EUV at 13.5 nanometres, numerical aperture 0.33: about 12 nanometres. High numerical aperture EUV at 0.55: about 7 nanometres.
5. Notice the jump. Moving to EUV bought more resolution in one step than the previous twenty years of optical improvement.

■ 3.8.4 PLAIN – what is really happening inside

1. Here is how EUV light is actually made. It is not a lamp and not a normal laser.

```

tin droplet generator
  | drops of molten tin, about 25 um across
  | released about 50,000 times per second
  v
[ pre-pulse laser ] -> flattens the drop into a disc
  |
  v
[ main pulse laser ] -> vaporizes it into hot plasma
  |
  v
plasma over 200,000 degrees emits 13.5 nm light
  |
  v
collector mirror -> multilayer mirrors -> reticle
  |
  v
more mirrors -> wafer

```

2. A carbon dioxide laser of over ten kilowatts average power fires twice at each falling droplet.
3. The first shot squashes the sphere flat. The second turns it into plasma, an extremely hot ionized gas radiating at 13.5 nanometres.
4. Everything absorbs this light. Air absorbs it, glass absorbs it. So there are no lenses, and the path must be in vacuum.
5. Instead the machine uses mirrors of about forty to fifty alternating pairs of molybdenum and silicon, each layer a few nanometres thick.

6. Even those reflect only around 70 percent. After ten bounces most of the light is gone, which is why source power is such a struggle.
7. The reticle is a mirror too, not a transparent plate. That is a fundamental break from all earlier lithography.
8. Tin debris coats the collector and shortens its life, so hydrogen gas is flowed through to clean it.

■ 3.8.5 TECHNICAL – the engineer’s version

Generation	Wavelength	Source	Leading use
g-line	436 nm	Mercury lamp	1980s
i-line	365 nm	Mercury lamp	late 1980s to 90s
KrF DUV	248 nm	Krypton fluoride	mid 1990s to 2000s
ArF DUV	193 nm	Argon fluoride	2001 onward
ArF immersion	193 nm in water	ArF plus water	2007 onward
EUV	13.5 nm	Tin plasma	2019 onward

1. The 157 nanometre fluorine laser generation was researched hard and abandoned around 2003, because 193 immersion proved cheaper and better.
2. Immersion entered volume production around 2007, with water enabling numerical apertures up to 1.35.
3. EUV entered high-volume manufacturing in 2019, first at Samsung and TSMC for 7 nanometre class layers.
4. Low numerical aperture tools such as the ASML TWINSCAN NXE:3800E have numerical aperture 0.33 and are reported at 180 to 220 million US dollars.
5. High numerical aperture EUV, the TWINSCAN EXE:5200 family, has numerical aperture 0.55 and is reported at about 380 million US dollars per tool.
6. Anamorphic means the reduction differs by axis, 4x one way and 8x the other. The field halves to about 26 by 16.5 millimetres, so large dies are stitched.
7. ASML of Veldhoven, the Netherlands, is the only supplier of EUV scanners.
8. Zeiss SMT supplies the optics and TRUMPF the carbon dioxide laser, both of Germany. The tin source traces to Cymer, now part of ASML.
9. Where experts disagree: whether High-NA EUV beats low-NA plus double patterning on cost per layer. Intel has pushed High-NA; some analysts disagree. As of 2026 this is unsettled.
10. Established fact: EUV works and is in volume production. Active research: higher source power and dry resists. Marketing claim: any statement that a named node “requires” High-NA today.

■ 3.8.6 WORDS – remember these

1. Deep ultraviolet — the 248 and 193 nanometre light — DUV from krypton fluoride and argon fluoride excimer lasers.
2. Immersion lithography — printing through water — a water film between final lens and wafer, raising NA above 1.
3. Multi-patterning — printing one layer in several passes — LELE, SADP and SAQP decomposition of a dense layer.
4. Extreme ultraviolet — 13.5 nanometre light — EUV, in the soft X-ray region, needing vacuum and reflective multilayer optics.
5. Laser-produced plasma — light made by vaporizing tin — LPP source, a dual carbon dioxide laser pulse at 50 kilohertz.

6. High-NA — the next EUV generation — numerical aperture 0.55 anamorphic optics with a halved exposure field.

3.9 The other steps

■ 3.9.1 PLAIN – in simple words

1. Printing patterns is half the story. We also add and change material.
2. Doping means adding a tiny controlled amount of a foreign element to change how the silicon conducts.
3. The main method is ion implantation. We turn the dopant into charged atoms, accelerate them, and fire them into the wafer.
4. That smashes the crystal, so afterwards we heat the wafer briefly. The lattice repairs and the dopants settle into place.
5. Oxidation means growing a glass layer by heating silicon in oxygen. The wafer rusts, in a useful and controlled way.
6. Then three ways of laying material on top: chemical deposition, physical deposition, and one-atomic-layer-at-a-time deposition.
7. Finally the transistors are wired together, in a stack of copper layers above them, fifteen or more.
8. A modern chip goes through many hundreds of process steps and spends about three months inside the fab.

■ 3.9.2 PLAIN – a picture in your head

1. Think of building a city on a plain, where you may only work from above and must finish each floor before the next.
2. Doping is changing the soil chemistry in specific plots so buildings there behave differently.
3. Chemical vapour deposition is a fine mist that reacts and settles everywhere, even inside narrow trenches.
4. Physical vapour deposition is shot-blasting from one direction. It coats what it can see and struggles down deep holes.
5. Atomic layer deposition is placing one tile at a time across the whole city. Very slow, very exact.
6. The metal stack is the road network built above the buildings, with vertical lifts between levels.
7. Where this comparison breaks: a city is built once. Wafer layers are built, partly cut away and rebuilt, and nothing buried is reachable again.

■ 3.9.3 PLAIN – a worked example

1. Here is the damascene method, which is how the copper wiring is made.

Start: a flat insulating layer over the transistors

- 1 etch a trench and a via hole into the insulator
- 2 line the hole with a thin barrier film (TaN/Ta)
- 3 sputter a thin copper seed layer
- 4 electroplate copper until it overflows the trench
- 5 polish the whole surface flat with CMP
- 6 the only copper left is inside the trench

Result: a wire buried in glass, top surface perfectly flat

2. Notice what did not happen. We never etched copper. Copper is hard to etch cleanly, so we cut the shape first and fill it.

3. That is damascene, named after the inlay metalwork of Damascus. Doing trench and vertical connection in one fill is dual damascene.
4. Repeat fifteen or more times, with wires getting wider and thicker as you go up.
5. Bottom layers carry signals inside one circuit block. Top layers are thick and carry power across the chip.

■ 3.9.4 PLAIN – what is really happening inside

1. Ion implantation: a gas of boron, phosphorus or arsenic is ionized, sorted by mass with a magnet, then accelerated.
2. Energies run from under one thousand electron volts for shallow layers up to millions for deep wells.
3. Dose is counted as ions per square centimetre, typically ten to the twelve through ten to the sixteen, and is very repeatable.
4. Annealing: the wafer is heated in seconds, or milliseconds, to around 1,000 degrees Celsius, to repair damage and activate dopants.
5. It must be brief, because heat also makes dopants diffuse and smear the tiny features we just made.
6. Oxidation: heating silicon in oxygen or steam at 800 to 1,200 degrees Celsius grows silicon dioxide, consuming some silicon.
7. Chemical vapour deposition: gases decompose on the hot wafer. Physical vapour deposition: an argon plasma knocks atoms off a target onto the wafer.
8. Atomic layer deposition: two gases are fed alternately, never together. Each pulse adds at most one atomic layer, so thickness is counted in cycles.

■ 3.9.5 TECHNICAL – the engineer’s version

Method	How it works	Typical use
Thermal oxidation	Grows from substrate	Gate oxide, isolation
LPCVD and PECVD	Gas reacts on surface	Nitride, oxide, poly
PVD sputtering	Ions knock off target	Barriers, seed, pads
ALD	One layer per cycle	High-k, liners, spacers

1. The gate insulator was silicon dioxide for about forty years. Intel replaced it with hafnium-based high-k plus metal gates at 45 nanometres in 2007.
2. The honest version: since 2007 “gate oxide” is largely historical. The layer is a hafnium oxide film laid by atomic layer deposition.
3. Interconnect switched from aluminum to copper. IBM announced copper interconnect in 1997, using dual damascene with CMP.
4. Copper needs a barrier, historically tantalum nitride and tantalum, because copper diffuses into silicon and poisons devices.
5. Dielectrics moved from silicon dioxide, near 3.9, to carbon-doped low-k films near 2.5 to 3.0, to cut wire capacitance.
6. Advanced logic in 2026 uses 15 to 20 metal levels, from roughly 20 to 30 nanometre pitch at the bottom to micrometres at the top.
7. Backside power delivery is the current change. Intel calls its version PowerVia, shipped in 18A; TSMC’s equivalent arrives with A16.
8. Step counts: advanced logic is commonly described as more than 1,000 process steps with roughly 80 to 100 mask layers. Exact counts are confidential, so treat these as approximate.

9. Cycle time: a leading-edge logic wafer spends about 3 months in the fab, often quoted as 12 to 16 weeks. With packaging and test, four to five months.

■ 3.9.6 WORDS – remember these

1. Doping — adding a trace element to change conduction — introducing acceptors or donors to set carrier type and concentration.
2. Ion implantation — firing dopant atoms into the wafer — mass-analyzed ion beam implantation, dose in ions per square centimetre.
3. Annealing — a short hot step to repair and settle — rapid thermal, spike, flash or laser anneal for damage repair and activation.
4. ALD — one atomic layer per cycle — atomic layer deposition, self-limiting surface reactions giving exact conformal thickness.
5. Damascene — cut the shape then fill it — trench-first copper metallization finished by CMP, in single and dual forms.
6. High-k — a better gate insulator than glass — a high permittivity dielectric, usually hafnium based, in use since 2007.

3.10 Testing, dicing, packaging

■ 3.10.1 PLAIN – in simple words

1. At the end of the fab, the wafer holds hundreds of finished chips, still joined in one disc. Some are broken.
2. So we test every one, while they are still attached.
3. A machine lowers a head with hundreds of fine needles onto each chip, sends in signals, and checks the answers.
4. Bad chips are marked for scrap. Good chips are sorted by how well they performed, which is called binning.
5. The same design, from the same wafer, can sell as an expensive fast part or a cheaper slow part.
6. Then the wafer is cut into individual chips. Each is glued into a package, connected electrically, and covered.
7. The package is not just protection. It carries power in, carries heat out, and fans tiny pads out to board-sized contacts.

■ 3.10.2 PLAIN – a picture in your head

1. Think of biscuits baked as one sheet, which must be checked, graded, cut and boxed.
2. Testing is tasting each one before cutting. Cheaper than boxing a bad one and finding out later.
3. Binning is grading: perfect ones in the premium box, uneven ones in the value box, from the same recipe.
4. Modern high-end products do not use one big biscuit. They use several smaller ones side by side, connected together.
5. Where this comparison breaks: biscuits need not line up to a micrometer, and none needs 1,000 watts of heat carried off a postage stamp.

■ 3.10.3 PLAIN – a worked example

1. Take a 300 millimetre wafer, about 70,700 square millimetres, and a die of 100 square millimetres.
2. A circle cannot be tiled perfectly by rectangles, so you get about 600 full dies, not 707.
3. Apply defects at 0.1 per square centimetre and about 90 percent are good, so about 540 good dies.
4. If the processed wafer cost 18,500 US dollars, that is about 34 US dollars per good die.

Die area	Dies per wafer	Yield	Cost per good die
50 sq mm	about 1,240	95 pct	about 16 USD
100 sq mm	about 600	90 pct	about 34 USD
200 sq mm	about 290	82 pct	about 78 USD
600 sq mm	about 88	55 pct	about 380 USD

5. Doubling die area more than doubles cost per working chip. You lose twice over: fewer dies and worse yield.
6. That table, and nothing else, explains why the industry moved to chiplets.

■ 3.10.4 PLAIN – what is really happening inside

1. Wafer probe, or wafer sort, is the first electrical test. A probe card with hundreds of needles contacts the pads, often at several voltages and temperatures.
2. Dicing separates the dies. A diamond blade saw is traditional. Laser stealth dicing makes a weak plane inside the silicon, then the wafer is stretched and splits.
3. Die attach fixes the die down, face up on a pad or face down onto a substrate.
4. Wire bonding joins the die pads to the package with fine gold or copper wires, welded by heat and ultrasound. Cheap and mature.
5. Flip-chip turns the die upside down and connects through solder bumps across the whole face, allowing many more connections and shorter power paths.
6. Above the die goes a lid or heat spreader, joined by a thermal interface material: a paste, a pad, or soldered metal.
7. Underneath sits the substrate, a small multilayer board fanning fine connections out to larger solder balls or pins.
8. Chiplets change the picture. Several dies sit on a shared carrier and behave as one product.

■ 3.10.5 TECHNICAL – the engineer’s version

1. Yield is modelled from defect density. Two standard forms are used.

```
Poisson:  $Y = \exp(-A * D0)$ 
Murphy:  $Y = ((1 - \exp(-A * D0)) / (A * D0)) ^ 2$ 

A = die area in square centimetres
D0 = defect density per square centimetre
```

2. Murphy’s model is usually closer to reality, because defects cluster rather than spreading evenly.
3. Mature high-volume nodes run D0 near 0.05 to 0.1 per square centimetre. A new node starts far worse and improves over quarters.
4. Reported N2 test-chip yields in early 2026 were about 70 to 80 percent for TSMC, with Intel and Samsung lower. These are press reports, not disclosure.
5. Binning sorts by maximum stable frequency, leakage, and functional core count. Fusing off a defective core is normal and deliberate.
6. The reticle limit sets the largest single die at about 26 by 33 millimetres, 858 square millimetres, at low numerical aperture EUV.
7. 2.5D packaging places multiple dies on a silicon interposer. TSMC’s platform is CoWoS, introduced in 2012; Intel’s bridge alternative is EMIB.
8. 3D packaging stacks dies with through-silicon vias. Examples are Intel Foveros, announced 2018, and TSMC SoIC.

9. High Bandwidth Memory is a JEDEC standard: stacked DRAM joined by through-silicon vias, placed beside the logic die. HBM4 was standardized in 2025.

Product	Year	Transistors
NVIDIA H100	2022	80 billion
NVIDIA B200	2024	208 billion, 2 dies
Cerebras WSE-3	2024	4 trillion
NVIDIA Rubin	announced 2026	336 billion, 4 tiles

10. The known good die problem is central. Stacking three good dies with one bad one wastes all four, so pre-assembly test coverage must be very high.
11. Advanced packaging capacity, not wafer capacity, has been the binding constraint on artificial intelligence accelerator supply since 2023.

■ 3.10.6 WORDS – remember these

1. Wafer probe — testing chips before cutting — wafer sort using a probe card and automated test equipment.
2. Yield — the share of chips that work — good die over gross die, modelled from area and defect density.
3. Binning — sorting identical chips into product grades — speed, power and core-count sorting of one die design.
4. Flip-chip — die mounted face down on bumps — controlled collapse chip connection, giving area-array input and output.
5. Interposer — a carrier that wires dies together — a silicon or organic substrate with fine-pitch routing, the basis of 2.5D.
6. Chiplet — a small die that is part of a bigger product — a partitioned die assembled with others into one package.
7. HBM — stacked memory beside the processor — High Bandwidth Memory, a JEDEC standard using through-silicon-via DRAM stacks.

3.11 What “3 nm” actually means now

■ 3.11.1 PLAIN – in simple words

1. Chip generations have names like 7 nanometre, 5 nanometre, 3 nanometre, 2 nanometre.
2. Most people assume the number is the size of something on the chip. It is not, and has not been for over a decade.
3. Today it is a generation label meaning “our next, better process”, and each company picks its own numbers.
4. A silicon atom is about 0.2 nanometres across. Nothing useful can be three atoms wide and still work as a transistor.
5. On a real 3 nanometre process, the spacing between neighbouring gates is about 45 nanometres, fifteen times the name.
6. What did change, really and physically, is the shape of the transistor.
7. It went from flat, to a standing fin gripped on three sides, to stacked ribbons with the gate wrapped all the way around.
8. Those shape changes are real engineering. The numbers on the box are marketing.

■ 3.11.2 PLAIN – a picture in your head

1. Think of a garden hose with a hand squeezing it. The hand is the gate; the water is the current.
2. Flat transistors are a hose on the ground with one hand pressing from the top. It works, but water leaks past.
3. A FinFET stands the hose on edge so the hand grips three sides. Much better control, much less leak.
4. Gate-all-around wraps the hand fully around the hose, with several thin hoses stacked above one another.
5. That is why the industry moved: at small sizes a flat gate simply stopped being able to turn the current off.
6. Where this comparison breaks: no water flows. Current is carried by electrons or holes, and “off” means suppressing quantum tunnelling and thermal leakage.

■ 3.11.3 PLAIN – a worked example

1. Here is the growth in transistor count, with real parts and real years.

Chip	Year	Transistors
Intel 4004	1971	about 2,300
Intel 8086	1978	29,000
Intel 80386	1985	275,000
Intel Pentium	1993	3.1 million
Pentium 4	2000	42 million
Apple M4	2024	28 billion
NVIDIA B200	2024	208 billion
Cerebras WSE-3	2024	4 trillion

2. From 2,300 to 208 billion is a factor of about 90 million, in 53 years.
3. The honest version on the 4004: Intel has long quoted 2,300, and that is the number in most books. Careful die counts give about 2,250.
4. A second honesty note: Apple publishes no transistor count for every part. None was published for the A19 Pro of 2025, so the table uses the M4.
5. The Cerebras WSE-3 is a special case. It is not diced at all. It is one square cut from a wafer, about 46,225 square millimetres, with 900,000 cores.

■ 3.11.4 PLAIN – what is really happening inside

1. Originally the node name did mean something. In the 1970s and 1980s it tracked the gate length.
2. Then it drifted to meaning half the spacing between the first metal wires.
3. Then, from roughly the 22 and 20 nanometre generations, it stopped tracking anything measurable.
4. Companies chose different numbering, so an Intel node and a TSMC node with the same name were not the same thing.
5. Intel renamed its own nodes in 2021 to line up with the competition, which is itself an admission that the numbers are labels.
6. Look instead at transistor density, in millions of transistors per square millimetre, plus the gate and metal pitches.
7. Even density is arguable, because it depends on which cell library you count and what mix of logic you assume.

■ 3.11.5 TECHNICAL – the engineer’s version

Node	Gate pitch	Density	Structure
TSMC N3	about 45 nm	about 197 MTr/sq mm	FinFET
Samsung 3GAE	about 40 nm	about 150 MTr/sq mm	Nanosheet GAA
TSMC N2	not disclosed	about 15 pct over N3E	Nanosheet GAA
Intel 18A	not disclosed	not disclosed	RibbonFET GAA

1. Planar bulk MOSFET dominated from the 1960s to about 2011.
2. The FinFET concept was demonstrated as DELTA by Hisamoto and colleagues at Hitachi in 1989.
3. It was then developed and named FinFET at the University of California, Berkeley around 1999, by a team including Chenming Hu, Tsu-Jae King Liu and Jeffrey Bokor.
4. Intel put it into volume production first, as Tri-Gate at 22 nanometres, announced 2011 and shipping 2012.
5. Gate-all-around nanosheet reached mass production first at Samsung, with 3GAE in 2022.
6. TSMC moved to nanosheet at N2, starting mass production in late 2025 at Fab 20 in Hsinchu and ramping through 2026.
7. Reported N2 wafer price is about 30,000 US dollars, against roughly 20,000 to 25,000 for 3 nanometre class.
8. Intel 18A, with RibbonFET gate-all-around plus PowerVia backside power, entered high-volume manufacturing in 2025 with Panther Lake.
9. Where experts disagree: how to compare nodes fairly. Density metrics weighted across cell types are contested, and there is no neutral referee.
10. Established fact: nanosheet is in production in 2026. Active research: complementary FET, stacking n-type and p-type vertically, and two-dimensional channel materials. Marketing claim: node names as measurements.
11. The honest version: if someone says a 3 nanometre chip has 3 nanometre features, they are repeating a press release. Ask for gate pitch and density.

■ 3.11.6 WORDS – remember these

1. Node — a process generation — a named technology offering, no longer tied to any physical dimension.
2. Gate length — how far current travels under the gate — the physical channel length, now decoupled from the node name.
3. Contacted poly pitch — spacing between transistor gates — CPP, a real comparable dimension, about 45 nanometres at TSMC N3.
4. FinFET — a transistor standing on edge — a fin channel with the gate on three sides, in production from 2011.
5. Gate-all-around — the gate wrapped fully around — GAA nanosheet or ribbon channels, in production from 2022.

3.12 Who makes chips and why it matters

■ 3.12.1 PLAIN – in simple words

1. Designing a chip and making a chip used to be the same business. Not any more.
2. Most chip companies do not own a factory. They design and pay someone else to build. That is the fabless model, and the builder is a foundry.

3. The largest foundry by far is TSMC, in Taiwan. It builds for companies that compete with each other and sells no branded chips.
4. Samsung, in South Korea, is both a foundry and a chip company of its own.
5. Intel, in the United States, historically built only its own chips, and is now trying to be a foundry for others too.
6. There is a second layer of licensing. ARM, in the United Kingdom, makes no chips. It designs instruction sets and cores, and licenses them.
7. RISC-V is a newer alternative, where the instruction set itself is open and free to use.
8. The important fact is concentration. Very few buildings on Earth can make the most advanced chips.

■ 3.12.2 PLAIN – a picture in your head

1. Think of book publishing. An author writes, a printing plant prints, and a typeface company licenses the letterforms.
2. Fabless chip companies are the authors. Foundries are the printing plants, owning the expensive machines and printing for whoever pays.
3. ARM is the typeface company. Nobody buys a typeface alone, but almost every book uses one, and the fee is small per copy and huge in total.
4. RISC-V is a typeface released free for anyone to use and modify.
5. Where this comparison breaks: there are thousands of printing plants. There are fewer than five organizations that can print at the leading edge, and one supplier of the key machine.

■ 3.12.3 PLAIN – a worked example

1. Trace an Apple phone chip from idea to shipment.
2. Apple designs it in California, using an ARM instruction set under an architecture licence, then sends layout files to TSMC.
3. TSMC makes masks, then runs wafers in Taiwan on ASML machines from the Netherlands, with optics from Germany.
4. The blank wafers most likely came from Japan, where Shin-Etsu and SUMCO dominate 300 millimetre supply.
5. Finished wafers are tested, diced and packaged, often in Taiwan, then assembled into phones elsewhere in Asia.
6. Not one country in that chain could complete it alone. That is the point.

■ 3.12.4 PLAIN – what is really happening inside

1. The reason for the split is cost. A leading-edge fab now costs 25 to 35 billion US dollars and must be upgraded every few years.
2. Only a company running many customers' products can keep such a fab full enough to pay for itself.
3. So designers stopped buying fabs, and the surviving fab owners got larger and fewer.
4. This is not a conspiracy. It follows from rising fixed costs in a business that needs volume to cover them.
5. The same logic applies at ASML, where the research cost of EUV was so large that only one company saw it through.
6. Concentration then becomes a security problem, because one earthquake or export restriction can affect the whole world's supply.
7. Governments responded with subsidy programmes, including the United States CHIPS and Science Act of 2022 and the European Chips Act of 2023.

■ 3.12.5 TECHNICAL – the engineer’s version

Company	Role	Base
TSMC	Pure-play foundry	Taiwan
Samsung Foundry	Foundry plus own chips	South Korea
Intel Foundry	IDM plus foundry	United States
ASML	EUV and DUV scanners	Netherlands

1. TSMC was founded in 1987 by Morris Chang and created the pure-play foundry model. Before that, building for others was a side business.
2. Early fabless companies include Xilinx, founded 1984, Qualcomm, founded 1985, and NVIDIA, founded 1993.
3. ARM was founded in 1990 as Advanced RISC Machines, a joint venture of Acorn Computers, Apple and VLSI Technology.
4. ARM licences come in two tiers: core licences, where you use ARM’s design, and architecture licences, where you build your own compatible core. Apple holds the latter.
5. RISC-V began at the University of California, Berkeley in 2010. RISC-V International moved its legal home to Switzerland in 2020.
6. Important distinction: RISC-V being open does not make any particular RISC-V chip open. The instruction set is open; implementations may be proprietary.
7. Market share: TSMC held roughly 67 to 71 percent of the global foundry market in early 2026, depending on the analyst. Its leading-edge share is higher.
8. In materials, Shin-Etsu Chemical and SUMCO, both Japanese, supply most of the world’s 300 millimetre polished wafers.
9. TSMC’s Arizona site began 4 nanometre class production in 2024. The company has announced about 265 billion US dollars of United States investment.
10. Established fact: leading-edge capacity is concentrated in Taiwan and South Korea. Active development: Arizona, Japan and Dresden fabs ramping. Marketing claim: that any subsidy programme has already ended that concentration.
11. To see part of this chain yourself: on Linux, `lscpu` reports vendor and microarchitecture; on macOS, `sysctl -n machdep.cpu.brand_string` names the part.

■ 3.12.6 WORDS – remember these

1. Fabless — designs chips, owns no factory — a company outsourcing all wafer manufacturing to a foundry.
2. Foundry — a factory that builds other people’s designs — a contract wafer manufacturer, pure-play if it sells no branded chips.
3. IDM — a company that designs and builds — integrated device manufacturer, the older model, as at Intel and Samsung.
4. Tape-out — sending the finished design to be made — release of final layout data to mask making.
5. RISC-V — an open, free instruction set — an open standard ISA governed by RISC-V International.

3.98 Common wrong ideas

1. Wrong: chips are made from ordinary beach sand. Right: they are made from selected lump quartz or quartzite above 99 percent silica, chosen for low iron content.
2. Wrong: silicon is mined as a metal. Right: silicon never occurs pure. It is always bound to oxygen and must be reduced out with carbon at about 1,900 degrees Celsius.

3. Wrong: 99 percent pure is good enough. Right: that is one foreign atom in a hundred. Chips need nine to eleven nines, roughly one per billion or better.
4. Wrong: a wafer is just a slice of purified silicon. Right: it must be one single crystal with no grain boundaries, grown from a seed, with specified orientation and resistivity.
5. Wrong: the mask is a stencil shaped like the circuit. Right: it is a 4x enlarged plate of computed shapes, distorted by optical proximity correction so the printed result is correct.
6. Wrong: EUV is just a brighter ultraviolet lamp. Right: it is 13.5 nanometre soft X-ray light made by vaporizing tin droplets, absorbed by air and glass, so it needs vacuum and mirrors.
7. Wrong: a 3 nanometre chip has 3 nanometre features. Right: the node name is a marketing label. Gate pitch on a 3 nanometre class process is around 45 nanometres.
8. Wrong: sand is cheap, so chips should be cheap. Right: the blank wafer is well under 1 percent of a processed leading-edge wafer. The value is in more than a thousand process steps.
9. Wrong: a faster and a slower processor from one family are different designs. Right: they are frequently the same die, binned by measured performance, sometimes with defective parts fused off.
10. Wrong: making the chip is hard and packaging is trivial. Right: since 2023, advanced packaging capacity rather than wafer capacity has limited artificial intelligence accelerators.

3.99 Chapter summary in 20 lines

1. Silicon is about 27.7 percent of the Earth's crust by mass, second only to oxygen, and never occurs pure.
2. It is locked in silica and silicate minerals, and the cleanest common source is the mineral quartz.
3. Chip feedstock is lump quartz or quartzite above 99 percent silica, not beach sand, for both purity and particle size.
4. A submerged-arc furnace at about 1,900 degrees Celsius reduces quartz with carbon, giving metallurgical-grade silicon at 98 to 99.5 percent.
5. That silicon becomes trichlorosilane, is distilled, and is deposited back onto hot filaments by the Siemens process at about 1,100 degrees Celsius.
6. The result is polysilicon at nine to eleven nines purity, roughly one foreign atom per billion or fewer.
7. Polysilicon is melted and regrown as one single crystal by the Czochralski method, using a seed, a thin neck, and slow rotating pulling.
8. A 300 millimetre boule is about 2 metres long and weighs about 265 kilograms, with a defined orientation and dopant level.
9. Float-zone growth gives purer, higher-resistivity silicon without a crucible, but is limited in diameter and used for power and detector devices.
10. The boule is cropped, ground, notched, wire sawn, lapped, etched and polished into 775 micrometre mirror wafers.
11. 300 millimetres is the industry limit. The 450 millimetre transition stalled when its consortium collapsed around 2016 and 2017.
12. Fabs run at ISO 14644-1 cleanliness with hundreds of air changes per hour, ultrapure water at 18.2 megaohm-centimetre, and vibration isolation.
13. A 2 nanometre class fab module costs roughly 25 to 35 billion US dollars, most of it tools rather than building.
14. Photolithography prints one 26 by 33 millimetre field at a time through a 4x reticle, then steps across the wafer and repeats.

15. Resolution follows the Rayleigh criterion, so shorter wavelength and higher numerical aperture give smaller features, at the cost of depth of focus.
16. Light went from 436 and 365 nanometre mercury lamps, to 248 and 193 nanometre lasers, to 193 nanometre immersion, to 13.5 nanometre EUV.
17. EUV comes from tin droplets struck twice by a carbon dioxide laser, 50,000 times a second, focused by molybdenum-silicon mirrors in vacuum, and only ASML builds the machine.
18. Doping, annealing, oxidation, CVD, PVD, ALD and 15 or more copper damascene metal layers add up to over 1,000 process steps and about three months in the fab.
19. Wafers are probe tested, binned, diced and packaged, and defect density punishes large dies so hard that the industry moved to chiplets, interposers and stacked memory.
20. Node names are marketing labels, transistors went planar to FinFET to gate-all-around, and the ability to build at the leading edge sits in very few hands.

The Transistor — The Switch That Built The World

4.0 What this chapter gives you

1. You will be able to say what a transistor is, in one sentence, without lying.
2. You will understand doping: how a few foreign atoms turn silicon from a poor conductor into a controllable one.
3. You will understand what a “hole” is, and why it behaves like a positive particle.
4. You will be able to explain a PN junction and a diode from first principles.
5. You will know the real history of the 1947 invention, including the argument about credit.
6. You will understand the MOSFET properly: gate, oxide, threshold, inversion, and its two operating regions.
7. You will understand CMOS, and be able to calculate the power a chip burns.
8. You will be able to state Moore’s Law correctly and explain why it was always an economic claim, not a promise about speed.

4.1 Before the transistor: relays and glowing glass

■ 4.1.1 PLAIN – in simple words

1. A computer needs one thing above all: a switch that another wire can flip.
2. The first such switch was the **relay**: an electromagnet that pulls a metal arm onto a contact.
3. The next switch was the **vacuum tube**: a glass bulb with the air pumped out and a hot wire inside.
4. Put a wire mesh in between, and a small voltage on that mesh throttles the flow. A switch with no moving parts.
5. Tubes are about a thousand times faster than relays. They are also hot, fragile, power-hungry, and they die.

■ 4.1.2 PLAIN – a picture in your head

1. A relay is a drawbridge with a motor. A small signal starts the motor, the bridge lowers, and heavy traffic crosses.
2. A vacuum tube is a hose with a valve. The water is already under pressure; your hand on the valve barely works at all.
3. The tube’s valve is a voltage on a mesh. Turning it costs almost nothing, but it controls a large current.
4. Where this comparison breaks: a drawbridge and a valve must physically move. In a tube only electrons move, which is why it is so much faster.

5. And no plumbing picture covers the heater. The tube burns power all the time just to keep electrons boiling off, whether or not it is doing anything.

■ 4.1.3 PLAIN – a worked example

1. Konrad Zuse built the Z3 in Berlin from telephone relays and presented it to an audience of scientists on 12 May 1941.
2. It used about 2,600 relays, roughly 1,400 of them for memory, and ran at about 5 to 10 cycles per second.
3. ENIAC, built from tubes at the University of Pennsylvania, was formally dedicated on 15 February 1946.
4. Its machine cycle was 200 microseconds and it did about 5,000 additions per second.

Machine	Year	Switch	Additions per second
Zuse Z3	1941	Relay	About 1
ENIAC	1946	Vacuum tube	About 5,000

5. The Z3 itself was destroyed on 21 December 1943 in an Allied bombing raid on Berlin.

■ 4.1.4 PLAIN – what is really happening inside

1. The **cathode** is metal heated by a filament to roughly 1,000 degrees Celsius. Hot metal throws electrons off its surface.
2. The **anode**, or plate, sits opposite at a positive voltage, often 100 to 300 volts, so electrons stream across the empty space. That is the current.
3. The **grid**, a fine mesh, sits in the path. A small negative voltage on it pushes electrons back, thinning the stream.
4. Because the grid is negative, almost no current flows into it. Large plate current, tiny grid current. That is amplification.
5. Now the costs. Every heater runs constantly, so ENIAC drew about 150 kilowatts, much of it just keeping filaments hot.
6. Filaments burn out, because that is physically what they are. And the machine cannot be used the instant you switch it on: cathodes need tens of seconds to reach temperature. That is warm-up time.

■ 4.1.5 TECHNICAL – the engineer's version

1. John Ambrose Fleming patented the thermionic diode in 1904. Lee de Forest added the control grid in 1906, producing the triode he called the Audion.
2. ENIAC used ordinary octal-base radio tubes: 6SN7 flip-flops in the decimal accumulators, with 6L7, 6SJ7, 6SA7 and 6AC7 types in logic roles.

ENIAC figure	Value
Vacuum tubes	About 17,468
Crystal diodes	7,200
Resistors	70,000
Capacitors	10,000
Soldered joints	About 5,000,000
Power draw	150 kW
Weight	About 30 short tons

ENIAC figure	Value

3. The tube count is quoted differently by different sources. 17,468 is the usual figure for the machine as built. Penn and several standard references round it to 18,000, and the count changed over its working life.
4. Early on, several tubes failed almost every day and ENIAC was unusable roughly half the time. High-reliability tubes arrived in 1948, after which failures dropped to about one every two days.
5. The longest failure-free run was 116 hours, close to five days, in 1954. ENIAC was switched off for the last time on 2 October 1955.
6. The honest version: failures were reduced partly by never switching the machine off. Thermal cycling, not steady running, kills filaments.
7. A 6SN7 heater draws 6.3 V at 0.6 A, about 3.8 W per tube. Multiplied by roughly 17,000 tubes that is about 65 kW of pure heater load, a large share of the 150 kW total.

■ 4.1.6 WORDS – remember these

1. Relay — an electromagnet that flips a metal switch — electromechanical single-pole element with millisecond operate time.
2. Vacuum tube — a glass bulb with a hot wire controlling electron flow — thermionic valve using emission into an evacuated envelope.
3. Triode — a tube with cathode, mesh and plate — three-electrode thermionic device with a control grid.
4. Thermionic emission — hot metal throwing off electrons — thermally activated escape over the material's work function barrier.
5. Warm-up time — the wait before a tube machine works — the time for indirectly heated cathodes to reach emission temperature, 10 to 60 seconds.

4.2 Doping, properly

■ 4.2.1 PLAIN – in simple words

1. Pure silicon is a poor conductor: not an insulator, not a metal, something annoying in between.
2. Each silicon atom has four outer electrons, and in a crystal every one is locked into a shared bond with a neighbour.
3. So we cheat. We add a very small number of foreign atoms. This is called **doping**.
4. Phosphorus and arsenic have five outer electrons. Four join the bonds; the fifth wanders off nearly free. Now there are spare electrons.
5. Silicon doped this way is **N-type**, N for negative, because the loose carriers are negative electrons.
6. Boron has three outer electrons. It fills three bonds and leaves the fourth with a gap in it.
7. That gap is a **hole**, and silicon doped this way is **P-type**.

■ 4.2.2 PLAIN – a picture in your head

1. Picture a cinema where every seat is taken. Nobody can move. That is pure silicon.
2. Make one person stand in the aisle. That is N-type: a free electron that can walk anywhere.
3. Now instead take one person out of a seat in the middle, leaving it empty. That is P-type.
4. Nobody is standing, but the empty seat moves: the person to its left slides over, so the gap travels across the room while each person shifts one place. The empty seat is the hole.
5. It is not a thing. It is an absence. But film only the gap and you would swear a real object was moving, in the opposite direction to the people.

- Also, holes are genuinely harder to push. In silicon a hole moves at roughly a third the speed of an electron under the same push. Hold on to that; it matters enormously later.

■ 4.2.3 PLAIN – a worked example

- Silicon has about 5 times 10 to the 22 atoms per cubic centimetre.
- Now dope it. A typical light doping is 1 times 10 to the 16 dopant atoms per cubic centimetre.
- Ratio: $1e16$ divided by $5e22$ is 2 times 10 to the minus 7, which is one dopant atom per five million silicon atoms.
- But free carriers rise from $1e10$ to $1e16$. A million times more.

```
Pure silicon: 5e22 atoms, ~1e10 free electrons -> poor conductor
Add 1e16 P:   5e22 atoms, ~1e16 free electrons -> useful N-type
```

```
Dopant to silicon ratio : 1 in 5,000,000
Change in free carriers : about 1,000,000 times
```

- One foreign atom in five million changes conductivity by a factor of about a million. The knob is absurdly sensitive.

■ 4.2.4 PLAIN – what is really happening inside

- A phosphorus atom takes a silicon lattice site. It is a similar size, so the crystal barely notices, and four of its five outer electrons pair with the four silicon neighbours.
- The fifth is only weakly bound, needing about 0.045 electron-volts to break free. Room temperature heat energy is about 0.026 electron-volts, and given how many attempts happen per second that frees essentially all of them.
- The freed electron leaves behind a phosphorus atom with one more proton than electron: a fixed positive ion.
- Boron is the mirror image. It fills three bonds and leaves one incomplete. An electron from a nearby bond hops in to complete it, so the gap moves, and boron becomes a fixed negative ion.
- So the picture is mobile carriers wandering, and charged ions frozen in place. Section 4.3 depends entirely on that distinction.
- The honest version: a hole is not a particle. It is the collective behaviour of a nearly full band of electrons, which physics lets us describe exactly as one positive particle with its own effective mass.

■ 4.2.5 TECHNICAL – the engineer's version

- Silicon is a group 14 element with an indirect bandgap of 1.12 eV at 300 K and an atomic density of 5.0 times 10 to the 22 per cubic centimetre.
- Intrinsic carrier concentration n_i at 300 K is about 1.0 times 10 to the 10 per cubic centimetre. Older texts quote $1.45e10$; the lower value is the modern accepted figure.
- The mass action law holds in equilibrium: n times p equals n_i squared. So N-type doping of $1e16$ gives n about $1e16$ and p about $1e4$ per cubic centimetre.
- Mobility at 300 K in lightly doped silicon: electrons about 1,400 cm squared per volt-second, holes about 450. A ratio near 3 to 1.

Region	Doping (per cm ³)	Role
Lightly doped body	$1e15$ to $1e16$	Substrate
Channel or well	$1e17$ to $1e18$	Sets threshold
Source and drain	$1e20$ to $1e21$	Low resistance

5. Measurement tools: a four-point probe gives sheet resistance in ohms per square; secondary ion mass spectrometry (SIMS) gives the dopant depth profile.
6. Note the convention: N-type and P-type name the majority carrier, not a net charge. Both materials are electrically neutral in bulk.

■ 4.2.6 WORDS – remember these

1. Doping — adding a tiny amount of another element — controlled introduction of substitutional impurities to set carrier concentration.
2. N-type — silicon with spare electrons — donor-doped semiconductor with electrons as majority carrier.
3. P-type — silicon with gaps in its bonds — acceptor-doped semiconductor with holes as majority carrier.
4. Hole — a missing electron that acts positive — an empty valence band state, treated as a quasi-particle of charge plus q .
5. Intrinsic carrier concentration — how conductive pure silicon is — n_i , about $1e10$ per cubic centimetre at 300 K.

4.3 The PN junction and the diode

■ 4.3.1 PLAIN – in simple words

1. Take N-type silicon, with spare electrons, and P-type silicon, with holes, and join them as one crystal with the doping changing partway through.
2. Electrons on the N side are crowded and the P side has almost none, so they spill across. Holes spill the other way.
3. That leaves a thin strip near the boundary with no free carriers at all, called the **depletion region**.
4. But the fixed ions remain, positive on the N side and negative on the P side, so a voltage now exists across that strip. Nobody applied it. It built itself, and it is a barrier to further spilling.
5. Push current one way and the barrier shrinks, so current flows easily. Push the other way and the barrier grows, so almost nothing flows.
6. A device that passes current one way and blocks the other is a **diode**.

■ 4.3.2 PLAIN – a picture in your head

1. Picture two adjoining rooms at a party with an open doorway. The left room is packed, the right nearly empty.
2. People spill through the doorway, because crowds spread out. That is diffusion. But every person who crosses must hand over their coat at the door, and the coats pile up until crossing stops.
3. The coat pile is the depletion region. It builds itself out of what already crossed, and it is what stops any more crossing.
4. A doorman who pushes people toward the doorway compresses the pile and crossing resumes. That is forward bias.
5. Where this comparison breaks: coats are a passive obstruction, but the depletion region is an electric field, acting instantly and at a distance.

■ 4.3.3 PLAIN – a worked example

1. Take a 1N4148 small-signal silicon diode. Connect its P side (the anode) to plus and its N side (the cathode) to minus. That is forward bias.

Forward voltage	Current	Change
0.60 V	1 mA	baseline
0.66 V	10 mA	10 times
0.72 V	100 mA	100 times
0.78 V	1 A	1000 times

2. Look at the pattern. Every extra 60 millivolts multiplies the current by ten. That is not a straight line, it is exponential.
3. Now reverse it. At minus 5 volts you might measure 10 nanoamps. At minus 50 volts, still nanoamps.
4. Forward current at 0.7 V is about 10 milliamps. Reverse current is about 10 nanoamps. A ratio of a million to one.

■ 4.3.4 PLAIN – what is really happening inside

1. At the moment of joining, electrons diffuse into the P side and holes diffuse into the N side.
2. Every electron that leaves the N side leaves a fixed positive donor ion behind, and every hole that leaves the P side leaves a fixed negative acceptor ion. Neither ion can follow.
3. So a wall of positive fixed charge builds on the N side of the boundary and a wall of negative fixed charge on the P side.
4. Separated charge makes an electric field that pushes electrons back toward the N side, exactly opposing the diffusion that created it. Balance is reached when the two cancel, and nothing net flows.
5. The voltage across the region is the **built-in potential**, typically 0.6 to 0.8 volts in silicon. You cannot measure it with a voltmeter: the two metal-to-silicon contacts produce equal and opposite offsets.
6. Apply an external voltage with the P side positive and the depletion region narrows, the barrier drops, and carriers flood across.
7. Apply the opposite and the region widens, the barrier grows, and majority carriers cannot get over it at all.
8. The honest version: current is not simply blocked one way. Diffusion and drift currents always flow in both directions. Diode current is the small difference between two large, nearly equal opposing flows.

■ 4.3.5 TECHNICAL – the engineer's version

1. The Shockley diode equation: $I = I_s \text{ times } (\exp(V \text{ divided by } (n \text{ times } V_T)) \text{ minus } 1)$.
2. V_T is the thermal voltage, kT divided by q , equal to 25.85 mV at 300 K. I_s is the reverse saturation current, typically $1\text{e-}12$ to $1\text{e-}15$ A for small silicon diodes, roughly doubling every 10 K. n is the ideality factor, between 1 and 2.
3. The decade-per-60-mV rule follows directly: $1 \text{ times } 25.85 \text{ mV times } 2.303$ equals 59.5 mV.
4. Built-in potential is $V_{bi} = V_T \text{ times } \ln(N_a \text{ times } N_d \text{ divided by } n_i \text{ squared})$. With N_a and N_d both $1\text{e}17$ and n_i equal to $1\text{e}10$, $V_{bi} = 0.02585 \text{ times } 32.2$, which is 0.83 V.

Diode type	Forward drop	Typical use
Silicon PN	0.6 to 0.7 V	General rectifier
Schottky	0.2 to 0.45 V	Fast switching
Germanium PN	0.25 to 0.3 V	Legacy detectors

Diode type	Forward drop	Typical use
Red LED	About 1.8 V	Indicator
Blue LED	2.8 to 3.4 V	Lighting, display

- Reverse breakdown has two mechanisms: Zener tunnelling below about 5 V with a negative temperature coefficient, and avalanche multiplication above about 6 V with a positive one. Near 5.6 V they cancel, which is why 5.6 V Zener diodes were once used as temperature-stable references.
- A Schottky diode is a metal-semiconductor junction with no stored minority charge, so it has essentially no reverse recovery. That is why it dominates high-frequency switching supplies.
- To observe the curve: a curve tracer or a source measure unit sweep. In simulation, a SPICE DC sweep with the .DC directive plots it from the model parameters IS, N, RS and BV.

■ 4.3.6 WORDS - remember these

- PN junction — where N-type meets P-type — a metallurgical junction with a self-forming space-charge region.
- Depletion region — the empty strip at the boundary — space-charge region containing only fixed ionized dopants.
- Built-in potential — the voltage that appears by itself — V_{bi} , set by doping levels and n_i , about 0.7 V in silicon.
- Forward bias — pushing the easy way — P side positive, barrier lowered, exponential injection current.
- Reverse bias — pushing the blocked way — N side positive, barrier raised, only saturation current flows.

4.4 The first transistor, and the argument about it

■ 4.4.1 PLAIN - in simple words

- Bell Telephone Laboratories had a business problem in the 1940s. Long distance calls needed amplifiers, amplifiers meant vacuum tubes, and tubes kept dying.
- On 16 December 1947, two of their people made one work.
- Their device was a small slab of very pure germanium with two gold contacts pressed onto its surface, extremely close together.
- A signal on one contact controlled a larger current through the other, amplifying by up to about a hundred times.
- On 23 December 1947 they demonstrated it to management. Their group leader, William Shockley, called it a magnificent Christmas present.
- Bell Labs kept it quiet for six months, then announced it at a New York press conference on 30 June 1948.

■ 4.4.2 PLAIN - a picture in your head

- Imagine controlling a river using only the shape of the riverbed just under the surface, without touching the water.
- That was the field-effect idea, tried since the 1920s. It kept failing and nobody knew why.
- Bardeen worked out why in 1947. The semiconductor surface was covered in trapped electrons acting like a shield.
- So Bardeen and Brattain stopped pushing from outside and went in through the surface directly, with two metal points almost touching.

- Where this comparison breaks: the point-contact device is not really a field-effect transistor at all, and exactly how it worked was argued about for years afterwards.

■ 4.4.3 PLAIN – a worked example

- Brattain wrapped gold foil around the point of a small plastic wedge.
- He cut the foil at the tip with a razor blade, leaving two gold edges about 50 micrometres apart. That is roughly half the width of a human hair.
- He pressed the wedge onto a slab of high-purity germanium with a spring. One gold edge was the emitter, the other the collector, and the germanium slab itself was the base.
- They measured a power gain. That is the test that matters; voltage gain alone can be had from a transformer.

Date	Event
16 Dec 1947	First working device
23 Dec 1947	Demonstrated to management
26 Feb 1948	Patent priority date
30 Jun 1948	Public announcement
Jan 1948	Junction transistor conceived
1951	Junction transistor working
1956	Nobel Prize in Physics

■ 4.4.4 PLAIN – what is really happening inside

- Shockley was the group leader who had pushed the field-effect programme for years, and it had failed repeatedly. The device that finally worked was built by two of his subordinates, using Bardeen's surface-state insight, in a direction Shockley had not chosen.
- Then Bell Labs lawyers examined the patent and found Shockley's own field-effect writing was uncomfortably close to patents Julius Edgar Lilienfeld had filed in the 1920s.
- To keep the patent safe they left Shockley's name off it. The application "Three-electrode circuit element utilizing semiconductive materials", with a priority date of 26 February 1948, named Bardeen and Brattain.
- Shockley was furious. Over the New Year he worked alone, largely in a Chicago hotel room, and in January 1948 conceived a better device.
- Instead of two metal points scratching a surface, he proposed a sandwich of three doped layers grown into one crystal: the **junction transistor**. His own application carries a priority date of 26 June 1948.
- It was more robust and manufacturable, and it made the point-contact device obsolete. Gordon Teal and Morgan Sparks at Bell Labs produced working grown-junction transistors in 1951. Bardeen left for the University of Illinois that year.
- In 1956 all three shared the Nobel Prize in Physics, "for their researches on semiconductors and their discovery of the transistor effect."
- Where experts disagree: some historians treat Shockley as the essential figure, since the junction transistor is the ancestor of everything since. Others treat him as a manager who claimed a result he did not produce. The safest reading is that invention and improvement both mattered, and came from different people.

■ 4.4.5 TECHNICAL – the engineer’s version

1. The 1947 device was a point-contact transistor on N-type germanium with two gold point contacts about 50 micrometres apart, amplifying the input by up to about 100 times.
2. Germanium, not silicon. Germanium has a 0.66 eV bandgap and much higher carrier mobility, and in 1947 it could be purified far better than silicon.
3. Gordon Teal at Texas Instruments announced the first commercial silicon transistor in 1954. Silicon’s wider 1.12 eV bandgap gives far better high-temperature behaviour.
4. The Regency TR-1, announced on 18 October 1954, was the first commercial transistor radio. It used four TI germanium transistors and sold for \$49.95.
5. Shockley left Bell Labs in 1955 and founded Shockley Semiconductor Laboratory in Mountain View, California, in 1956.
6. In 1957 eight of his staff resigned over his management and founded Fairchild Semiconductor. Fairchild’s descendants include Intel and AMD. That is the direct reason Silicon Valley is where it is.
7. John Bardeen won a second Nobel Prize in Physics in 1972, with Leon Cooper and John Robert Schrieffer, for the BCS theory of superconductivity. He remains the only person to have won the physics prize twice.

■ 4.4.6 WORDS – remember these

1. Point-contact transistor — two metal points on a germanium slab — the 1947 device, with emitter and collector point contacts on a base.
2. Surface states — trapped charge that shields the inside — states at a semiconductor surface that pin the Fermi level and screen applied fields.
3. Junction transistor — a sandwich of three doped layers — Shockley’s 1948 bipolar structure, ancestor of the BJT.
4. Power gain — output power larger than input power — the test that separates an amplifier from a transformer.
5. Germanium — the first material that worked — group 14 semiconductor, 0.66 eV bandgap, superseded by silicon for thermal reasons.

4.5 The bipolar junction transistor

■ 4.5.1 PLAIN – in simple words

1. Take three doped layers in a row: N, then P, then N. That is an **NPN** transistor. Reverse every layer and you have a **PNP**.
2. The base is deliberately made very thin and only lightly doped. That is the whole secret.
3. Feed a small current into the base and a much larger current flows from collector to emitter, typically 100 to 300 times larger.
4. There is a catch, and it is the catch that killed this device for computing.
5. Base current means power burned. Millions of these would burn power constantly, even doing nothing.

■ 4.5.2 PLAIN – a picture in your head

1. Picture a wide fast river with a narrow island in the middle.
2. Water hitting the island drains away down a small side channel. That side channel is the base current.
3. Because the island is so narrow, almost all the water sweeps past it and reaches the far bank. Maybe two hundred litres arrive for every one that drains.
4. But the side channel must keep draining. Block it and the whole flow stops.
5. And the small side flow is never optional. It is a permanent cost, and section 4.7 is about a device that removed it.

■ 4.5.3 PLAIN – a worked example

1. Take a 2N3904, one of the most common small NPN transistors ever made. Its current gain, beta, is roughly 100 to 300 at 10 milliamps.
2. Say we want to switch a 100 milliamp load. Use the worst-case beta of 100, never the typical value, so base current needed = 100 mA divided by 100 = 1 mA.
3. In practice you overdrive by 5 times to force it hard on, so use 5 mA.
4. With a 5 V control signal, the base-emitter junction drops about 0.7 V, so the resistor sees 4.3 V, and $R = 4.3 \text{ V} \text{ divided by } 0.005 \text{ A} = 860 \text{ ohms}$. Use the standard value 820 ohms.

```

5V ---[820 ohm]--- base
                    |
100 mA load ---> collector
                    |
                emitter --- ground

Base power   : 5 mA x 0.7 V   = 3.5 mW
Switch loss  : 100 mA x 0.2 V = 20 mW
Total burnt  : about 23.5 mW, continuously

```

5. Holding this one transistor on costs about 23.5 milliwatts forever, doing nothing.
6. A billion of them would burn 23.5 megawatts. That is a power station, and that number is why bipolar logic was never going to build a modern processor.

■ 4.5.4 PLAIN – what is really happening inside

1. In an NPN the emitter is heavily doped, the base thin and lightly doped, and the collector moderately doped and physically larger.
2. Forward bias lowers the emitter-base barrier, so electrons flood from the N emitter into the P base, where they are minority carriers and ought to recombine.
3. But the base is thinner than a micrometre, so most cross it before meeting a hole, and at the far edge the reverse-biased base-collector field sweeps them straight into the collector.
4. So perhaps 99.5 percent arrive at the collector and 0.5 percent recombine in the base and must be replaced by base current.
5. Beta is just the ratio of those two numbers. Lose 0.5 percent and beta is about 200. Gain comes from geometry and doping, not magic.
6. **Cut-off** is when base-emitter voltage is below about 0.6 V. Nothing is injected and the device is off.
7. The honest version: calling the BJT current-controlled is a convention, not a law. The physics is exponentially controlled by base-emitter voltage. But that exponential is so steep and so temperature-sensitive that designing with base current is far more reliable, so everyone does.

■ 4.5.5 TECHNICAL – the engineer’s version

1. The Ebers-Moll model describes the BJT. In forward active mode $I_C = I_S \times \exp(V_{BE} \text{ divided by } V_T)$.
2. Common-emitter gain beta equals I_C over I_B . Common-base gain alpha equals I_C over I_E . They relate as $\beta = \alpha \text{ divided by } (1 \text{ minus } \alpha)$.

Parameter	2N3904 typical	Meaning
h_{FE} (beta)	100 to 300	Current gain
$V_{CE(sat)}$	0.2 V	On-state drop
$V_{BE(on)}$	0.65 to 0.75 V	Turn-on voltage
$V_{CEO \text{ max}}$	40 V	Breakdown limit
f_T	About 300 MHz	Gain bandwidth

3. Bipolar logic families ran RTL, then DTL, then TTL. The 7400 series launched by Texas Instruments in 1964 and dominated digital design for twenty years.
4. Standard TTL dissipated roughly 10 mW per gate at rest; the 74LS low-power Schottky family reduced this to about 2 mW. Emitter-coupled logic was faster still at 25 mW or more per gate, and Cray supercomputers cooled it with liquid.
5. Bipolar devices still dominate analogue and radio work: low noise, high transconductance per unit current, excellent matching. They lost digital, not everything. Silicon-germanium heterojunction bipolar transistors reach f_T above 300 GHz.
6. SPICE models BJTs with the Gummel-Poon model, which extends Ebers-Moll to cover high injection and base-width modulation.

■ 4.5.6 WORDS – remember these

1. BJT — a sandwich transistor controlled by current — bipolar junction transistor, using both electrons and holes as carriers.
2. Emitter — the layer that supplies carriers — heavily doped terminal that injects minority carriers into the base.
3. Base — the thin middle layer that controls things — lightly doped region, thinner than the minority carrier diffusion length.
4. Collector — the layer that catches carriers — reverse-biased terminal that sweeps arriving minority carriers out.
5. Beta or h_{FE} — how many times the current is multiplied — common-emitter DC current gain, I_C over I_B , typically 20 to 500.

4.6 The MOSFET, the transistor inside your computer

■ 4.6.1 PLAIN – in simple words

1. Almost every transistor in the device you are reading this on is a MOSFET.
2. It has three parts you must know. The **source**, where carriers come from. The **drain**, where they go. The **gate**, which decides whether they go.
3. Because the gate is insulated, no current flows into it. It is a capacitor plate, not a wire into the device.
4. Put a positive voltage on the gate of an N-channel MOSFET and its electric field reaches through the insulator into the silicon below.

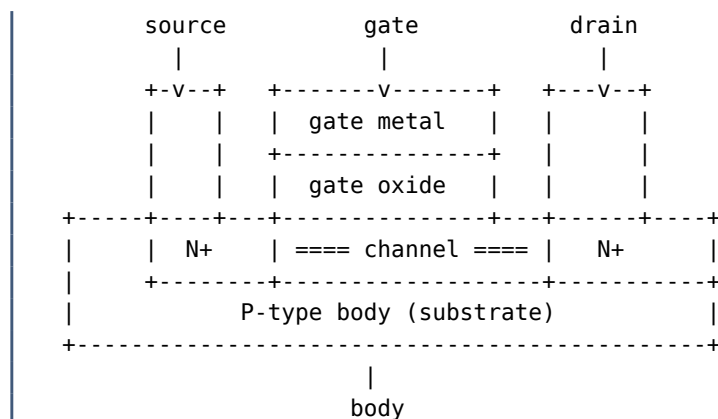
5. That field drags electrons up to the surface, forming a thin conducting layer bridging source to drain. That layer is the **channel**.
6. Here is the point that changes everything. Because the gate is insulated, holding the switch on costs no continuing current. You charge the gate once and it stays.

■ 4.6.2 PLAIN – a picture in your head

1. Imagine a dry canal between two full reservoirs. Nothing can cross.
2. Lower the plate and it does not push water. Instead it pulls groundwater up from below, filling the canal bed.
3. That is the gate, and that is why MOSFET logic can sit still without burning power.
4. Where this comparison breaks: lowering a real plate costs energy each time because of gravity, and so does switching a real gate, because its capacitance must be charged and discharged. Holding is free; changing is not.
5. Also, the insulator is not perfect. At modern thicknesses a few electrons tunnel straight through. The plate leaks, slightly.

■ 4.6.3 PLAIN – a worked example

1. Here is a planar N-channel MOSFET, sliced through the middle.



2. The channel is the dashed strip. It does not exist until the gate says so.
3. Now real numbers. Threshold voltage, the gate voltage at which the channel appears: 0.4 V. Supply: 1.0 V. Channel length L: 20 nanometres. Channel width W: 100 nanometres.
4. Below 0.4 V on the gate the device is off, passing only leakage.
5. At 1.0 V on the gate the overdrive is 1.0 minus 0.4, which is 0.6 V, and the device is fully on.
6. Double the width to 200 nanometres and on-current doubles, because a wider channel is a wider pipe.
7. Halve the length to 10 nanometres and on-current doubles too, because a shorter pipe has less resistance.
8. That is why engineers speak of the W over L ratio. It is the shape knob for strength.

■ 4.6.4 PLAIN – what is really happening inside

1. Start with the gate at zero volts, on an N-channel device built in P-type silicon. The body is full of holes and the two N+ islands are full of electrons.
2. Source to body is one PN junction and body to drain is another, facing the other way. Two back-to-back diodes cannot conduct either direction, so the device is firmly off.
3. Raise the gate a little. The field pushes holes downward, away from the surface, which is now depleted: no holes, no electrons. Still off, but the barrier is thinning.

4. Raise the gate more and the field starts pulling electrons up to the surface. Electrons in P-type silicon are minority carriers and very rare, but the field only needs a thin layer.
5. At some gate voltage the electron count at the surface equals the hole count in the bulk. That point is the **threshold voltage**.
6. Push past it and electrons outnumber holes at the surface. The surface has stopped behaving as P-type and started behaving as N-type.
7. That flip is called **inversion**, and it is the heart of the device. The material did not change. Its carrier population did.
8. Now source, channel and drain are all N-type. One continuous path. Current flows, and what happens next depends on the drain voltage.
9. If drain voltage is small, the channel is roughly even from end to end and the device behaves as a resistor whose value the gate sets. That is the **linear** or triode region.
10. Raise drain voltage and the channel is squeezed at the drain end, because the gate-to-channel voltage there is smaller. When its thickness there reaches zero, the device has reached **pinch-off**.
11. Past that, raising drain voltage barely raises current. The device behaves as a current source set by the gate. That is the **saturation** region.
12. Careful, this word is a trap. In a BJT, saturation means fully on with the lowest voltage drop. In a MOSFET it means pinched off at constant current. Same word, opposite feeling.
13. A **PMOS** device is the mirror image: P+ source and drain in an N-type well, turned on by a negative gate voltage relative to the source, which inverts the surface to P-type.
14. The honest version: “the channel disappears at pinch-off” is not quite true. A thin high-field region forms at the drain end and carriers are swept across it. Current does not stop, it stops increasing.

■ 4.6.5 TECHNICAL – the engineer’s version

1. Julius Edgar Lilienfeld filed the first field-effect patents in the mid 1920s and Oskar Heil filed a British patent in 1934. Neither produced a working device.
2. Shockley’s 1945 field-effect attempts failed. Bardeen’s 1947 surface state theory explained why: interface traps pinned the Fermi level and screened the applied field.
3. The solution was a clean thermally grown silicon dioxide interface. Mohamed Atalla and Dawon Kahng at Bell Labs built the first working MOSFET in 1959 and reported it in 1960.
4. Linear region, long-channel model: $I_D = \mu \text{ times } C_{ox} \text{ times } (W \text{ over } L) \text{ times } ((V_{GS} \text{ minus } V_{th}) \text{ times } V_{DS} \text{ minus } V_{DS}^2 \text{ over } 2)$, for V_{DS} below $V_{GS} \text{ minus } V_{th}$.
5. Saturation region, long-channel model: $I_D = \text{one half times } \mu \text{ times } C_{ox} \text{ times } (W \text{ over } L) \text{ times } (V_{GS} \text{ minus } V_{th})^2$, for V_{DS} at or above $V_{GS} \text{ minus } V_{th}$. C_{ox} is gate oxide capacitance per unit area, the oxide permittivity divided by oxide thickness t_{ox} .
6. The square law is a long-channel result. Modern short-channel devices are velocity saturated, so drain current is close to linear in $(V_{GS} \text{ minus } V_{th})$, not quadratic. Textbooks teaching only the square law are teaching a device that stopped being manufactured decades ago.
7. Below threshold the device is not off. Subthreshold current falls exponentially, at a rate called **subthreshold swing** S , in millivolts per decade.
8. S has a hard floor of $\ln(10) \text{ times } kT \text{ over } q$, which is 59.6 mV per decade at 300 K. Real planar devices reach 70 to 100; FinFETs reach about 65 to 70.
9. That floor is thermodynamic, not an engineering limit. It is the single biggest reason supply voltage stopped scaling, and therefore the reason for the power wall in section 4.10.
10. Intel reported reaching 1.2 nm of silicon dioxide, about five atomic layers, on its 65 nm process.

Below that, direct tunnelling leakage became intolerable.

Quantity	Typical modern value
Threshold voltage	0.2 to 0.5 V
Supply voltage	0.65 to 1.1 V
Gate oxide EOT	0.8 to 1.0 nm
Subthreshold swing	65 to 80 mV/decade
Gate leakage	Picoamps per device

■ 4.6.6 WORDS – remember these

1. MOSFET — a switch controlled by a voltage on an insulated plate — metal oxide semiconductor field-effect transistor, a unipolar device.
2. Source and drain — where carriers come from and go to — heavily doped regions of opposite type to the body.
3. Gate — the insulated control plate — electrode capacitively coupled to the channel, drawing only tunnelling leakage.
4. Gate oxide — the thin insulator under the gate — dielectric layer, silicon dioxide historically, hafnium-based high-k since 2007.
5. Threshold voltage — the gate voltage where it switches on — V_{th} , the point of strong inversion at the semiconductor surface.
6. Channel — the conducting bridge the gate creates — inversion layer of minority carriers at the oxide interface.
7. NMOS and PMOS — the electron version and the hole version — N-channel and P-channel devices, on with positive and negative gate overdrive.

4.7 CMOS: the pairing that made low power possible

■ 4.7.1 PLAIN – in simple words

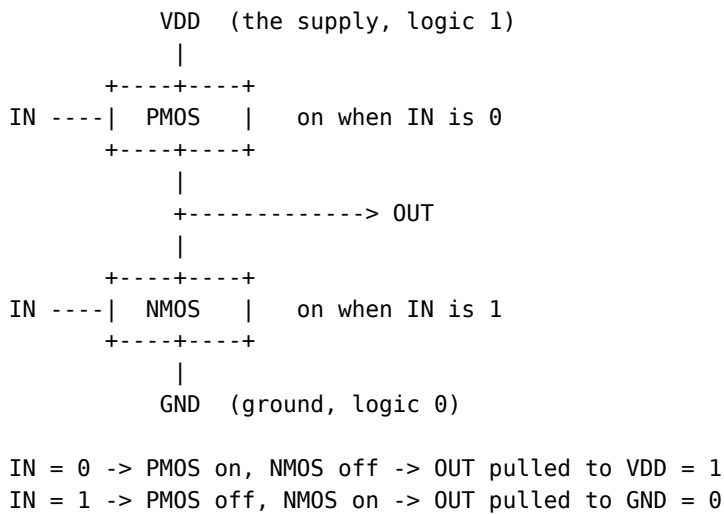
1. One MOSFET alone still wastes power. Pull a signal low through a resistor and current keeps flowing through that resistor.
2. Put a PMOS on top connected to the supply, and an NMOS below connected to ground. Join their gates and join their drains.
3. Feed in a 0. The PMOS turns on, the NMOS turns off, and the output is pulled up to the supply, a 1.
4. In both steady states exactly one transistor is on, so there is never a path from supply to ground. No path means no current.
5. **CMOS** stands for Complementary Metal Oxide Semiconductor. Complementary means the two types are used as a matched opposing pair.

■ 4.7.2 PLAIN – a picture in your head

1. Think of a sink with a tap above and a plug below, joined by one lever.
2. Push the lever one way and the tap opens while the plug closes, so the basin fills.
3. The mechanism guarantees tap and plug are never open together, so water never runs straight down the drain.
4. Where this comparison breaks: during the flick of the lever both really are slightly open for an instant. That overlap is called crowbar or short-circuit current, and it is a real cost in fast circuits.

■ 4.7.3 PLAIN – a worked example

1. Here is the CMOS inverter as a circuit.



2. Now calculate the power a real chip burns. The formula is $P = \alpha \times C \times V^2 \times f$.
3. C is the capacitance being charged, V the supply voltage, f the clock frequency, and α the activity factor, the fraction of nodes that change on a given tick.
4. Take a plausible processor core cluster: one billion switching nodes, 0.5 femtofarads each, 1.0 V supply, 3 GHz clock, activity factor 0.05.
5. Capacitance switched per cycle = 0.05 times $1e9$ times $0.5e-15$ farads = 2.5 times 10^{-8} farads, that is 25 nanofarads.
6. Energy per cycle = $C \times V^2$ = 25 nF times 1.0 squared = 25 nanojoules.
7. Power = 25 nanojoules times 3 billion cycles per second = 75 watts.
8. Now change one number. Raise the supply to 1.2 V and power scales with V^2 : 75 times 1.44 = 108 watts, for zero extra speed.

■ 4.7.4 PLAIN – what is really happening inside

1. When the input flips from 1 to 0, the PMOS turns on and connects the output wire to the supply.
2. That output wire, plus the gates of everything it drives, forms a capacitance that must be charged from 0 volts up to the supply voltage.
3. Charging a capacitor C to voltage V stores one half $C V^2$ and burns the same amount as heat in the transistor doing the charging. On the way back, the NMOS dumps the stored half to ground and that is lost too.
4. So a full 0 to 1 to 0 cycle costs $C \times V^2$ in total. That is where the formula comes from. Nothing at all is spent while the output sits still, which is the property bipolar logic could never offer.
5. Two other losses exist. First, crowbar loss: for the moment when the input is halfway, both transistors conduct and a spike passes through.
6. Second, and worse, leakage. Every off transistor passes a small current, and every gate leaks a tunnelling current through its oxide.
7. In the 1990s leakage was negligible. By the 90 nanometre generation, around 2003 to 2005, it had become a large share of total chip power.

■ 4.7.5 TECHNICAL – the engineer’s version

1. CMOS was invented by Frank Wanlass and Chih-Tang Sah at Fairchild in 1963. Their ISSCC paper described it as “nanowatt logic”.
2. US patent 3,356,858 was filed 18 June 1963 and issued 5 December 1967. RCA shipped the CD4000 logic family in 1968.
3. Dynamic power: $P_{\text{dyn}} = \alpha \times C_L \times V_{DD}^2 \times f$. Static power: $P_{\text{static}} = V_{DD} \times I_{\text{leak}}$, summed over every device. Short-circuit power is typically 5 to 15 percent of dynamic power.
4. Leakage components are subthreshold conduction, gate oxide tunnelling, junction band-to-band tunnelling, and gate-induced drain leakage (GIDL).
5. Subthreshold leakage rises exponentially as threshold voltage falls, at roughly one decade per S millivolts, with S about 70 mV per decade.

Era	Node	Supply	Leakage share
1990	800 nm	5.0 V	Under 1 percent
2000	180 nm	1.8 V	A few percent
2004	90 nm	1.2 V	20 to 40 percent
2010	32 nm HKMG	1.0 V	Reduced again
2024	3 nm GAA	0.7 V	Managed by design

6. Mitigations in current use: high-k metal gate from Intel’s 45 nm in 2007, multi-threshold cell libraries, power gating with sleep transistors, clock gating, dynamic voltage and frequency scaling, and FinFET or nanosheet geometry for better electrostatic control.
7. Why clocks stopped climbing: Intel’s Pentium 4 Prescott reached 3.8 GHz in November 2004 and stopped there.
8. In May 2004 Intel cancelled the Tejas and Jayhawk projects, which had aimed at much higher clock rates, and redirected to dual-core designs. The first dual-core desktop part, the Pentium D, shipped in May 2005.
9. The reason is arithmetic. Raising frequency requires raising voltage to keep timing closed, and power then rises with roughly the cube of frequency.
10. Two cores at 3 GHz do more total work than one core at 4.5 GHz for less power, provided the software can use them. That proviso is the entire subject of parallel programming.
11. Top desktop parts in 2025 boost to roughly 5.7 to 6.0 GHz, about 1.5 times the 2004 peak across twenty years. In the twenty years before 2004, clock rates rose more than a thousand times.

```
# read package energy counter, microjoules, Intel RAPL
cat /sys/class/powercap/intel-rapl:0/energy_uj

# per-core frequency, temperature and package watts
sudo turbostat --interval 1
```

■ 4.7.6 WORDS – remember these

1. CMOS — using an N and a P transistor as an opposing pair — complementary MOS logic with no static path from supply to ground.
2. Inverter — the gate that flips 0 to 1 — series PMOS pull-up and NMOS pull-down sharing gate and drain nodes.
3. Dynamic power — the cost of changing — $\alpha C V^2 f$, paid per transition.
4. Static power — the cost of merely existing — V_{DD} times total leakage current, paid continuously.
5. Activity factor — how often a wire actually changes — α , typically 0.01 to 0.2 in real designs.

4.8 The same device, two jobs

■ 4.8.1 PLAIN – in simple words

1. A transistor does not know whether it is in a computer or a radio. It is the same device doing the same thing.
2. Digital circuits use only the two extremes: fully off and fully on.
3. In between is a region where a small input change makes a large output change. Analogue circuits live there.
4. Digital design crosses that middle region as fast as possible, because it wastes power and its output is undefined.
5. Analogue design treats the same region as the whole point, because that is where gain comes from.

■ 4.8.2 PLAIN – a picture in your head

1. Think of a light switch versus a dimmer knob.
2. A light switch has two positions and you flip past the middle quickly. That is digital.
3. A dimmer sits deliberately in the middle, where a small turn makes a noticeable change. That is analogue.
4. Where this comparison breaks: a mechanical dimmer wastes heat in proportion to how much it dims, and so does a transistor in its middle region, because it is dropping voltage while passing current.
5. That is why a class A audio amplifier is warm with no music playing, and why digital chips never sit in the middle if they can help it.

■ 4.8.3 PLAIN – a worked example

1. The **transfer curve** plots output voltage against input voltage for one CMOS inverter. Read it left to right.

Input	Output	Region
0.0 V	1.0 V	Flat, logic 1 out
0.40 V	0.95 V	Starting to move
0.50 V	0.50 V	The cliff, high gain
0.60 V	0.05 V	Nearly done
1.0 V	0.0 V	Flat, logic 0 out

2. At 0 V in, the NMOS is off and the PMOS fully on, so the output sits at the full supply and the curve is flat. At about 0.5 V, the switching point, it plunges almost vertically to near 0 V, then flattens again.
3. So the curve is flat, then a cliff, then flat again.
4. Digital design wants the two flat ends. Flat means a sloppy input still gives a clean output. That property is noise margin, and it is why digital signals can be copied endlessly without decay.
5. Analogue design wants the cliff. Its slope is the voltage gain, often 20 to 50 for a simple CMOS inverter.
6. Bias an inverter at its switching point with a feedback resistor and you have an amplifier, not a logic gate. This is a real technique, used in crystal oscillator circuits.

■ 4.8.4 PLAIN – what is really happening inside

1. In the flat regions one transistor is in its linear region acting as a small resistance and the other is fully cut off.
2. Almost the full supply appears across the off device and almost none across the on device, so current is essentially zero and so is power.
3. In the cliff region both devices are in saturation, acting as current sources. Two opposing current sources fight, and the output swings wildly for a small input nudge.
4. That is where gain comes from. It is not amplification of energy. The energy comes from the supply, and the transistor is a valve deciding how much of it reaches the output.
5. In the cliff region there is a real path from supply to ground, so current flows and heat is produced. Digital circuits cross that region in tens of picoseconds; analogue circuits sit in it and pay for it in heat.
6. The honest version: a transistor never amplifies power in the sense of creating it. It modulates a large power flow using a small one. Gain is a ratio of signals, never a violation of energy conservation.

■ 4.8.5 TECHNICAL – the engineer's version

1. Small-signal voltage gain of a CMOS inverter at its switching threshold is minus (g_{m_n} plus g_{m_p}) times (r_{o_n} in parallel with r_{o_p}).
2. Intrinsic gain g_m times r_o for a single modern short-channel device is only about 10 to 30, and it has fallen as nodes shrank. That is why analogue design got harder, not easier, with scaling.
3. Noise margins: $NMH = V_{OH} \text{ minus } V_{IH}$, and $NML = V_{IL} \text{ minus } V_{OL}$. For standard CMOS at 5 V, V_{IH} is 3.5 V, V_{IL} is 1.5 V, V_{OH} is 4.9 V and V_{OL} is 0.1 V, giving margins near 1.4 V each way.
4. Radio front ends still commonly use bipolar or silicon-germanium devices for low-noise amplifiers, because their transconductance per unit current and their noise behaviour beat CMOS at the same power.
5. Where experts disagree: whether analogue functions should share the same advanced node as the digital logic. Integration saves packaging and interconnect; separate older nodes give better analogue devices and lower cost. Chiplet packaging has shifted this argument, not settled it.

■ 4.8.6 WORDS – remember these

1. Transfer curve — the plot of output against input — DC voltage transfer characteristic, flat at both ends with a steep transition.
2. Noise margin — how much interference a signal can take — the voltage gap between guaranteed output levels and required input levels.
3. Gain — how much bigger the output change is — the slope of the transfer curve at the operating point.
4. Bias — where you park the device on the curve — the DC operating point set before any signal is applied.
5. Class A — an amplifier that conducts all the time — conduction over the full 360 degrees of the input cycle, at most 50 percent efficient.

4.9 The changing shape of the transistor

■ 4.9.1 PLAIN – in simple words

1. For about forty years transistors were flat. The channel lay along the top surface of the wafer with the gate above it. That is a **planar** transistor.

2. As transistors shrank, the flat design began to fail in a specific way. The gate could only touch the channel from one side, the top, while the drain, sitting right next to a very short channel, started to influence it too.
3. So the gate lost the argument. It could no longer switch the channel fully off, and leakage rose sharply.
4. The fix was to change the shape: stand the channel up as a thin vertical fin and wrap the gate around three of its sides. That is a **FinFET**.
5. The next step wraps the gate around all four sides, making the channel a stack of thin horizontal ribbons with gate material threaded between them.
6. That is **gate-all-around**, also called nanosheet, and it is the current leading-edge shape.

■ 4.9.2 PLAIN – a picture in your head

1. Imagine stopping water in a pipe by pressing with one hand from above.
2. Now use both hands and a thumb underneath. Three points of grip, much better control.
3. Now imagine a ring clamp closing all the way around. Total control from every direction.
4. Where this comparison breaks: the gate never squeezes anything. It only produces an electric field. But “surrounding it more completely gives more control” is exactly right.

■ 4.9.3 PLAIN – a worked example

1. Here is the timeline of the shape change, with companies and years.

Shape	First volume use	Who
Planar bulk	1960s to 2011	Everyone
SOI planar	From 1998	IBM, AMD
FinFET (Tri-Gate)	2012	Intel 22 nm
FinFET	2015	TSMC, Samsung
GAA nanosheet	2022	Samsung 3 nm
GAA nanosheet	2025	TSMC N2, Intel 18A

2. Intel announced its 22 nanometre Tri-Gate process on 4 May 2011 and shipped Ivy Bridge processors using it in April 2012. That was the first high-volume FinFET product.
3. Samsung began production with 3 nanometre gate-all-around, marketed as MBCFET, in June 2022.
4. TSMC’s N2 process entered volume production in the fourth quarter of 2025. It is TSMC’s first gate-all-around node.

■ 4.9.4 PLAIN – what is really happening inside

1. The problem the shape solves has a name: **short-channel effects**.
2. When the channel is long, the gate alone sets the barrier between source and drain. When it is very short, the drain’s own depletion region reaches close to the source.
3. Now raising the drain voltage lowers the source barrier by itself, with no help from the gate. That is drain-induced barrier lowering, or DIBL.
4. Wrapping the gate around more of the channel restores its authority, because the gate’s field now dominates from several directions at once.
5. The results are all bad: threshold voltage drops as the channel shortens, the subthreshold slope degrades so turn-off is less sharp, and off current rises.
6. Silicon-on-insulator, or SOI, attacks a related problem differently. A buried layer of oxide sits under the whole device, cutting off deep leakage paths through the substrate, reducing junction capacitance, and removing the latch-up failure mode entirely.

7. The gate insulator changed too, for a hard physical reason. Silicon dioxide was thinned generation after generation to keep the gate's grip strong, and Intel reached 1.2 nanometres of it, about five atomic layers, at 65 nanometres.
8. Five atoms is not a film any more. Electrons tunnel straight through it and leakage explodes.
9. The escape was a material with a higher dielectric constant, called **high-k**, which can be physically thicker while giving the same electrical coupling. Thicker means far less tunnelling.
10. Hafnium-based oxides have a dielectric constant around 20 to 25 against 3.9 for silicon dioxide, which buys back several atomic layers of thickness.
11. High-k films do not work with polysilicon gates, so metal gates had to return at the same time. Hence the paired name, high-k metal gate.

■ 4.9.5 TECHNICAL – the engineer's version

1. Intel introduced high-k metal gate at 45 nanometres in 2007 with the Penryn family. Gordon Moore called it the biggest change to transistor technology since the polysilicon gate in the late 1960s.
2. Intel's published figures for that change: gate oxide leakage reduced more than 10 times, source-to-drain leakage more than 5 times, and drive current improved by more than 20 percent.
3. Equivalent oxide thickness (EOT) is the standard figure of merit: the thickness of silicon dioxide that would give the same capacitance per area. Modern EOT is around 0.8 to 1.0 nanometres, at a physical thickness of 2 to 3 nanometres.
4. FinFET geometry parameters are fin height, fin width (typically 5 to 8 nm) and fin pitch (roughly 24 to 30 nm at recent nodes).
5. FinFET width is quantized. You cannot ask for 1.5 fins, so device strength comes in integer multiples of one fin. That constrains circuit design in a way planar devices never did. Nanosheet removes the quantization, because sheet width is drawn continuously.
6. FD-SOI, fully depleted silicon-on-insulator, is a live alternative at 22 nanometres and below, offered by GlobalFoundries as 22FDX. It allows back-bias tuning of threshold voltage at runtime, which FinFET cannot do.
7. Where experts disagree: whether FD-SOI or FinFET suits low-power and radio products at moderate nodes. FinFET won the leading edge outright; FD-SOI retains real advocates for internet-of-things and automotive parts.
8. Beyond nanosheet, the announced direction is CFET, stacking an NMOS device directly on a PMOS device. As of 2026 that is active research and pathfinding, not production.

■ 4.9.6 WORDS – remember these

1. Planar transistor — the old flat design — channel in the wafer surface with a gate above it on one side only.
2. FinFET — a channel on edge with the gate around three sides — tri-gate device with width quantized in integer fins.
3. Gate-all-around — gate wrapped completely around the channel — nanosheet or nanowire device, marketed as MBCFET by Samsung and RibbonFET by Intel.
4. Short-channel effects — the gate losing control as things shrink — DIBL, threshold roll-off, punch-through, subthreshold slope degradation.
5. SOI — a buried insulating layer under the devices — silicon-on-insulator, cutting substrate leakage, junction capacitance and latch-up.

4.10 Scale, Moore's Law, and the power wall

■ 4.10.1 PLAIN – in simple words

1. A silicon atom is about 0.22 nanometres across, so a gate insulator one nanometre thick is four or five atoms.
2. In 1971 the Intel 4004 held 2,250 transistors. In 2024 the Apple M4 held about 28 billion. That is a factor of over twelve million in fifty-three years.
3. **Moore's Law** is the name for that trend. It is not a law of physics. It is an observation about industry that turned into a plan.
4. It is also not about speed, despite what almost everyone says.
5. It has been slowing since around 2015. Doubling now takes closer to three years than two, and each step costs far more than the last.

■ 4.10.2 PLAIN – a picture in your head

1. Imagine a printing press that gets twice as good every two years, so a page holds twice as many words. For decades each new press also printed faster and used no more electricity. Better in every direction, free.
2. Then something changed. The presses kept holding more words but stopped getting faster, and started running hot.
3. That is exactly what happened to processors around 2005. Cores multiplied because clock speed could not rise.
4. Where this comparison breaks: printing more pages in parallel always helps, but computing in parallel only helps if the task can be split. Many cannot. That limit is called Amdahl's Law and it is not going away.

■ 4.10.3 PLAIN – a worked example

1. Here is the transistor count table, with real parts and real years.

Chip	Year	Transistors
Intel 4004	1971	2,250
Intel 8086	1978	29,000
Intel 80386	1985	275,000
Intel Pentium	1993	3.1 million
Pentium 4	2000	42 million
Core 2 Duo	2006	291 million
Apple A7	2013	About 1 billion
Apple M1	2020	16 billion
Apple M4	2024	28 billion
Nvidia GB200	2024	208 billion
Cerebras WSE-3	2024	4 trillion

2. Now count switching events. Take the Apple M4: 28 billion transistors, a 4 GHz clock, and an activity factor of 5 percent.
3. $28 \times 10^9 \text{ times } 4 \times 10^9 = 1.12 \text{ times } 10^{20}$. Times 0.05 gives 5.6 times 10^{18} switching events every second.
4. That is 5.6 billion billion switches per second, in a chip you can cover with a thumb, drawing a few tens of watts.

5. The industry as a whole made about 250 times 10 to the 18 transistors in 2014, according to an IEEE Spectrum estimate. That is roughly 8 trillion transistors manufactured every second of that year.

■ 4.10.4 PLAIN – what is really happening inside

1. Gordon Moore, then at Fairchild, wrote an article for Electronics magazine published on 19 April 1965, titled “Cramming more components onto integrated circuits”.
2. His actual observation: the complexity for minimum component costs had increased at a rate of roughly a factor of two per year, and he expected that to continue for at least ten more years.
3. Read that carefully. “Complexity for minimum component costs” means the number of components on the chip that is cheapest per component.
4. In 1975, at the IEEE International Electron Devices Meeting, Moore revised it: doubling yearly until about 1980, then slowing to about every two years.
5. The “every 18 months” version everyone quotes is not Moore’s. David House, an Intel executive, said in 1975 that more transistors plus faster ones would double performance about every 18 months.
6. Now the second law, the one that actually stopped. Robert Dennard and colleagues at IBM published a 1974 paper on designing ion-implanted MOSFETs with very small physical dimensions.
7. Their scaling rules: shrink every dimension by a factor k , shrink the supply voltage by k , and raise the doping by k .
8. The consequences are beautiful. Area falls by k squared, delay falls by k , power per device falls by k squared, and power per unit area stays constant.
9. Constant power density is the magic line. It means you can double the transistor count, run them faster, and the chip does not get hotter.
10. For roughly thirty years that held, and engineers came to treat it as normal. It was not normal. It was a gift.
11. It ended because supply voltage could not keep falling. Supply voltage must stay well above threshold voltage, and threshold voltage cannot fall without subthreshold leakage rising exponentially.
12. Section 4.6 gave the reason: the 59.6 millivolt per decade thermal floor. That is physics, at any temperature above absolute zero. So voltage stalled near 1 volt around 2005, having fallen from 5 volts.
13. Transistors kept shrinking, but the V squared term stopped shrinking.
14. Power density therefore began to climb. That is the **power wall**.
15. The industry’s answer was to stop raising clock speed, add cores, and leave large parts of the chip powered down at any moment. That last practice has a name: dark silicon.

■ 4.10.5 TECHNICAL – the engineer’s version

1. Dennard constant-field scaling, with factor k greater than 1: dimensions divided by k , voltage divided by k , doping times k , current divided by k , capacitance divided by k , delay divided by k , power per device divided by k squared, power density unchanged.
2. Post-Dennard reality since roughly 2005: dimensions still divided by k , but voltage nearly constant, so power density rises by roughly k squared.
3. Node names are marketing, not measurements. This is a convention, not a standard. No physical feature on a “3 nm” process measures 3 nanometres.
4. The meaningful density metrics are contacted poly pitch (CPP), metal 1 pitch and cell height in track count. Leading nodes in 2025 sit around 45 to 50 nm CPP and 20 to 24 nm minimum metal pitch. Transistor density is quoted around 200 to 300 million per square millimetre.

Cause of slowdown	Effect
Atomic dimensions	Variability, few dopants
EUV tool and mask cost	Cost per wafer rises
Fab capital cost	Above 20 billion dollars
Design and verification	NRE cost per chip rises
Interconnect resistance	Wires now limit delay

- Interconnect is the underrated one. Copper resistivity rises sharply as wire cross-sections approach the electron mean free path, so wire delay has scaled far worse than transistor delay.
- Where experts disagree, sharply: whether cost per transistor still falls at the leading edge. Foundries state that it does. Several analysts argue it has been roughly flat since the 28 nanometre node. Nvidia's chief executive said publicly in 2022 that Moore's Law is dead. The honest answer is that it depends on volume, yield and which costs you count.
- Established fact: transistor counts per chip are still rising. Active research: CFET stacking, two-dimensional channel materials such as molybdenum disulphide, and backside power delivery. Marketing claim: any specific node name, and any single number offered as "the density".
- Tools to observe scale in practice: `lscpu` and `/proc/cpuinfo` on Linux for core and cache topology, and `hwloc -ls` for the physical layout.

```
# core count, cache sizes and topology
lscpu

# hierarchical view of packages, cores and caches
hwloc -ls
```

■ 4.10.6 WORDS – remember these

- Moore's Law — chips roughly double in transistor count every two years — an economic observation about components at minimum cost per component.
- Dennard scaling — shrinking used to make chips faster and no hotter — constant-field scaling giving constant power density per unit area.
- Power wall — the point where you cannot cool it any faster — the limit reached when power density stopped being constant under scaling.
- Dark silicon — parts of the chip that must stay off — the fraction of a die that cannot be powered at once within the thermal budget.
- Process node — the name of a manufacturing generation — a marketing label since roughly 2011, not a measurement of any feature.

4.11 What a transistor costs

■ 4.11.1 PLAIN – in simple words

- The most important number about the transistor is not its size or its speed. It is its price.
- In the mid 1950s one transistor cost a few dollars. You counted them individually and designed circuits to use as few as possible.
- Today a single dollar buys hundreds of millions of them.
- That is a fall of roughly a billion times, and it is the reason every other chapter of this book is possible.
- Caches, branch predictors, graphics units, encryption engines and neural accelerators all exist because transistors became too cheap to ration.

■ 4.11.2 PLAIN – a picture in your head

1. Imagine paper cost as much as gold. You would write only what mattered, on both sides, in tiny letters.
2. Now imagine paper becomes free. You print drafts, take notes you never read, and keep three copies of everything.
3. Where this comparison breaks: transistors are not free, only individually cheap. A leading-edge chip design still costs hundreds of millions of dollars to bring to market.
4. The cost simply moved. It moved off the individual component and onto the design, the masks and the factory.

■ 4.11.3 PLAIN – a worked example

1. Here are real price points across seventy years, as quoted at the time and not adjusted for inflation.

Year	Price per transistor	Note
1954	A few dollars	Early TI devices
1960	About 1 dollar	Small-quantity discrete
1964	About 4 dollars	Noyce, low volume
1971	Under 1 cent	Intel 4004 era
2024	A few billionths	Leading-edge logic

2. Work the modern number. Take a chip with 28 billion transistors whose finished die costs, say, 80 dollars to make.
3. 80 dollars divided by 28 billion is about 2.9 times 10 to the minus 9 dollars per transistor, roughly three billionths of a dollar.
4. Compared with about 1 dollar in 1960, that is a ratio of about 350 million to one.
5. In 1954 the Regency TR-1 pocket radio, announced on 18 October 1954, sold for \$49.95 and contained four transistors.

■ 4.11.4 PLAIN – what is really happening inside

1. The price collapse comes from one structural fact: cost scales with wafer area, not with transistor count.
2. Processing a wafer costs roughly the same whether the features on it are large or small. The steps are the same steps.
3. So halving the linear dimension fits four times as many transistors on the same wafer for nearly the same processing cost. That is the true engine of Moore's Law, and why Moore framed his observation in terms of cost per component.
4. Yield modifies this. A wafer has a certain number of defects, and a bigger die catches more of them, so bigger dies yield worse and not linearly. That is why the industry moved to chiplets: several small high-yielding dies packaged together.
5. The relationship has weakened in the last decade, because leading-edge wafers now cost far more, thanks to extreme ultraviolet lithography tools and masks.
6. So density still improves, but cost per transistor improves much less than it used to, and at some nodes it may not improve at all.
7. The honest version: "transistors keep getting cheaper" was true without qualification until roughly 2012. Since then it depends on the node, the volume and who is doing the accounting.

■ 4.11.5 TECHNICAL – the engineer’s version

1. Die cost is approximately wafer cost divided by (gross die per wafer times yield).
2. Yield follows a negative binomial model: $Y = (1 + (D0 \text{ times } A) \text{ divided by } \alpha)^{\alpha}$ raised to the power minus alpha, where D0 is defect density per square centimetre, A is die area and alpha is a clustering factor.
3. Worked figures: at D0 of 0.1 defects per square centimetre and alpha of 3, a 100 square millimetre die yields about 90 percent, while a 600 square millimetre die yields about 42 percent. That yield curve, not physics, is why reticle-limit dies cost what they do.

Cost item	Rough scale, 2025
Leading-edge 300 mm wafer	18,000 to 30,000 USD
EUV mask set	Tens of millions USD
Leading-node design NRE	300 to 700 million USD
New leading-edge fab	20 billion USD and up

4. These figures are approximate and vary by customer, volume and contract. Foundry pricing is confidential, so all public numbers are estimates.
5. Non-recurring engineering cost is now the dominant barrier to entry. That is why only a handful of companies design at the leading node and everything else stays on mature 28 nanometre and 16 nanometre processes.
6. The consequence for a software engineer: compute is cheap and engineer-hours are expensive, so the industry trades transistors for productivity. That trade is why abstraction layers keep multiplying.
7. Established fact: cost per transistor fell by orders of magnitude from 1960 to about 2012. Active debate: whether it still falls at the leading edge. Marketing claim: any single cost-per-transistor number offered without stating node, volume and yield assumptions.

■ 4.11.6 WORDS – remember these

1. Cost per transistor — what one switch costs you — die cost divided by transistor count, the metric Moore’s 1965 paper was actually about.
2. Yield — the fraction of chips that work — the proportion of die on a wafer passing test, modelled against defect density and die area.
3. Defect density — how dirty the process is — D0, defects per square centimetre, around 0.1 or below for a mature leading process.
4. NRE — the one-off cost before you make any chips — non-recurring engineering cost, covering design, verification and mask sets.
5. Chiplet — several small dies packaged as one — a partitioning strategy trading packaging complexity for much better yield.

4.98 Common wrong ideas

1. Wrong: a transistor is a tiny mechanical switch. Right: nothing moves. A voltage or current changes how many charge carriers are available in a region, and that changes its conductivity.
2. Wrong: N-type silicon is negatively charged and P-type is positively charged. Right: both are electrically neutral. The letters name the majority carrier, not a net charge.
3. Wrong: a hole is a real positive particle. Right: it is a missing electron in a bond. The physics lets us treat it exactly as a positive particle, which is why the model is used, but nothing positive is moving.

4. Wrong: the gate of a MOSFET draws current to control the channel. Right: the gate is insulated and draws only a tiny tunnelling leakage. That is the whole reason CMOS logic can idle cheaply.
5. Wrong: saturation means the same thing in a BJT and a MOSFET. Right: BJT saturation means fully on with a low voltage drop. MOSFET saturation means pinched off with a nearly constant current.
6. Wrong: CMOS uses no power when idle. Right: it uses very little dynamic power when idle, but leakage is paid continuously, and since about the 90 nanometre generation it has been a major share of total power.
7. Wrong: Moore's Law says computers get twice as fast every 18 months. Right: Moore's 1965 statement was about components per chip at minimum cost, revised in 1975 to a two-year doubling. The 18-month performance version came from David House at Intel.
8. Wrong: clock speeds stopped rising because we ran out of transistor speed. Right: they stopped because power density stopped being constant when Dennard scaling ended, so faster clocks needed more voltage and produced unmanageable heat.
9. Wrong: a "3 nanometre" process has 3 nanometre transistors. Right: node names have been marketing labels since roughly 2011. No feature measures the node name. Use contacted poly pitch if you want a real number.
10. Wrong: Shockley invented the transistor. Right: Bardeen and Brattain built the first working device in December 1947 and are named on its patent. Shockley invented the later junction transistor. All three shared the 1956 Nobel Prize.

4.99 Chapter summary in 20 lines

1. A computer needs an electrically controlled switch, and the transistor is the best one ever found.
2. Relays worked but had to physically move, so they were slow. Zuse's Z3 of 1941 used about 2,600 of them at a few hertz.
3. Vacuum tubes were about a thousand times faster, but needed hot filaments, burned enormous power and failed constantly. ENIAC used about 17,468 of them and drew 150 kilowatts.
4. Pure silicon barely conducts, because every outer electron is locked into a bond.
5. Adding phosphorus or arsenic, with five outer electrons, gives spare electrons and makes N-type silicon.
6. Adding boron, with three outer electrons, leaves gaps called holes and makes P-type silicon. A hole is an absence that moves and behaves as a positive carrier.
7. One dopant atom per five million silicon atoms changes conductivity about a million-fold. That sensitivity is why purity matters so much.
8. Joining N and P material makes a depletion region and a built-in voltage of about 0.7 volts. That is a diode: easy one way, blocked the other.
9. Bardeen and Brattain made the first working transistor on 16 December 1947 at Bell Labs, announced on 30 June 1948, using germanium and two gold contacts.
10. Shockley, left off that patent, conceived the junction transistor in January 1948. It was working by 1951. All three shared the 1956 Nobel Prize.
11. The bipolar junction transistor uses a thin base to let a small base current control a collector current 100 to 300 times larger.
12. It burns power continuously, because base current can never stop while the device is on. That cost ruled it out for large digital chips.
13. The MOSFET, demonstrated by Atalla and Kahng at Bell Labs in 1959 and 1960, controls a channel through an insulator using an electric field alone.

14. Its gate draws essentially no current, so holding it on is free. Above the threshold voltage the surface inverts and a channel forms.
15. CMOS, invented by Wanlass and Sah at Fairchild in 1963, pairs an NMOS with a PMOS so exactly one is on in each steady state, leaving no path from supply to ground.
16. Dynamic power is $\alpha C V^2 f$. A billion nodes at 0.5 femtofarads, 1 volt, 3 gigahertz and 5 percent activity gives about 75 watts.
17. The same device is a switch at the flat ends of its transfer curve and an amplifier on the steep cliff between them.
18. Shapes changed because the gate lost control of short channels: planar until 2011, FinFET from Intel's 22 nanometre process in 2012, and gate-all-around nanosheet from Samsung in 2022 and TSMC and Intel in 2025.
19. Moore's 1965 observation was about components per chip at minimum cost, revised to two years in 1975. Dennard scaling ended around 2005, power density began rising, and cores multiplied instead of clock speeds.
20. The real story is price: from roughly one dollar per transistor in 1960 to a few billionths of a dollar today. Everything else in this book rests on that.

Logic Gates and Arithmetic — How Switches Learn to Add

5.0 What this chapter gives you

1. You will be able to explain how a rule about true and false became a rule about electricity, and who made that jump and when.
2. You will be able to draw a CMOS inverter, a NAND gate and a NOR gate from real transistors, and count how many transistors each one needs.
3. You will know all seven basic gates by truth table and by plain meaning, and be able to say what each one is for.
4. You will be able to build every gate out of NAND gates only, and say why chip factories care about that.
5. You will be able to take a written specification, turn it into a truth table, and shrink it with a Karnaugh map.
6. You will be able to build a half adder, a full adder and an 8-bit adder, and trace a carry moving through it bit by bit.
7. You will be able to explain how the same adder does subtraction, and how the hardware notices that a result went wrong.
8. You will be able to describe a multiplexer, decoder, comparator, barrel shifter and priority encoder, and say where each sits in a CPU.
9. You will be able to compute the maximum clock speed of a circuit from the delays of its gates.
10. You will be able to explain, from the hardware upward, why 0.1 plus 0.2 does not equal 0.3 on almost every computer you will ever touch.

5.1 Boolean algebra: true and false as 1 and 0

■ 5.1.1 PLAIN – in simple words

1. Ordinary algebra works with numbers. You add them, multiply them, and follow rules.
2. Boolean algebra works with only two values: true and false.
3. We write true as 1 and false as 0. That is all the values there are.
4. It has three basic operations: AND, OR and NOT.
5. AND is true when both of its two inputs are true.
6. OR is true when at least one of its two inputs is true.
7. NOT flips a value. It turns true into false and false into true.
8. That tiny system is enough to describe every calculation a computer does.
9. A man named George Boole invented it in the 1840s and 1850s.

10. He was doing philosophy and mathematics. He was not thinking about machines, because there were no electronic machines yet.
11. Almost a hundred years later, a student noticed that electrical switches obey exactly those same rules.
12. That student was Claude Shannon, and his observation is why computers work the way they do.

■ 5.1.2 PLAIN - a picture in your head

1. Think of a torch (a flashlight) with two switches wired one after the other along the same wire.
2. Current can only reach the bulb if switch one is closed AND switch two is closed.
3. One switch open anywhere in the line, and the bulb stays dark.
4. That wiring is an AND gate. Two switches in a line.
5. Now rewire the two switches side by side, each with its own path to the bulb.
6. Now the bulb lights if switch one is closed OR switch two is closed, or both.
7. That wiring is an OR gate. Two switches side by side.
8. So “in a line” means AND, and “side by side” means OR. Nothing more complicated than that is needed to start.
9. Where this comparison breaks: real gates do not pass the input current through to the output.
10. A real gate reads its inputs and then uses its own power supply to drive a fresh output. It is a decision, not a pipe.
11. That difference matters, because it is why you can chain a million gates together without the signal fading away.

■ 5.1.3 PLAIN - a worked example

1. Take a door that should unlock only when a valid card is presented and the time is inside working hours.
2. Call the card signal C. C is 1 when a valid card is seen.
3. Call the time signal T. T is 1 when the clock is between 09:00 and 18:00.
4. The unlock rule is $U = C \text{ AND } T$.
5. Here is every case.

C (card)	T (in hours)	U (unlock)
0	0	0
0	1	0
1	0	0
1	1	1

6. Now add a fire alarm. If the alarm F is 1, the door must open no matter what.
7. The new rule is $U = (C \text{ AND } T) \text{ OR } F$.
8. If F is 1, the OR makes U 1 whatever the left side says.
9. If F is 0, the OR passes the left side through unchanged.
10. You have just written a real access-control specification in Boolean algebra, and it can be built directly out of two gates.

■ 5.1.4 PLAIN - what is really happening inside

1. Inside a chip there is no “true” and no “false”. There is only voltage.
2. A wire is held at one of two voltages. In a typical modern core, about 0.75 volts or about 0 volts.
3. The high voltage is agreed to mean 1. The low voltage is agreed to mean 0.

4. That agreement is a choice made by the designers, not a law of nature.
5. There is a band of voltages in the middle that means nothing. Circuits are designed to pass through it as fast as possible and never rest there.
6. A gate is a small circuit that senses the voltages on its input wires.
7. It then connects its output wire either up to the power rail or down to ground, depending on what it senses.
8. Because the gate drives the output from its own power supply, the output is a clean full-strength 1 or 0.
9. That is called restoring the signal, and it is what lets you build circuits thousands of gates deep.
10. The honest version: the two voltages are not exact. A real 1 might be 0.71 volts on one wire and 0.78 volts on another.
11. The circuit only has to keep every 1 above a guaranteed minimum and every 0 below a guaranteed maximum. Those two numbers are in the datasheet.

■ 5.1.5 TECHNICAL – the engineer’s version

1. George Boole published “The Mathematical Analysis of Logic” in 1847 and “An Investigation of the Laws of Thought” in 1854.
2. Boole’s system was a two-valued algebra over the set $\{0, 1\}$ with the operations we now call conjunction, disjunction and complement.
3. Augustus De Morgan, a contemporary and correspondent of Boole, published “Formal Logic” in 1847, containing the two duality laws that carry his name.
4. Claude Elwood Shannon submitted his master’s thesis at the Massachusetts Institute of Technology on 10 August 1937, titled “A Symbolic Analysis of Relay and Switching Circuits”.
5. A revised, abridged version appeared in the Transactions of the American Institute of Electrical Engineers, volume 57, in 1938, pages 713 to 723.
6. The paper won the Alfred Noble Prize in 1939. That prize is named after an American engineer, not after Alfred Nobel of the Nobel Prizes.
7. In 1985 the psychologist Howard Gardner described it as “possibly the most important, and also the most famous, master’s thesis of the century”.
8. Shannon’s contribution was the mapping: a relay contact in series is a Boolean product, a relay contact in parallel is a Boolean sum, and a normally-closed contact is a complement.
9. That mapping turned circuit design from trial and error into algebra. You can now simplify a circuit by simplifying an expression.
10. Modern logic levels are defined per family. Some real figures:

Family	Supply	Minimum input high
74LS TTL	5.0 V	2.0 V
74HC CMOS	5.0 V	3.5 V
74LVC CMOS	3.3 V	2.0 V
LVC MOS18	1.8 V	1.17 V

11. These are standards written in vendor datasheets and in JEDEC documents, not conventions. A part that violates them is defective.
12. Positive logic means high voltage equals 1, and it is a near-universal convention. Negative logic, where low means 1, still appears on reset and chip-select pins, marked with an overbar or a leading letter n.

■ 5.1.6 WORDS – remember these

1. Boolean algebra — maths with only true and false — a two-element algebraic structure over $\{0, 1\}$ with AND, OR and NOT.
2. Bit — one true-or-false value — one binary digit, the smallest unit of information.
3. Logic level — which voltage counts as 1 — a specified voltage band with guaranteed input and output thresholds.
4. Gate — a small circuit that makes one decision — a combinational element implementing one Boolean function.
5. Truth table — a list of every input case and its answer — the complete functional specification of a combinational circuit.
6. Positive logic — high voltage means 1 — the near-universal signalling convention in current CMOS families.

5.2 Building gates from real MOSFETs

■ 5.2.1 PLAIN – in simple words

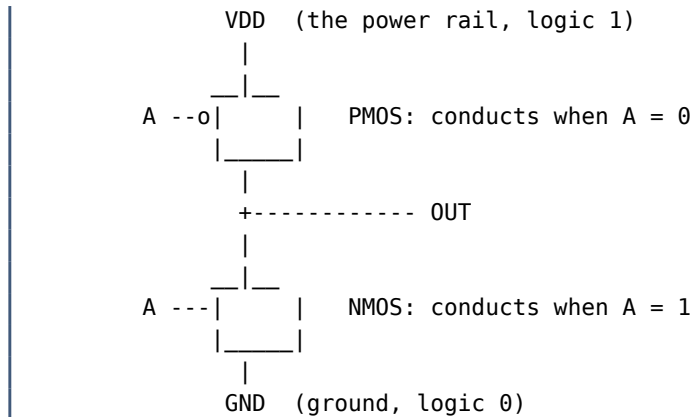
1. A transistor is a switch with no moving parts, controlled by a voltage on a third wire called the gate.
2. Modern chips use two kinds of transistor together. The style is called CMOS.
3. An NMOS transistor conducts when its gate is high. High turns it on.
4. A PMOS transistor conducts when its gate is low. Low turns it on. It is the opposite.
5. Every CMOS gate has two halves. A group of PMOS transistors on top, joined to the power supply.
6. And a group of NMOS transistors underneath, joined to ground.
7. The output wire sits between the two halves.
8. The design rule is simple. Exactly one half must be conducting at any time.
9. If the top half conducts, the output is pulled up to the power supply and reads as 1.
10. If the bottom half conducts, the output is pulled down to ground and reads as 0.
11. If both halves ever conducted at once, current would pour straight from power to ground and the chip would waste energy and heat up.

■ 5.2.2 PLAIN – a picture in your head

1. Imagine a bucket with a tap above it and a plug hole below it.
2. The tap is the PMOS half. The plug hole is the NMOS half.
3. Open the tap and close the plug: the bucket fills. That is output 1.
4. Close the tap and open the plug: the bucket empties. That is output 0.
5. The whole trick of CMOS is that the two are wired to be opposites. Whenever one opens, the other closes.
6. So the bucket is always either full or empty, and never both filling and draining at once.
7. Where this comparison breaks: water takes real time to fill a bucket, and so does charge, but the amount of charge involved is tiny.
8. The other break is that a real gate does pass a very small current while it is switching over, in the moment when both halves are briefly part-open.
9. That brief overlap is called short-circuit current, and it is a genuine part of a chip's power budget, though usually a small part.

■ 5.2.3 PLAIN – a worked example

- Here is the simplest CMOS gate. It has one input and one output, and it flips the value. It is the NOT gate, also called an inverter.

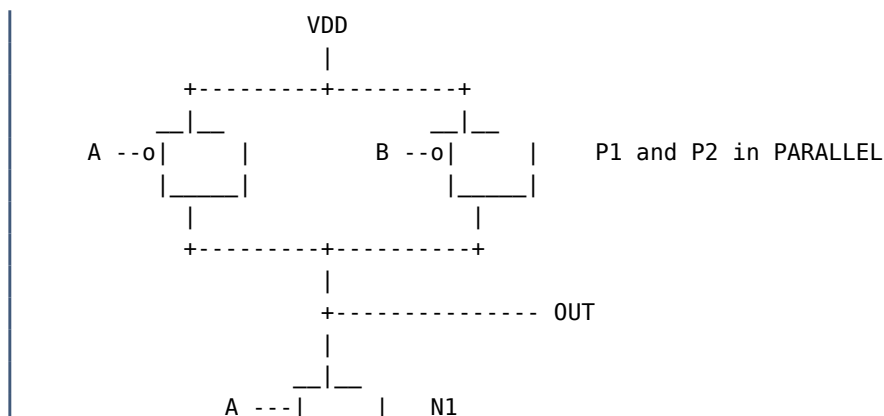


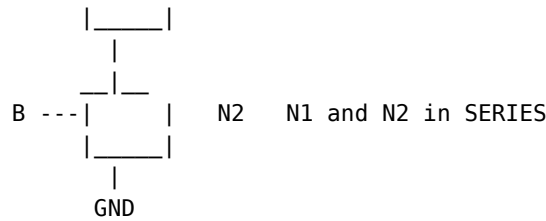
- The small circle on the PMOS gate is the standard way to mark “this one is on when the input is low”.
- Case $A = 0$. The PMOS is on, because its gate is low. The NMOS is off, because its gate is low. The output is connected up to VDD. $OUT = 1$.
- Case $A = 1$. The PMOS is off, because its gate is high. The NMOS is on. The output is connected down to GND. $OUT = 0$.
- Two transistors. That is the entire NOT gate.
- Note what the table shows: the output is always the opposite of the input, which is exactly the definition of NOT.

A	PMOS	NMOS	OUT
0	on	off	1
1	off	on	0

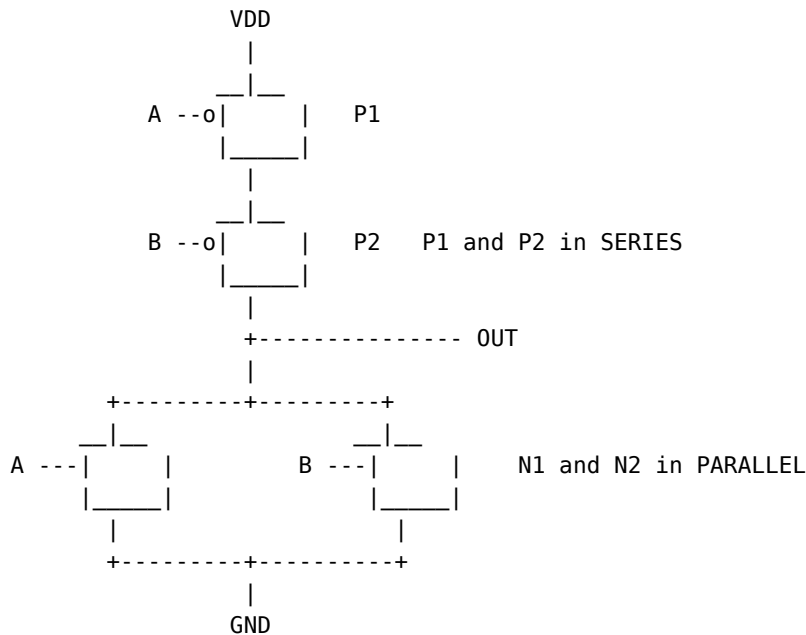
■ 5.2.4 PLAIN – what is really happening inside

- Now the NAND gate. NAND means NOT-AND. Its output is 0 only when both inputs are 1.
- The rule for building any CMOS gate is a rule of shapes. For AND-like behaviour, put transistors in series. For OR-like behaviour, put them in parallel.
- The NMOS half of a NAND has the two transistors in series, so the output can only be pulled down when both inputs are high.
- The PMOS half has the two transistors in parallel, so the output is pulled up if either input is low.





5. Case $A=0, B=0$. Both PMOS are on. Both NMOS are off. Output pulled up. $OUT = 1$.
6. Case $A=0, B=1$. P1 is on, P2 is off, so the parallel pull-up still conducts. N1 is off, so the series pull-down is broken. $OUT = 1$.
7. Case $A=1, B=0$. P1 is off, P2 is on, so the pull-up still conducts. N2 is off, so the pull-down is broken. $OUT = 1$.
8. Case $A=1, B=1$. Both PMOS are off. Both NMOS are on, so the series chain is complete. Output pulled down. $OUT = 0$.
9. Four transistors. That is the entire NAND gate.
10. The NOR gate is the same idea with the two halves swapped over. PMOS in series on top, NMOS in parallel underneath.



11. Case $A=0, B=0$. Both PMOS on, series chain complete, pull-up wins. $OUT = 1$.
12. Case $A=0, B=1$. P2 is off, so the series pull-up is broken. N2 is on, so the parallel pull-down conducts. $OUT = 0$.
13. Case $A=1, B=0$. P1 is off, pull-up broken. N1 is on, pull-down conducts. $OUT = 0$.
14. Case $A=1, B=1$. Both PMOS off. Both NMOS on. $OUT = 0$.
15. Four transistors again.
16. Notice something important. Both of these natural, cheap, four-transistor gates produce an inverted output. CMOS is naturally inverting.
17. To get a plain AND you must build a NAND and then add an inverter, which costs two more transistors. AND is six transistors, not four.

■ 5.2.5 TECHNICAL - the engineer's version

1. MOSFET stands for metal-oxide-semiconductor field-effect transistor. The gate terminal is insulated from the channel by a thin oxide layer.

2. An NMOS device conducts when the gate-to-source voltage exceeds its threshold voltage, typically around 0.2 to 0.4 volts in modern low-voltage processes.
3. CMOS as a circuit style was invented in 1963 by Frank Wanlass and Chih-Tang Sah at the Fairchild Semiconductor research laboratory.
4. They presented “Nanowatt Logic Using Field-Effect Metal-Oxide Semiconductor Triodes” at the International Solid-State Circuits Conference on 20 February 1963.
5. Wanlass received United States patent 3,356,858, filed 18 June 1963 and issued 5 December 1967.
6. The headline property of CMOS is that static power is near zero. Current flows mainly during switching, to charge and discharge load capacitance.
7. The dynamic power of a CMOS node is approximately $P = a \times C \times V \times V \times f$, where a is the activity factor, C the switched capacitance, V the supply voltage and f the frequency.
8. Because power scales with the square of the voltage, supply voltages fell from 5 volts in the 1980s to roughly 0.7 to 0.9 volts in current logic.
9. The honest version: static power is no longer negligible. At small geometries, sub-threshold leakage and gate-oxide tunnelling leakage can be a large share of total power, which is why unused blocks are power-gated.
10. Transistor counts for static complementary CMOS gates:

Gate	Transistors	Note
NOT (inverter)	2	1 PMOS, 1 NMOS
2-input NAND	4	PMOS parallel
2-input NOR	4	PMOS series
2-input AND	6	NAND plus inverter

11. A 2-input OR is likewise 6 transistors: a NOR plus an inverter.
12. A static CMOS 2-input XOR is commonly drawn with 12 transistors. A transmission-gate XOR needs 6, and various pass-transistor designs need 4. The exact count is an implementation detail of the standard-cell library, not a property of XOR.
13. In a real chip you do not draw transistors. You instantiate cells from a standard-cell library such as NAND2_X1 or INVX4, where the trailing number is the drive strength.
14. Tools that let you observe this: Yosys and OpenLane for open synthesis flows, ngspice for transistor-level simulation, and Magic or KLayout for viewing layout.

■ 5.2.6 WORDS – remember these

1. MOSFET — a voltage-controlled switch — a field-effect transistor with an insulated gate over a channel.
2. NMOS — the type that turns on with a high gate — an n-channel enhancement MOSFET conducting when V_{gs} is above threshold.
3. PMOS — the type that turns on with a low gate — a p-channel enhancement MOSFET conducting when V_{gs} is below its negative threshold.
4. CMOS — using both types together — complementary metal-oxide-semiconductor logic with a pull-up network and a complementary pull-down network.
5. Pull-up network — the half that connects the output to power — the PMOS network conducting for input patterns that produce a 1.
6. Pull-down network — the half that connects the output to ground — the NMOS network conducting for input patterns that produce a 0.

7. Standard cell — a ready-made gate layout — a characterized, pre-laid-out logic cell in a foundry library with published timing and power data.

5.3 The seven gates

■ 5.3.1 PLAIN - in simple words

1. There are seven gates you must know. Three basic ones and four built from them.
2. AND: output 1 only when both inputs are 1. Plain meaning: “both of them”.
3. OR: output 1 when at least one input is 1. Plain meaning: “either one, or both”.
4. NOT: one input, output is the opposite. Plain meaning: “the reverse”.
5. NAND: AND with the answer flipped. Plain meaning: “not both of them”.
6. NOR: OR with the answer flipped. Plain meaning: “neither of them”.
7. XOR: output 1 when the two inputs differ. Plain meaning: “exactly one of them”.
8. XNOR: XOR with the answer flipped. Plain meaning: “the two are the same”.
9. That is the whole vocabulary. Every digital circuit ever built is these seven, repeated.

■ 5.3.2 PLAIN - a picture in your head

1. Picture two people voting on whether to go out.
2. AND is a strict rule: we go only if both say yes.
3. OR is a relaxed rule: we go if anybody says yes.
4. NAND is a veto rule reversed: we go unless both say yes.
5. NOR is a very strict rule: we go only if nobody says yes.
6. XOR is a disagreement detector: it lights up exactly when the two people disagree.
7. XNOR is an agreement detector: it lights up exactly when they agree.
8. Where this comparison breaks: people can abstain, hesitate, or change their mind halfway. A gate input is always exactly 0 or exactly 1 at the moment the answer is read.
9. Real wires do spend a few picoseconds in between while changing. Circuits are designed so nobody reads the answer during that window. Chapter 6 covers how that is arranged with a clock.

■ 5.3.3 PLAIN - a worked example

1. Here are all seven truth tables. Each has at most two inputs.

AND — “both of them”:

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

OR — “either one, or both”:

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

NOT — “the reverse”:

A	NOT A
0	1
1	0

NAND — “not both of them”:

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

NOR — “neither of them”:

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

XOR — “exactly one of them”:

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

XNOR — “the two are the same”:

A	B	A XNOR B
0	0	1
0	1	0
1	0	0
1	1	1

2. Read down the last column of NAND and compare with AND. Every row is flipped. That is all “N” in front of a name means.
3. Read down XOR. It is 1 in exactly the two rows where A and B are different.
4. XOR is the single most useful gate in arithmetic, because adding 1 and 1 in binary gives 0 with a carry, and XOR gives exactly that 0.

■ 5.3.4 PLAIN – what is really happening inside

1. Nothing about a gate knows the word “AND”. The gate is a shape of transistors, and the truth table is what that shape does.
2. A gate does not compute in steps. It settles. You change the inputs, and after a short delay the output arrives at the right value.
3. That delay is called propagation delay, and it is the single most important number about a gate.
4. Gates with more inputs are slower, because more transistors sit in series in one of the two halves.
5. That is why a 4-input NAND is slower than a 2-input NAND, and why libraries usually stop at three or four inputs and build wider functions as trees.
6. XOR is not a natural CMOS shape. There is no simple series and parallel arrangement that produces “the inputs differ”.
7. So XOR is built from other gates or from special transistor tricks, and it is always more expensive and slower than NAND or NOR.
8. This has a real consequence you will meet in section 5.6: an adder is mostly XOR gates, and that is a large part of why addition is not free.

■ 5.3.5 TECHNICAL – the engineer’s version

1. Standard symbols come from two competing sets: the distinctive shapes of ANSI/IEEE 91-1984, and the rectangular symbols of IEC 60617-12.
2. In practice, distinctive shapes dominate American and textbook usage, and rectangles dominate European formal drawings. This is a convention split, not a correctness issue.
3. In hardware description languages the operators are written as symbols. Verilog uses & for AND, | for OR, ~ for NOT and ^ for XOR.

```

wire y_and  = a & b;
wire y_or   = a | b;
wire y_not  = ~a;
wire y_nand = ~(a & b);
wire y_nor  = ~(a | b);
wire y_xor  = a ^ b;
wire y_xnor = ~(a ^ b);

```

4. Discrete parts in the 7400 family, first shipped by Texas Instruments as the military 5400 series in October 1964 and the commercial plastic 7400 series in the third quarter of 1966:

Part	Function	Gates per package
7400	quad 2-input NAND	4
7402	quad 2-input NOR	4
7404	hex inverter	6
7486	quad 2-input XOR	4

5. The 7408 is quad 2-input AND and the 7432 is quad 2-input OR. It is not an accident that the very first part in the family, the 7400, was a NAND.
6. Typical transistor counts in a static CMOS standard-cell library:

Cell	Transistors	Relative delay
INV	2	1.0
NAND2	4	about 1.4

Cell	Transistors	Relative delay
NOR2	4	about 1.8
XOR2	8 to 12	about 2.5

- The relative delay figures are indicative for a typical library at equal drive strength, not a specification. Exact values come from the foundry characterization data, usually in Liberty format files with a .lib extension.
- NOR2 is slower than NAND2 at equal area, because a NOR's series transistors are PMOS. Hole mobility is two to three times lower than electron mobility, so those PMOS must be made wider.
- Fan-in is the number of inputs to a gate. Fan-out is the number of gate inputs a single output drives. Both increase delay, fan-out roughly linearly.
- The standard unit for comparing logic speed across process nodes is the fan-out-of-four inverter delay, written FO4. It is the delay of an inverter driving four copies of itself.

■ 5.3.6 WORDS – remember these

- XOR — exactly one of them is 1 — the two-input parity function, also called modulo-2 addition.
- XNOR — the two are the same — the equivalence function, the complement of XOR.
- NAND — not both of them — the complement of conjunction, and a functionally complete operator.
- NOR — neither of them — the complement of disjunction, also functionally complete.
- Propagation delay — how long a gate takes to answer — the time from an input crossing 50 percent of the supply to the output crossing 50 percent.
- Fan-in — how many inputs a gate has — the count of gate inputs, which raises delay through series transistor stacking.
- Fan-out — how many inputs one output feeds — the capacitive load on a driver, which raises delay roughly in proportion.
- FO4 — a fair speed yardstick — the delay of an inverter loaded by four identical inverters, used to compare process technologies.

5.4 Universality: why NAND alone is enough

■ 5.4.1 PLAIN – in simple words

- Here is a surprising fact. You do not need seven kinds of gate. You need one.
- If you have an unlimited supply of NAND gates and wire, you can build every other gate.
- And if you can build every other gate, you can build every digital circuit that has ever existed.
- A gate that can do this on its own is called universal, or functionally complete.
- NAND is universal. NOR is universal too, on its own, in exactly the same way.
- AND is not universal. Neither is OR. Neither is XOR. None of them can produce a NOT.
- The reason is simple: AND, OR and XOR of two 0s all give 0. They can never turn a 0 into a 1 on their own.
- NAND can. NAND of two 0s gives 1. That ability to invert is the key.

■ 5.4.2 PLAIN – a picture in your head

- Think of a language with only one word, where the word changes meaning depending on how you repeat and arrange it.
- That sounds useless, but it is exactly what NAND is. One operation, rearranged, becomes all the others.

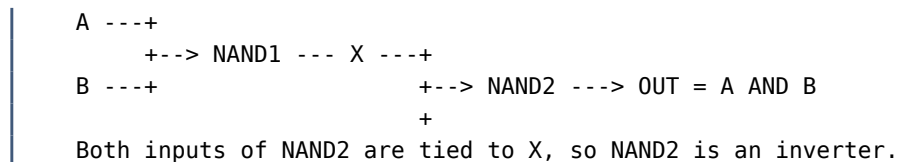
3. A closer everyday comparison is Lego bricks. There are many shapes, but you could build almost any model from one single brick shape, given enough of them.
4. It would be less elegant and use more bricks, but it would work, and you would only need one bin of parts.
5. Where this comparison breaks: with Lego, using one brick shape is a handicap. With NAND, it is often an advantage.
6. NAND is the cheapest and fastest gate CMOS can make, so building things out of NAND is not a sacrifice. It is frequently the best option.

■ 5.4.3 PLAIN – a worked example

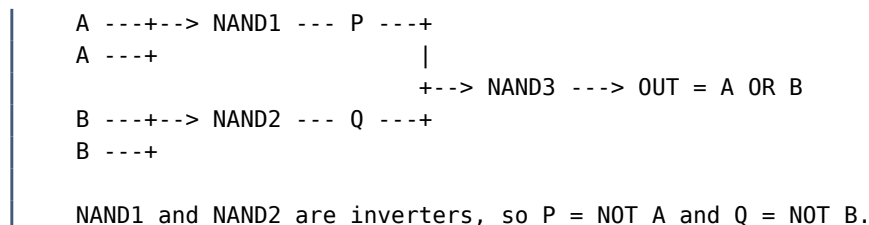
1. NOT from one NAND. Tie both inputs together.



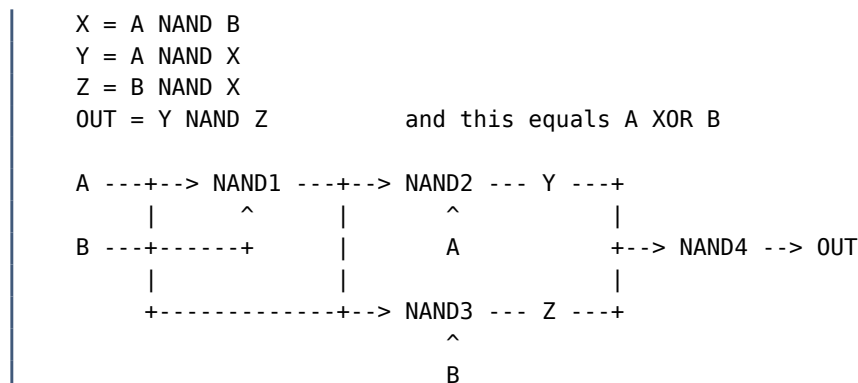
2. Check it. If $A = 0$, then $\text{NAND}(0,0) = 1$. If $A = 1$, then $\text{NAND}(1,1) = 0$. That is NOT.
3. AND from two NANDs. NAND then invert.



4. Written plainly: $X = A \text{ NAND } B$, then $\text{OUT} = X \text{ NAND } X$. Two gates.
5. OR from three NANDs. Invert both inputs first, then NAND them.



6. Written plainly: $P = \text{NOT } A$, $Q = \text{NOT } B$, $\text{OUT} = P \text{ NAND } Q$. Three gates.
7. Why does that give OR? Because $\text{NAND}(\text{NOT } A, \text{NOT } B)$ is $\text{NOT}(\text{NOT } A \text{ AND } \text{NOT } B)$, and “not (neither)” means “at least one”. That is OR.
8. XOR from four NANDs. This is the classic arrangement.



9. Step by step with $A = 1$, $B = 0$.
10. $\text{NAND1} = \text{NAND}(1, 0) = 1$.

11. $\text{NAND2} = \text{NAND}(A, \text{NAND1}) = \text{NAND}(1, 1) = 0$.
12. $\text{NAND3} = \text{NAND}(B, \text{NAND1}) = \text{NAND}(0, 1) = 1$.
13. $\text{NAND4} = \text{NAND}(0, 1) = 1$. And $\text{XOR}(1, 0) = 1$. Correct.
14. Now $A = 1, B = 1$. $\text{NAND1} = 0$. $\text{NAND2} = \text{NAND}(1, 0) = 1$. $\text{NAND3} = \text{NAND}(1, 0) = 1$. $\text{NAND4} = \text{NAND}(1, 1) = 0$. And $\text{XOR}(1, 1) = 0$. Correct.

Gate built	NAND gates needed
NOT	1
AND	2
OR	3
XOR	4

15. NOR from NAND takes 4 as well: build OR with 3, then invert with 1.

■ 5.4.4 PLAIN – what is really happening inside

1. The deep reason NAND works is that it combines two abilities in one gate.
2. It can combine two signals, and it can invert. Any gate that can do both can build everything.
3. A formal way to see it: every Boolean function can be written as a sum of products, meaning ORs of ANDs of inputs and inverted inputs.
4. If you can make AND, OR and NOT, you can write any sum of products. And NAND makes all three.
5. So NAND builds every possible function of any number of inputs. There are no exceptions and no special cases.
6. NOR is universal for the mirror-image reason: it can combine and it can invert, so it builds product of sums instead.
7. The honest version: universality is a statement about what is possible, not about what is sensible.
8. Building a 64-bit multiplier out of nothing but 2-input NAND gates is possible and would be enormous and slow. Real designs use whatever cell in the library is cheapest for the job.

■ 5.4.5 TECHNICAL – the engineer's version

1. Functional completeness of the Sheffer stroke was shown by Henry M. Sheffer in his 1913 paper in the Transactions of the American Mathematical Society. The Sheffer stroke is NAND.
2. Charles Sanders Peirce had found the same result earlier, in unpublished work dated around 1880. The NOR operator is still called Peirce's arrow.
3. NAND is preferred over NOR in CMOS for a physical reason. In a NAND, the series stack is NMOS and the parallel network is PMOS.
4. Electron mobility in silicon is roughly 1400 square centimetres per volt-second, while hole mobility is roughly 450. The ratio is about 3 in bulk, and about 2 to 2.5 for carriers in a MOSFET inversion layer.
5. Because PMOS devices carry less current per unit width, a series PMOS stack in a NOR must be made two to three times wider than the equivalent NMOS stack in a NAND to reach the same speed.
6. Wider transistors mean more area, more input capacitance and more power. So NOR2 costs more than NAND2 for the same performance.
7. Consequence: standard-cell libraries are NAND-heavy, synthesis tools map preferentially onto NAND and inverter, and NAND2_X1 is typically the most instantiated cell on a die.
8. NAND flash memory is named for the same series-stack idea, with memory cells in series along a bit line. It is a separate use of the word from NAND logic and the two should not be confused.

9. In FPGA design the picture changes completely. An FPGA has no gates. It has lookup tables, typically 6-input LUTs in current Xilinx and AMD parts and in Intel and Altera adaptive logic modules.
10. A 6-input LUT is a 64-bit memory addressed by the six inputs. It implements any function of six variables at identical cost, so gate-count reasoning does not apply. This is an implementation detail of FPGAs, not of logic.

■ 5.4.6 WORDS – remember these

1. Universal gate — one gate that can build all the rest — a functionally complete operator whose closure is all Boolean functions.
2. Functional completeness — nothing else is needed — the property that every Boolean function is expressible using only the given operator set.
3. Sheffer stroke — another name for NAND — the binary connective written as a vertical bar, proved complete by Sheffer in 1913.
4. Peirce arrow — another name for NOR — the dual complete connective, attributed to Charles Sanders Peirce.
5. LUT — a small table that fakes any gate — a lookup table in an FPGA, a configuration memory addressed by the input signals.
6. Technology mapping — turning a design into real cells — the synthesis step that covers a Boolean network with cells from a target library.

5.5 Writing and simplifying logic

■ 5.5.1 PLAIN – in simple words

1. A truth table says what a circuit must do. An expression says how to build it. You need to move between the two.
2. Going from a table to an expression is mechanical. Look at every row where the output is 1.
3. For each such row, write an AND of all the inputs, with a NOT on any input that is 0 in that row.
4. Then OR all those AND terms together. You now have a working circuit.
5. That form is called sum of products, because OR behaves like a sum and AND behaves like a product.
6. The expression you get this way is correct but usually wasteful. It has one AND term for every 1 in the table.
7. Simplifying means finding a smaller expression that gives exactly the same table. Fewer gates, less area, less power, less delay.
8. There are two ways to simplify. By algebra, using rules. Or by drawing a picture called a Karnaugh map and spotting groups by eye.

■ 5.5.2 PLAIN – a picture in your head

1. Think of writing driving directions. The literal version lists every single junction: “at junction 1 go straight, at junction 2 go straight, at junction 3 turn left”.
2. The simplified version says: “go straight until the church, then turn left”.
3. Both get you to the same place. The second is shorter because it noticed a run of identical steps and merged them.
4. Simplifying logic is exactly that. You look for groups of rows that agree on most inputs, and merge them into one shorter rule.
5. Where this comparison breaks: with directions there is one obvious way to merge. With logic there can be several different smallest answers, all equally good.

6. Also, the smallest expression is not always the fastest circuit. A shallower circuit with more gates can beat a deeper circuit with fewer.

■ 5.5.3 PLAIN – a worked example

1. Here are the core identities. A prime mark means NOT.

Name	Rule
Identity	$A + 0 = A, A \cdot 1 = A$
Null	$A + 1 = 1, A \cdot 0 = 0$
Idempotent	$A + A = A, A \cdot A = A$
Complement	$A + A' = 1, A \cdot A' = 0$
Involution	$(A')' = A$
Absorption	$A + A \cdot B = A$
Distributive	$A \cdot (B + C) = A \cdot B + A \cdot C$
Consensus	$A \cdot B + A' \cdot C + B \cdot C = A \cdot B + A' \cdot C$

2. In these lines a dot means AND and a plus means OR. That is the standard written shorthand.
 3. Now De Morgan's two laws, proved by writing out every case.
 4. First law: NOT (A AND B) equals (NOT A) OR (NOT B).

A	B	$(A \cdot B)'$	$A' + B'$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

5. The last two columns match on every row. The law holds. That is a complete proof, because there are only four cases and we checked all four.
 6. Second law: NOT (A OR B) equals (NOT A) AND (NOT B).

A	B	$(A + B)'$	$A' \cdot B'$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

7. Again both columns match on every row. The law holds.
 8. In plain words: “not both” is the same as “either one is missing”, and “neither” is the same as “this one is missing and that one is missing”.

■ 5.5.4 PLAIN – what is really happening inside

1. Now a complete Karnaugh map for four variables, from the table to the answer.
 2. The specification: four sensor bits A, B, C and D arrive, with A as the most significant bit. Output F must be 1 for these input values, and 0 otherwise.
 3. This function is invented for teaching. The method is the real thing.

Value	A	B	C	D	F
0	0	0	0	0	1
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	1
11	1	0	1	1	1
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	1
15	1	1	1	1	1

4. Step 1. Write the raw sum of products straight from the ten rows with a 1. That is ten AND terms of four inputs each. It works and it is horrible.
5. Step 2. Draw the map. The rows are AB, the columns are CD, and both are listed in the order 00, 01, 11, 10.
6. That strange order is Gray code. Only one bit changes between neighbours, so squares that touch differ in exactly one variable.

	CD=00	CD=01	CD=11	CD=10	
AB=00	1	1	1	1	(m0 m1 m3 m2)
AB=01	1	1	0	0	(m4 m5 m7 m6)
AB=11	0	0	1	1	(m12 m13 m15 m14)
AB=10	0	0	1	1	(m8 m9 m11 m10)

7. Step 3. Find the biggest rectangles of 1s. Sizes must be 1, 2, 4, 8 or 16, and rectangles may wrap around the edges.
8. Group one: the whole top row, four squares, m0 m1 m3 m2. Across that row A is always 0 and B is always 0, while C and D take every value. So the term is $A'B'$.
9. Group two: the left two columns of the top two rows, m0 m1 m4 m5. Here A is always 0 and C is always 0. The term is $A'C'$.
10. Group three: the right two columns of the bottom two rows, which is m15, m14, m11 and m10. Across those four squares A is always 1 and C is always 1. The term is AC .
11. Step 4. Check the cover. $A'B'$ covers 0, 1, 2, 3. $A'C'$ covers 0, 1, 4, 5. AC covers 10, 11, 14, 15. Together that is all ten 1s and nothing else.
12. Step 5. Write the answer.

$$F = A'B' + A'C' + AC$$

13. Ten four-input AND terms became three two-input AND terms. That is three AND gates, one OR gate and two inverters, instead of dozens of gates.
14. Sanity check value 6, which is A=0 B=1 C=1 D=0. $A'B'$ is 0 because B is 1. $A'C'$ is 0 because C is 1. AC is 0 because A is 0. So $F = 0$, matching the table.
15. Sanity check value 11, which is A=1 B=0 C=1 D=1. $A'B'$ is 0 and $A'C'$ is 0, both because A is 1. AC is 1. So $F = 1$, matching the table.

■ 5.5.5 TECHNICAL – the engineer’s version

1. The canonical sum of products form is also called the minterm expansion or disjunctive normal form. The dual is the maxterm expansion, or conjunctive normal form.
2. A minterm is a product term containing every variable exactly once. Minterm m_5 of four variables is $A'BC'D$, because 5 is binary 0101.
3. The standard notation for the function above is $F(A,B,C,D) = \text{sum of } m(0, 1, 2, 3, 4, 5, 10, 11, 14, 15)$.
4. Maurice Karnaugh published “The Map Method for Synthesis of Combinational Logic Circuits” in the Transactions of the American Institute of Electrical Engineers in November 1953.
5. It refined Edward W. Veitch’s 1952 Veitch chart, which was itself a rediscovery of Allan Marquand’s logical diagram of 1881.
6. Karnaugh maps are practical up to four variables, awkward at five and six, and useless beyond that. Beyond four variables, use an algorithm.
7. The Quine-McCluskey algorithm, published by Willard Quine in 1952 and extended by Edward McCluskey in 1956, gives a guaranteed minimal cover but has exponential worst-case cost.
8. Production tools use Espresso, developed at the University of California, Berkeley in the early 1980s. Espresso is a heuristic minimizer: it is fast and very good, but it does not promise the absolute minimum.
9. A prime implicant is a product term that cannot be enlarged further without covering a 0. An essential prime implicant is one that uniquely covers at least one minterm.
10. In the worked map, all three terms are essential prime implicants, so the minimal cover is unique. That is not always the case.
11. Don’t-care conditions, written X or d, are input combinations that cannot occur. They may be read as 1 or 0, whichever makes groups larger. Binary-coded decimal decoders are the classic case: 10 to 15 never occur.
12. De Morgan’s laws generalize to any number of variables and are the formal basis of bubble pushing, the visual technique of moving inversion circles across a gate symbol while swapping AND for OR.
13. Modern practice: nobody minimizes by hand for production. You write the behaviour in Verilog, SystemVerilog or VHDL and a synthesis tool such as Synopsys Design Compiler, Cadence Genus or the open-source Yosys performs minimization and technology mapping.
14. You still do it by hand to read a datasheet, to debug a synthesis result, and to pass an interview.

■ 5.5.6 WORDS – remember these

1. Sum of products — ORs of ANDs — disjunctive normal form, a two-level AND-OR realization.
2. Minterm — one row of the table as a term — a product containing every variable in true or complemented form.
3. Karnaugh map — a grid that makes groups visible — a Gray-coded truth table arrangement placing logically adjacent minterms physically adjacent.
4. Gray code — an order where one bit changes at a time — a reflected binary code with unit Hamming distance between neighbours.
5. Prime implicant — a group that cannot grow — a product term implying the function that is not contained in any larger such term.
6. Don’t care — a case that cannot happen — an unspecified output used to enlarge groups during minimization.
7. De Morgan’s laws — not both means either is missing — the dual identities $(A.B)' = A' + B'$ and $(A+B)' = A'.B'$.

5.6 Adding

■ 5.6.1 PLAIN – in simple words

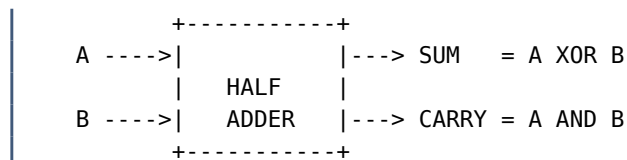
1. Binary addition has only four cases for a single column: $0+0$, $0+1$, $1+0$ and $1+1$.
2. The first three give 0, 1 and 1. The fourth gives 0 with a 1 carried into the next column.
3. So each column produces two outputs: a sum bit and a carry bit.
4. The sum bit is 1 when exactly one input is 1. That is XOR.
5. The carry bit is 1 when both inputs are 1. That is AND.
6. A circuit with two inputs producing those two outputs is called a half adder.
7. It is called half because it cannot accept a carry coming in from the column to its right.
8. Add that third input and you have a full adder. A full adder takes three bits in and produces a sum bit and a carry bit out.
9. Chain eight full adders together, feeding each carry-out into the next carry-in, and you can add two 8-bit numbers.

■ 5.6.2 PLAIN – a picture in your head

1. Think of a line of eight people, each holding two coins and passing notes to the left.
2. Each person adds their two coins plus any note passed to them from the right.
3. If the total is 2 or 3, they pass a note left saying “carry one”, and keep the remainder.
4. The person on the far right starts with no incoming note.
5. Nobody can finish until the person on their right has finished, because they are waiting for the note.
6. So the total time is not the time for one person. It is the time for the note to travel all the way from the right end to the left end.
7. That waiting chain is the single reason addition is not instant, and it is what carry-lookahead exists to fix.
8. Where this comparison breaks: the people do not really take turns. All eight full adders are working continuously and all the time, on whatever values currently sit on their wires.
9. The values just happen to be wrong until the carry has finished travelling. The circuit does not know it is wrong. It simply settles.

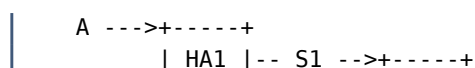
■ 5.6.3 PLAIN – a worked example

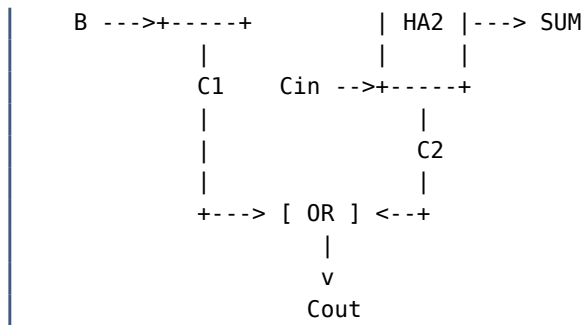
1. The half adder.



A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

2. The full adder, built from two half adders and one OR gate.





3. The first half adder adds A and B. The second adds that partial sum to the incoming carry.
4. A carry can be produced by either half adder, but never by both, so a plain OR is enough to combine them.
5. Here is the full adder truth table, split so no table is wider than four columns.

A	B	Cin	SUM
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

A	B	Cin	Cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

6. In equations: $SUM = A \oplus B \oplus Cin$, and $Cout = (A \text{ AND } B) \text{ OR } (Cin \text{ AND } (A \oplus B))$.
7. Read the Cout column again: it is 1 whenever two or three of the inputs are
 1. Cout is a majority vote of three bits.
8. Now the 8-bit ripple-carry adder. Eight full adders in a line.

```

Cin=0 -> [FA0] -c1-> [FA1] -c2-> [FA2] -c3-> [FA3] -c4->
        -> [FA4] -c5-> [FA5] -c6-> [FA6] -c7-> [FA7] -> Cout
  
```

Each FAn also takes bit n of A and bit n of B, and produces bit n of the sum.

9. Worked sum. Add 109 and 46, which are 0110 1101 and 0010 1110 in binary.
10. Bit 0 is the rightmost. Here is the full carry trace.

bit	A	B	Cin	SUM	Cout
0	1	0	0	1	0
1	0	1	0	1	0
2	1	1	0	0	1
3	1	1	1	1	1
4	0	0	1	1	0
5	1	1	0	0	1
6	1	0	1	0	1
7	0	0	1	1	0

11. Reading the SUM column from bit 7 down to bit 0 gives 1001 1011, which is 155. And 109 plus 46 is 155. Correct.
12. Watch bit 2 and bit 3. Bit 2 generates a carry, which bit 3 receives, and bit 3 passes another one on. That is a carry rippling.
13. The worst case for rippling is 0111 1111 plus 0000 0001, which is 127 plus
 1. The carry born at bit 0 must travel through every one of the eight stages before the answer is right.

■ 5.6.4 PLAIN - what is really happening inside

1. Every full adder in the chain starts computing the moment its inputs change. None of them waits for permission.
2. But seven of the eight are computing with a carry-in that has not settled yet, so seven of them are producing a wrong answer at first.
3. As the correct carry arrives at each stage in turn, that stage recomputes and its output may flip. The flipping travels left like a wave.
4. Those intermediate wrong values are called glitches. They are real voltage changes, they burn real power, and they are harmless as long as nothing reads the output too early.
5. The time to settle is roughly the delay of one stage's carry path multiplied by the number of stages.
6. Double the width from 8 bits to 16 bits and you roughly double the delay. That is a terrible scaling law for a 64-bit machine.
7. The fix is to stop waiting. Notice that a stage does not need the actual carry to know how it will behave.
8. A stage generates a carry if both its bits are 1, no matter what comes in. Call that G.
9. A stage propagates a carry if exactly one of its bits is 1, meaning it will pass on whatever comes in. Call that P.
10. G and P can be computed for all 64 bits simultaneously, because they depend only on A and B and not on any carry.
11. Then the carry into any position is a single wide formula built from the G and P values below it, computed in a fixed small number of gate levels.
12. That is carry-lookahead. It trades a lot of extra gates for a delay that grows like the logarithm of the width instead of linearly.

■ 5.6.5 TECHNICAL - the engineer's version

1. The generate and propagate signals are $G(i) = A(i) \cdot B(i)$ and $P(i) = A(i) \text{ XOR } B(i)$. Some texts use $P(i) = A(i) + B(i)$, which is equivalent for carry purposes but not for the sum.
2. The recurrence is $C(i+1) = G(i) + P(i) \cdot C(i)$.
3. Expanding for four bits gives the classic flat lookahead equation:

$$\begin{aligned} C1 &= G0 + P0 \cdot C0 \\ C2 &= G1 + P1 \cdot G0 + P1 \cdot P0 \cdot C0 \end{aligned}$$

$$C3 = G2 + P2.G1 + P2.P1.G0 + P2.P1.P0.C0$$

$$C4 = G3 + P3.G2 + P3.P2.G1 + P3.P2.P1.G0 + P3.P2.P1.P0.C0$$

4. Every one of those is two gate levels after P and G are available, so a 4-bit block produces its carry-out in about three levels rather than eight.
5. Flat lookahead does not scale past about four bits, because the AND terms grow in both count and fan-in. Real designs build a tree of 4-bit blocks with block-level group generate and group propagate signals.
6. Gerald B. Rosenberger of IBM filed for a binary carry-lookahead adder in 1957 and received United States patent 2,966,305 in 1960. Konrad Zuse is believed to have used carry anticipation in the Z1 in the 1930s.
7. Weinberger and Smith published “A One-Microsecond Adder Using One-Megacycle Circuitry” in the IRE Transactions on Electronic Computers in 1956, which is the standard reference for the lookahead equations above.
8. Delay comparison for a 16-bit adder, counting gate levels:

Adder type	Gate levels	Area cost
Ripple carry	about 31	lowest
4-bit block lookahead	about 8	moderate
Kogge-Stone prefix	about 9	highest

9. Prefix adders treat carry generation as a parallel prefix problem. Kogge-Stone, from Peter Kogge and Harold Stone in 1973, has minimum depth and maximum wiring. Brent-Kung, from 1982, trades depth for far less wiring.
10. Sklansky and Han-Carlson sit between those two. Experts disagree on the best default: some favour Kogge-Stone for depth, others argue its wiring load makes Han-Carlson faster in practice at modern process nodes.
11. The 7483 and 74283 are 4-bit full adders with internal fast carry. The 74182 is a lookahead carry generator designed to sit above four 74181 ALU slices.
12. In Verilog you simply write the plus sign and let the synthesis tool choose the architecture, guided by your timing constraint:

```

module add8 (input  [7:0] a, b,
              input    cin,
              output [7:0] sum,
              output    cout);
    assign {cout, sum} = a + b + cin;
endmodule

```

■ 5.6.6 WORDS - remember these

1. Half adder — adds two bits, no carry in — a circuit producing $S = A \text{ XOR } B$ and $C = A \text{ AND } B$.
2. Full adder — adds three bits — a circuit producing sum and carry from A, B and a carry-in.
3. Ripple carry — each stage waits for the last — a linear-delay adder where carry-out chains into the next carry-in.
4. Carry generate — this column makes a carry regardless — $G = A \text{ AND } B$.
5. Carry propagate — this column passes a carry through — $P = A \text{ XOR } B$.
6. Carry-lookahead — work out all carries at once — a logarithmic-depth carry network computed from G and P without waiting.

7. Glitch — a wrong value on the way to the right one — a transient logic hazard caused by unequal path delays.

5.7 Subtracting without a subtractor

■ 5.7.1 PLAIN – in simple words

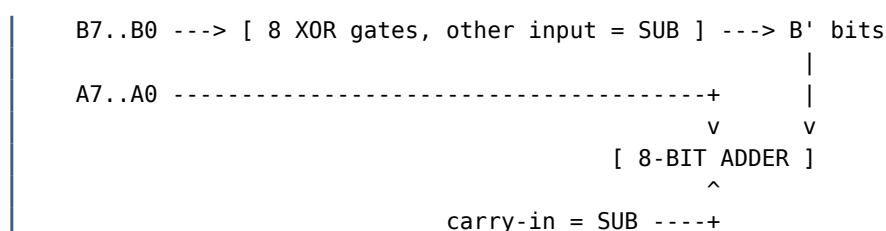
1. A computer does not contain a subtractor. It contains an adder, and it cheats.
2. To compute A minus B, it computes A plus the negative of B.
3. So the only question is how to represent a negative number in bits.
4. The scheme every modern machine uses is called two's complement.
5. The rule to negate a number is: flip every bit, then add 1.
6. Take 5 in 8 bits: 0000 0101. Flip every bit: 1111 1010. Add 1: 1111 1011. That is minus 5.
7. To check it, add 5 and minus 5. 0000 0101 plus 1111 1011 is 1 0000 0000. The ninth bit falls off the end and you are left with zero. Correct.
8. The top bit acts as a sign flag. If it is 1, the number is negative.
9. In 8 bits the range is minus 128 to plus 127. There is one more negative value than positive, because zero uses up one of the non-negative slots.

■ 5.7.2 PLAIN – a picture in your head

1. Think of a car odometer with only three digits, so it can show 000 to 999.
2. Wind it back one from 000 and it shows 999. So on that dial, 999 behaves exactly like minus one.
3. Add 5 to 999 and you get 1004, but the leading 1 cannot be shown, so the dial reads 004. And 5 minus 1 is 4. It worked.
4. Two's complement is that same wrap-around idea with two digits per position instead of ten.
5. The bits that fall off the left end are simply discarded, and the arithmetic still comes out right.
6. Where this comparison breaks: on an odometer, everything above 500 is not automatically treated as negative. It is just a big number.
7. In two's complement, the machine decides which half is negative by the top bit, and it is the instruction you choose that decides whether the same bits mean 200 or minus 56.

■ 5.7.3 PLAIN – a worked example

1. Compute 45 minus 67 in 8 bits.
2. 45 is 0010 1101. 67 is 0100 0011.
3. Flip every bit of 67: 1011 1100.
4. Feed that to the adder together with 45, and set the carry-in to 1. The carry-in supplies the “add one” for free, with no extra gate.
5. 0010 1101 plus 1011 1100 plus 1 gives 1110 1010, with a carry-out of 0.
6. Read 1110 1010 as a signed value. The top bit is 1, so it is negative.
7. To read it, flip and add one: 0001 0101 plus 1 is 0001 0110, which is 22. So the answer is minus 22.
8. And 45 minus 67 is minus 22. Correct.
9. Here is the whole trick in one diagram. One control wire, called SUB, does everything.



10. When SUB is 0, XOR with 0 leaves every B bit unchanged and the carry-in is 0. The circuit computes A plus B.
11. When SUB is 1, XOR with 1 flips every B bit and the carry-in is 1. The circuit computes A minus B.
12. Cost of adding subtraction to an adder: eight XOR gates and one wire. That is the whole subtractor.

■ 5.7.4 PLAIN – what is really happening inside

1. Two's complement works because of arithmetic modulo 256 for 8 bits, or modulo 2 to the power 64 for 64 bits.
2. Flipping every bit of B gives 255 minus B, because the all-ones pattern is 255 and flipping is the same as subtracting from all ones.
3. Adding 1 gives 256 minus B.
4. So A plus the flipped B plus 1 equals A plus 256 minus B.
5. The 256 is exactly one more than the largest 8-bit value, so it appears only in the ninth bit position, which the hardware drops.
6. What is left on the wires is A minus B. The mathematics is exact, not an approximation.
7. Now, the hardware must also spot when the answer does not fit. There are two separate failure signals, and they are not the same thing.
8. Carry-out means the result was too big for the width, treating the numbers as unsigned. It is the ninth bit that fell off.
9. Overflow means the result was too big for the width, treating the numbers as signed. It is detected differently.
10. The signed overflow rule is short: overflow happens when the carry into the top bit differs from the carry out of the top bit.
11. In one XOR gate: $V = C(n) \text{ XOR } C(n-1)$.
12. A second way to say the same thing: overflow happens if two numbers of the same sign are added and the answer has the opposite sign.
13. Adding a positive and a negative number can never overflow, because the answer is always between the two inputs.

■ 5.7.5 TECHNICAL – the engineer's version

1. Two's complement is the representation mandated by essentially every current general-purpose instruction set: x86-64, ARM AArch64, RISC-V and POWER.
2. Since the C23 standard, published in 2024, signed integers in C are required to use two's complement. Before that, C permitted sign-magnitude and one's complement too, though no mainstream compiler used them.
3. The asymmetry is real and it bites. In 8 bits, negating minus 128 gives back minus 128, because plus 128 does not exist. The same holds at every width.
4. In C, computing 0 minus INT_MIN, or INT_MIN divided by minus 1, is undefined behaviour. On x86-64 the IDIV instruction raises a divide-error exception, the same one as division by zero.
5. Worked signed overflow example: 100 plus 50 in 8 bits. 0110 0100 plus 0011 0010 gives 1001 0110.
6. As unsigned that is 150 and there was no carry-out, so the unsigned carry flag is 0.
7. As signed it is minus 106, which is wrong. The carry into bit 7 was 1 and the carry out of bit 7 was 0, so $V = 1 \text{ XOR } 0 = 1$. Overflow is flagged.
8. Flag conventions differ, and this is a genuine trap when porting code.

Machine	After SUB, carry means
x86 / x86-64	1 means a borrow happened
ARM AArch64	1 means no borrow
RISC-V	no flags at all

9. This is a convention, not a standard. On ARM the C flag is the literal carry out of A plus NOT B plus 1. On x86 it is inverted before being stored.
10. RISC-V has no condition-code register by design. Comparison is done with instructions such as SLT and SLTU that write a 0 or 1 into a normal register, and branches compare two registers directly.
11. Tools to observe this: in GDB, “info registers eflags” on x86-64 shows CF, ZF, SF and OF. In LLDB, “register read cpsr” shows N, Z, C and V on ARM.
12. Compilers exploit signed overflow being undefined behaviour in C and C++ to optimize loops. Build with -fwrapv to force wrapping, or use -fsanitize=signed-integer-overflow to trap it at run-time.

■ 5.7.6 WORDS – remember these

1. Two’s complement — flip the bits and add one — the radix complement representation where the top bit has weight minus 2 to the power n-1.
2. Sign bit — the top bit says positive or negative — bit n-1 of a two’s complement value.
3. Carry flag — a bit fell off the end, unsigned — the carry-out of the most significant adder stage, inverted after subtraction on x86.
4. Overflow flag — the signed answer is wrong — set when the carry into the sign bit differs from the carry out of it.
5. Modular arithmetic — numbers wrapping round a dial — arithmetic modulo 2 to the power n, which is what fixed-width binary hardware performs.
6. Undefined behaviour — the language makes no promise — a construct for which the C and C++ standards impose no requirements, permitting any result.

5.8 More building blocks

■ 5.8.1 PLAIN – in simple words

1. Beyond gates and adders there is a small set of standard parts. Every CPU is made mostly of copies of these.
2. A multiplexer is a chooser. It has many data inputs, a few select inputs, and one output. The select value picks which input reaches the output.
3. A demultiplexer is the reverse. One data input, many outputs, and the select value picks which output receives it.
4. A decoder turns a small binary number into one hot wire. Three input bits become eight output wires, exactly one of which is 1.
5. An encoder is the reverse of a decoder. One hot wire in, a binary number out.
6. A priority encoder is an encoder that copes when several inputs are 1 at once. It reports the highest-numbered one and ignores the rest.
7. A comparator takes two numbers and says whether they are equal, or which one is bigger.
8. A barrel shifter moves all the bits of a word left or right by any amount, in one step, with no looping.
9. A parity generator counts whether the number of 1 bits is odd or even. It is one XOR tree.

■ 5.8.2 PLAIN – a picture in your head

1. Think of a railway station. A multiplexer is the points where many tracks merge into one platform, and a lever chooses which train comes through.
2. A demultiplexer is the points at the far end, where one track fans out to many platforms.
3. A decoder is the departure board lighting up exactly one platform number.
4. A priority encoder is the station controller when three trains all signal at once: the express goes first and the others wait.
5. A barrel shifter is the whole train being moved sideways onto a parallel track, all carriages at once, rather than shunted one carriage at a time.
6. Where this comparison breaks: trains physically travel and take time proportional to distance. In a multiplexer no data moves through the unused paths at all. The unselected inputs are simply blocked.
7. Also, a real multiplexer does not queue. If two inputs change, both are read continuously. Only one is passed on.

■ 5.8.3 PLAIN – a worked example

1. A worked 3-to-8 decoder. Three inputs A2, A1, A0 and eight outputs Y0 to Y7.
2. The rule is: Y(n) is 1 when the input bits spell the number n in binary, and 0 otherwise. Exactly one output is ever 1. That is called one-hot.

A2	A1	A0	Value	Output that is 1
0	0	0	0	Y0
0	0	1	1	Y1
0	1	0	2	Y2
0	1	1	3	Y3
1	0	0	4	Y4
1	0	1	5	Y5
1	1	0	6	Y6
1	1	1	7	Y7

3. Each output is a single 3-input AND gate over the inputs, with inverters where a 0 is required.

$$\begin{aligned}
 Y0 &= A2' \cdot A1' \cdot A0' \\
 Y1 &= A2' \cdot A1' \cdot A0 \\
 Y2 &= A2' \cdot A1 \cdot A0' \\
 Y3 &= A2' \cdot A1 \cdot A0 \\
 Y4 &= A2 \cdot A1' \cdot A0' \\
 Y5 &= A2 \cdot A1' \cdot A0 \\
 Y6 &= A2 \cdot A1 \cdot A0' \\
 Y7 &= A2 \cdot A1 \cdot A0
 \end{aligned}$$

4. Cost: eight 3-input AND gates plus three inverters. Delay: two gate levels.
5. Check A2=1, A1=0, A0=1. Y5 needs A2 true, A1 false, A0 true. All hold, so Y5 = 1. Every other line has at least one term false, so all others are 0.
6. Now a 4-to-1 multiplexer with select bits S1 and S0.

$$OUT = S1' \cdot S0' \cdot D0 + S1' \cdot S0 \cdot D1 + S1 \cdot S0' \cdot D2 + S1 \cdot S0 \cdot D3$$

7. Notice the four product terms are exactly the four decoder outputs. A multiplexer is a decoder whose one-hot lines gate the data through.

8. A 2-to-1 multiplexer is the smallest useful case: $OUT = S'.A + S.B$. It is the atom of all control logic in a CPU.

■ 5.8.4 PLAIN – what is really happening inside

1. Where each part sits in a real CPU.
2. Multiplexers sit everywhere. In front of every register write port, choosing which value gets written.
3. In the forwarding paths of a pipeline, choosing whether an operand comes from the register file or straight from a later stage.
4. And at the program counter, choosing between the next sequential address and a branch target.
5. Decoders sit in the instruction decoder, turning an opcode field into control lines, and in the memory system, turning an address into one row select line inside a RAM array.
6. Demultiplexers sit on write paths, steering one result to one of many destination registers.
7. Priority encoders sit in the interrupt controller, choosing which of many simultaneous interrupt requests to serve first.
8. They also sit in a floating-point unit, finding the position of the highest set bit so a result can be normalized.
9. Comparators sit in branch units, in cache tag matching, and in address range checks in a memory protection unit.
10. A barrel shifter sits inside the ALU, so that shift and rotate instructions take one cycle instead of one cycle per bit position.
11. Parity generators sit on memory buses and on cache arrays, producing one extra check bit per byte or per word.
12. The honest version: a modern core has no part labelled “priority encoder”. It has a synthesized cloud of gates produced from a description of that behaviour. The names are how engineers think, not how silicon is drawn.

■ 5.8.5 TECHNICAL – the engineer’s version

1. Sizes and costs of the standard blocks:

Block	Inputs to outputs	Delay levels
2-to-1 mux	2 data, 1 select	2
3-to-8 decoder	3 to 8	2
8-to-3 priority encoder	8 to 3 plus valid	3 to 4
32-bit barrel shifter	32 plus 5 select	5

2. A barrel shifter is built as a chain of 2-to-1 multiplexer stages, one per bit of the shift amount. A 32-bit shifter needs five stages, shifting by 16, 8, 4, 2 and 1.
3. That gives logarithmic depth: shifting a 64-bit word by any amount takes six multiplexer stages, not 63 steps.
4. A funnel shifter is a barrel shifter fed by two concatenated words. It implements rotate, and on x86 the SHLD and SHRD double-precision shift instructions.
5. Classic discrete parts, all from the 7400 family:

Part	Function
74138	3-to-8 decoder / demultiplexer

Part	Function
74151	8-to-1 multiplexer
74148	8-to-3 priority encoder
7485	4-bit magnitude comparator

- The 74280 is a 9-bit odd and even parity generator and checker. Parity over n bits is an XOR tree of depth $\log_2 n$, so 64-bit parity is six levels deep.
- x86-64 has a POPCNT instruction that counts set bits, and a parity flag in EFLAGS covering only the low 8 bits of a result. That 8-bit limit is a hangover from the 8086 of 1978.
- An equality comparator is an XNOR per bit followed by a wide AND. A magnitude comparator is usually a subtraction whose sign and zero flags are read, which is why CMP on x86 is SUB with the result discarded.
- One-hot encoding is standard for finite state machines on FPGAs, because flip-flops are plentiful and the next-state logic becomes shallow. On ASICs, binary encoding is usually preferred to save flip-flops. This is a technology-dependent trade-off, not a rule.

■ 5.8.6 WORDS – remember these

- Multiplexer — a chooser — a combinational selector routing one of 2 to the power n data inputs to a single output under n select bits.
- Demultiplexer — a router — the inverse of a multiplexer, steering one input to one of many outputs.
- Decoder — small number in, one hot wire out — an n -to-2-to-the- n one-hot converter.
- Priority encoder — reports the most important active input — an encoder that resolves multiple simultaneous requests by fixed rank, usually with a valid output.
- Barrel shifter — shifts any distance in one step — a logarithmic-depth multiplexer network performing arbitrary shifts and rotates.
- One-hot — exactly one wire is 1 — an encoding using one signal per state or value.
- Parity — odd or even count of 1 bits — the XOR reduction of a word, used as a single-bit error detection code.

5.9 The ALU

■ 5.9.1 PLAIN – in simple words

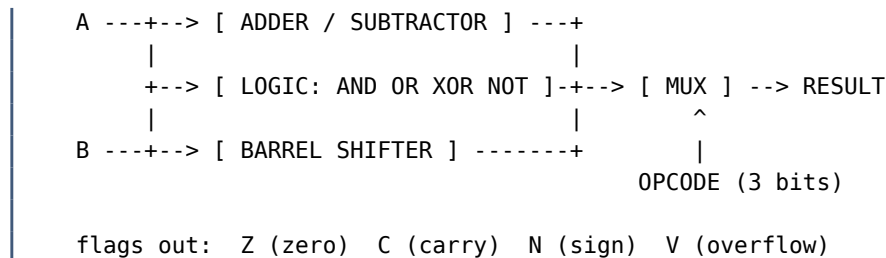
- An ALU is the arithmetic and logic unit. It is the part of a CPU that actually computes things.
- It is simpler than people expect. You build every operation you want, all in parallel, all at once.
- Then you put a multiplexer at the end and use a control code to pick which answer to keep.
- So the adder is always adding. The AND unit is always ANDing. The shifter is always shifting. Only one result is chosen.
- That sounds wasteful, and it is. But it is fast, because no operation waits for any other to finish.
- The control code that selects the operation comes from the instruction being executed. It is called the ALU control, or the opcode.
- Besides the result, the ALU also produces a few single-bit facts about that result. Those are called flags or condition codes.
- The four classic flags are zero, carry, sign and overflow.

■ 5.9.2 PLAIN – a picture in your head

1. Think of a kitchen where four cooks each prepare a different dish from the same two ingredients, all at the same time.
2. When the order ticket arrives it does not say “start cooking”. It says “serve dish number three”.
3. All four dishes were already made. Three are thrown away.
4. Serving is instant because the work was done speculatively, in parallel.
5. Where this comparison breaks: throwing away three meals costs real food, and in an ALU the unused operations cost real energy every cycle.
6. That is why modern designs add clock gating and operand isolation, which hold the inputs of unused units steady so those gates never switch and never burn power.
7. So the picture is right about the timing and slightly wrong about the cost.

■ 5.9.3 PLAIN – a worked example

1. Here is the shape of a small ALU.



2. A control table for an 8-operation ALU with a 3-bit opcode.

Opcode	Name	Result
000	ADD	A + B
001	SUB	A - B
010	AND	A AND B
011	OR	A OR B
100	XOR	A XOR B
101	NOT	NOT A
110	SHL	A shifted left by B
111	SHR	A shifted right by B

3. Note that opcode bit 0 doubles as the SUB control wire from section 5.7, feeding the eight XOR gates and the adder carry-in. One bit, two jobs.
4. Now the four flags, which are computed from the result and from the adder.

Flag	Set when
Z (zero)	every bit of the result is 0
C (carry)	carry out of the top adder stage
N (sign)	the top bit of the result is 1
V (overflow)	carry into top differs from out

5. Z is a wide NOR gate over all result bits. For 8 bits it is one NOR of eight inputs, or a tree of smaller NORs.
6. N is a wire. It is literally bit 7 of the result, with no gate at all.

7. C comes straight from the adder's carry-out.
8. V is one XOR gate: $V = C(8) \text{ XOR } C(7)$.
9. Worked case. Opcode 001, $A = 45$, $B = 67$, as in section 5.7. Result is 1110 1010.
10. Z is 0, because the result is not all zeros. N is 1, because bit 7 is 1. C is 0, meaning a borrow occurred on x86 convention. V is 0, because minus 22 fits fine in 8 bits.

■ 5.9.4 PLAIN – what is really happening inside

1. Not every instruction updates every flag, and the rules are precise. Getting them wrong is a classic source of bugs.
2. On x86-64, arithmetic instructions write the full set. Logic instructions write some and force others to zero.
3. Move instructions write none at all, which is why you can load a value between a compare and a conditional jump without destroying the comparison.

x86-64 instruction	Flags it writes
ADD, SUB, CMP, NEG	CF ZF SF OF AF PF
AND, OR, XOR, TEST	ZF SF PF; CF and OF forced 0
INC, DEC	ZF SF OF AF PF; CF untouched
MOV, LEA, PUSH, POP	none

4. INC and DEC deliberately leave the carry flag alone so that they can be used inside a multi-word addition loop without destroying the running carry.
5. SHL and SHR put the last bit shifted out into the carry flag, and set OF only when the shift count is exactly 1.
6. A shift by zero on x86 leaves all flags completely unchanged, which is a genuine special case in the specification and a well-known trap.
7. The order of events in one ALU operation: operands arrive on the A and B buses, every unit computes, the multiplexer selects, the flags are derived from that result, and the destination register captures it at the next edge.
8. Nothing is sequenced inside that. It is one settling of combinational logic.

■ 5.9.5 TECHNICAL – the engineer's version

1. The 74181 was introduced by Texas Instruments in February 1970 and was the first complete ALU on a single chip.
2. It is a 4-bit slice performing 16 arithmetic and 16 logic functions, selected by four function lines S0 to S3, a mode line M and a carry-in Cn.
3. Standard versions completed an operation in 22 nanoseconds. The 74S181 did it in 11 nanoseconds and the 74F181 in 7 nanoseconds.
4. It was used in the Data General Nova 1200, the Digital Equipment PDP-11 series, the Xerox Alto and the VAX-11/780.
5. Four 74181 slices plus one 74182 lookahead carry generator gave a 16-bit ALU. Sixteen 74S181 slices plus five 74S182 parts gave 64 bits.
6. Modern cores have several ALUs. Intel Skylake, released in August 2015, has four integer execution ports, p0, p1, p5 and p6, each able to retire a simple ALU operation every cycle.
7. That is why the reciprocal throughput of ADD on Skylake is 0.25 cycles: four independent adds can issue per cycle even though each takes one cycle of latency.
8. Condition-code architectures differ sharply. x86-64 and AArch64 have a flags register. RISC-V has none. MIPS has none for integers.

9. Flags are expensive in an out-of-order core, because the flags register is a single hot dependency that every arithmetic instruction writes. Designs cope by renaming the flags register and by splitting it into independently renamed groups.
10. The x86 partial-flag problem is real. INC writes some flags and leaves CF, so a later instruction reading all flags may need to merge two sources. Intel added a merging micro-operation for it, and compilers prefer ADD 1.
11. To observe flags: in GDB use “info registers eflags”; on AArch64 read the NZCV field of CPSR or PSTATE. In a disassembly, look for SETcc, CMOVcc and Jcc instructions, which are the consumers.

■ 5.9.6 WORDS - remember these

1. ALU — the part that computes — the arithmetic and logic unit, a combinational block selecting among parallel functional results.
2. Opcode — the code that says which operation — the instruction field driving the ALU function select multiplexer.
3. Flag — a one-bit fact about the result — a condition code recording zero, carry, sign, overflow or parity.
4. Zero flag — the answer was all zeros — the NOR reduction of the result bus.
5. Condition code register — where the flags live — EFLAGS or RFLAGS on x86, NZCV in PSTATE on AArch64.
6. Operand isolation — stop unused units switching — holding inputs of unselected functional units constant to save dynamic power.
7. Bit slice — one chip handling four bits — a design style where identical narrow ALU chips are cascaded to build a wider machine.

5.10 Multiply and divide in hardware

■ 5.10.1 PLAIN - in simple words

1. There is no such thing as a multiply gate. Multiplication has to be built out of adding and shifting.
2. The school method works in binary and is much easier there, because every digit of the multiplier is either 0 or 1.
3. For each 1 bit in the multiplier, you add a shifted copy of the multiplicand. For each 0 bit, you add nothing.
4. So multiplying two 32-bit numbers means adding up to 32 shifted copies.
5. Doing that one step at a time takes 32 clock cycles. That is the slow way, called shift-and-add.
6. The fast way is to build all 32 shifted copies at once, in wires, and then add them all together with a tree of adders.
7. Division is worse. You cannot know a quotient digit until you have tested it.
8. So division is inherently a guess-and-check loop, and that is why it is the slowest common instruction on every processor.

■ 5.10.2 PLAIN - a picture in your head

1. Multiplying by hand on paper is writing several rows and then adding the column totals.
2. A hardware array multiplier is that same sheet of paper, made of wire, with every row present at once.
3. Adding the rows is the slow part. Doing it one row at a time is like adding a long column top to bottom.
4. A Wallace tree instead pairs rows off in a knockout tournament: 32 rows become 22, then 15, then 10, and so on down to 2.

- Because the tournament halves the field each round, the number of rounds grows like the logarithm of the number of rows.
- Where this comparison breaks: in a knockout tournament the losers are eliminated. In a Wallace tree nothing is lost. Three rows are merged into two rows with the same total, using carry-save adders.
- Nothing is thrown away. The value is preserved exactly at every level.

■ 5.10.3 PLAIN – a worked example

- Multiply 13 by 11 in binary. 13 is 1101 and 11 is 1011.

	1 1 0 1	(13)
x	1 0 1 1	(11)

	1 1 0 1	bit 0 of 11 is 1, shift 0
	1 1 0 1	bit 1 of 11 is 1, shift 1
	0 0 0 0	bit 2 of 11 is 0, shift 2
	1 1 0 1	bit 3 of 11 is 1, shift 3

	1 0 0 0 1 1 1 1	= 143

- And 13 times 11 is 143. Correct.
- Note the product of two 4-bit numbers needs 8 bits. In general n bits times n bits needs $2n$ bits, which is why x86 MUL writes its result across two registers.
- Restoring division of 13 by 3, using 4-bit values. The rule each step is: shift the remainder left, bring down the next bit, try subtracting the divisor.
- If the subtraction goes negative, undo it and write a 0 quotient bit. If it does not, keep it and write a 1.

step	rem	bring	trial rem-3	keep?	quotient bit
1	0	1	1-3 = -2	no	0
2	1	1	3-3 = 0	yes	1
3	0	0	0-3 = -3	no	0
4	0	1	1-3 = -2	no	0

- Quotient 0100 is 4, remainder 1. And 13 divided by 3 is 4 remainder 1. Correct.
- The word restoring means undoing the subtraction whenever it went negative. That undo is a whole extra add, which is pure waste.
- Non-restoring division avoids the undo by allowing negative partial remainders and correcting at the end. It costs one add-or-subtract per bit instead of up to two.

■ 5.10.4 PLAIN – what is really happening inside

- Booth's algorithm speeds up multiplication by noticing that a long run of 1s can be handled with one subtraction and one addition.
- Multiplying by 0111 1111, which is 127, is the same as multiplying by 128 and then subtracting 1 copy. Two operations instead of seven.
- Modified Booth encoding, also called radix-4 Booth, looks at three multiplier bits at a time and emits one partial product per two bits.
- That halves the number of rows to add before the tree even starts. It is used in essentially every production multiplier.
- The adder tree uses carry-save adders. A carry-save adder is just a full adder with its carry not connected onward, so it takes three numbers in and produces two numbers out, with no carry rippling at all.

6. Only the very last step, turning the final two numbers into one, needs a real carry-propagating adder. That is where a fast prefix adder is used.
7. Division has no such trick, because the quotient bits are not known in advance. Each digit depends on the remainder produced by the previous digit.
8. SRT division, named after Sweeney, Robertson and Tocher, speeds this up by guessing several quotient bits at a time from a small lookup table and tolerating a slightly wrong guess.
9. That lookup table is where Intel's famous 1994 bug lived.

■ 5.10.5 TECHNICAL – the engineer's version

1. Andrew Donald Booth devised his signed multiplication technique in 1950 at Birkbeck College, London, while working on crystallography. It appeared as "A Signed Binary Multiplication Technique" in the Quarterly Journal of Mechanics and Applied Mathematics, volume 4, part 2.
2. Chris Wallace published "A suggestion for a fast multiplier" in the IEEE Transactions on Electronic Computers in February 1964. A Wallace tree has depth of order $\log n$ and places multiplication in complexity class NC1.
3. Luigi Dadda proposed the Dadda multiplier in 1965. It reduces later and uses fewer full adders than a Wallace tree for the same delay, at the cost of a slightly wider final adder.
4. An n by n array multiplier has delay of order n and area of order n squared. A Wallace or Dadda tree has delay of order $\log n$ and similar area.
5. Measured Intel Skylake figures, from the uops.info instruction database. Latency is in clock cycles; reciprocal throughput is cycles between independent issues, so smaller is better.

Instruction	Latency	Recip. throughput
ADD r64, r64	1	0.25
IMUL r64, r64	3	1.00
DIV r32	25 to 28	6.00
DIV r64	35 to 90	about 21

6. Floating-point division on the same core: DIVSS, single precision, latency up to 11 with reciprocal throughput 3.00; DIVSD, double precision, latency up to 13 with reciprocal throughput 4.00.
7. The legacy x87 FDIV on Skylake is approximately 14 to 16 cycles, per Agner Fog's instruction tables. Treat that as approximate; it is not listed in the same form by uops.info. Compilers targeting x86-64 emit DIVSD instead.
8. Integer division latency is data-dependent, which is why DIV r64 is a range and not a number. The hardware exits early when the operands are small. ADD and IMUL are fixed-latency.
9. Division has improved sharply in recent parts:

Microarchitecture	DIV r64 latency	Recip. tput
Skylake, 2015	35 to 90	about 21
Meteor Lake-P, 2023	14	10.0
AMD Zen 4, 2022	9 to 17	7.00
AMD Zen 5, 2024	10 to 18	7.00

10. This is why compilers replace division by a constant with a multiply by a magic reciprocal and a shift. Look for that pattern in any optimized build: a division by 10 becomes an IMUL and an SHR.

11. The Pentium FDIV bug: Thomas R. Nicely of Lynchburg College noticed inconsistencies on 13 June 1994, confirmed them on 19 October 1994 and reported them to Intel on 24 October 1994.
12. The cause was five cells missing from an SRT quotient-digit lookup table that should have held 1,066 populated entries. Intel announced full replacement on 20 December 1994, and took a pre-tax charge of 475 million US dollars.
13. The lesson engineers took from it: division hardware is intricate, and verifying it is now done with formal methods rather than test vectors alone.

■ 5.10.6 WORDS - remember these

1. Shift-and-add — the school method in binary — iterative multiplication adding a shifted multiplicand per set multiplier bit.
2. Partial product — one shifted row before adding — the term contributed by one multiplier bit or Booth group.
3. Booth encoding — handle runs of 1s in one step — a signed-digit recoding of the multiplier that halves the partial product count at radix 4.
4. Carry-save adder — three numbers in, two out — a full adder array with carries kept as a separate vector rather than propagated.
5. Wallace tree — a knockout tournament of adders — a logarithmic-depth partial product reduction network.
6. Restoring division — undo the subtraction if it went negative — a one-bit per cycle division algorithm with a correction step.
7. SRT division — guess several quotient bits from a table — the radix-4 or higher division method named after Sweeney, Robertson and Tocher.

5.11 Propagation delay and the critical path

■ 5.11.1 PLAIN - in simple words

1. A gate does not answer instantly. It takes a small, fixed amount of time to change its output after its inputs change.
2. That time is called propagation delay, or gate delay.
3. If you wire three gates in a line, the delays add up. Three gates in series take three gate delays.
4. A circuit has many paths from its inputs to its outputs. Each path has its own total delay.
5. The longest of all those paths is called the critical path.
6. The whole circuit is not finished until the critical path is finished. The fast paths finished long ago and are just waiting.
7. So the critical path sets the maximum clock speed. Nothing else does.
8. To make a circuit faster you must shorten its longest path. Speeding up anything else changes nothing at all.

■ 5.11.2 PLAIN – a picture in your head

1. Think of a group hike where everyone must reach the hut before dinner is served.
2. Dinner starts when the last walker arrives, not when the first does.
3. Making the fastest walker faster does nothing. Dinner is still at the same time.
4. Only helping the slowest walker moves dinner earlier.
5. That slowest walker is the critical path, and finding them is the whole job of a timing analysis tool.
6. Once you speed that walker up, someone else becomes the slowest, and you repeat.
7. Where this comparison breaks: walkers are independent, but circuit paths share gates. Enlarging a gate to speed up one path slows another, because a bigger gate is a bigger load on whatever feeds it.
8. That coupling is why timing closure is an iterative struggle and not a single calculation.

■ 5.11.3 PLAIN – a worked example

1. We will compute the maximum clock frequency of the 8-bit ripple-carry adder from section 5.6, using assumed but realistic delay figures.
2. Assumptions, which stand in for a slow discrete logic family:

Element	Delay
XOR gate	3.0 ns
AND gate	2.0 ns
OR gate	2.0 ns
Register clock-to-output	1.0 ns

3. The register setup time is 0.5 ns. Clock skew is assumed to be zero.
4. Recall the full adder equations: $SUM = A \oplus B \oplus C_{in}$, and $C_{out} = (A \text{ AND } B) \text{ OR } (C_{in} \text{ AND } (A \oplus B))$.
5. Delay from carry-in to carry-out of one stage: AND then OR, so 2.0 plus 2.0 equals 4.0 ns.
6. Delay from A or B to carry-out of the first stage: XOR then AND then OR, so 3.0 plus 2.0 plus 2.0 equals 7.0 ns.
7. Delay from carry-in to that stage's sum bit: one XOR, so 3.0 ns.
8. Now trace the critical path across all eight bits.

A0/B0 -> C1	7.0 ns
C1 -> C2 -> ... -> C8	7 hops x 4.0 = 28.0 ns
total A0/B0 -> Cout	35.0 ns
alternative ending:	
A0/B0 -> C7	7.0 + 6 x 4.0 = 31.0 ns
C7 -> S7 (one XOR)	3.0 ns
total A0/B0 -> S7	34.0 ns

9. The critical path is the longer of the two, 35.0 ns.
10. Now the clock period. It must fit three things: the register's own delay, the logic, and the setup time of the capturing register.

$T_{min} = \text{clock-to-output} + \text{critical path} + \text{setup}$
$T_{min} = 1.0 + 35.0 + 0.5 = 36.5 \text{ ns}$
$f_{max} = 1 / 36.5 \text{ ns} = 27.4 \text{ MHz}$

11. So this adder cannot be clocked faster than about 27 MHz, no matter how good the rest of the design is.
12. Now replace it with two 4-bit carry-lookahead blocks. Generating P and G takes one XOR, 3.0 ns. Each block's carry-out is then one AND plus one OR, 4.0 ns.
13. Carry out of the first block arrives at 3.0 plus 4.0 equals 7.0 ns. Carry into bit 7 arrives 4.0 ns later, at 11.0 ns. The final sum XOR adds 3.0 ns, giving 14.0 ns. Allow 15.0 ns with margin.

$$\begin{aligned} T_{\min} &= 1.0 + 15.0 + 0.5 = 16.5 \text{ ns} \\ f_{\max} &= 1 / 16.5 \text{ ns} = 60.6 \text{ MHz} \end{aligned}$$

14. The same arithmetic, more than twice as fast, purely by shortening the longest path.

■ 5.11.4 PLAIN - what is really happening inside

1. Setup time is how long before the clock edge the data must already be stable at the register input.
2. Hold time is how long after the clock edge the data must stay stable.
3. Setup violations are fixed by slowing the clock down. Hold violations are not. A hold violation means a path is too fast, and slowing the clock does not help at all.
4. Hold violations are fixed by inserting delay, usually pairs of buffers, into the offending path.
5. If either is violated, the register can enter a state that is neither 0 nor 1 for an unpredictable time. That is called metastability.
6. Chapter 6 covers clocks, registers, setup, hold and metastability properly. Here you only need the one idea that the longest path sets the speed.
7. One more honesty point. Gate delay is not a single number. It depends on temperature, on supply voltage, and on random manufacturing variation.
8. So every real timing figure comes as a set of corners: slow-slow at high temperature and low voltage, fast-fast at low temperature and high voltage, and typical in between.
9. A design must pass setup checks in the slow corner and hold checks in the fast corner. Both, always.

■ 5.11.5 TECHNICAL - the engineer's version

1. Propagation delay is measured between 50 percent supply crossings, from input transition to output transition. Rise and fall delays differ and are listed separately as tPLH and tPHL.
2. Real 2-input NAND propagation delays across the 7400 family, showing sixty years of progress:

Family	Introduced	Typical NAND delay
7400 standard TTL	1964	about 10 ns
74LS00	1971	about 9 ns
74HC00	1983	about 7 ns at 5 V
74LVC00	1993	about 3.5 ns at 3.3 V

3. The 74HC00 datasheet limit is 15 ns maximum at 6 volts with a 50 picofarad load. Datasheet maxima are guaranteed numbers; typicals are not.
4. Inside a modern chip, gate delays are in picoseconds, not nanoseconds. A fan-out-of-four inverter delay is roughly 5 to 15 picoseconds at leading process nodes, which is why cores run at several gigahertz.
5. The setup-time equation used by every static timing analysis tool is:

$$T_{\text{clk}} \geq t_{\text{cq}} + t_{\text{logic\_max}} + t_{\text{setup}} + t_{\text{skew\_uncertainty}}$$

6. And the hold-time check, which does not involve the clock period at all:

| $t_{cq} + t_{logic\_min} \geq t_{hold} + t_{skew}$

7. Static timing analysis, or STA, checks every path in a design against these two inequalities without simulating any vectors. Tools: Synopsys PrimeTime, Cadence Tempus, and the open-source OpenSTA.
8. Slack is the margin on a path: required arrival time minus actual arrival time. Negative slack is a violation. Worst negative slack, WNS, is the headline number a designer watches.
9. Logical effort, the design method published by Ivan Sutherland, Bob Sproull and David Harris in their 1999 book “Logical Effort”, gives a systematic way to size gates along a path for minimum delay.
10. Process, voltage and temperature corners are usually enumerated as SS, TT and FF for slow, typical and fast silicon, combined with voltage and temperature extremes. A modern signoff may check dozens of corner and mode combinations.

■ 5.11.6 WORDS – remember these

1. Propagation delay — how long a gate takes to answer — the 50 percent to 50 percent input-to-output transition time.
2. Critical path — the slowest route through the logic — the timing path with the largest delay, which bounds the clock period.
3. Slack — how much margin a path has — required arrival time minus actual arrival time, negative meaning a violation.
4. Setup time — data must be ready before the edge — the minimum stable time at a register input preceding the active clock edge.
5. Hold time — data must stay put after the edge — the minimum stable time following the active clock edge.
6. Static timing analysis — checking speed without simulating — exhaustive path-based timing verification against setup and hold constraints.
7. PVT corner — the worst case combination — a specified process, voltage and temperature condition at which timing must be met.

5.12 Why floating point maths is not exact

■ 5.12.1 PLAIN – in simple words

1. Type 0.1 plus 0.2 into almost any programming language and you get 0.30000000000000004, not 0.3.
2. This is not a bug in the language. It is not a bug in the processor. It is the direct consequence of building numbers out of a fixed number of bits.
3. In base ten, one third cannot be written exactly. It is 0.3333 forever, and you must stop somewhere.
4. In base two, one tenth cannot be written exactly either. It is 0.0001100110011 with 0011 repeating forever.
5. The hardware has room for 53 significant bits. It stores the closest value it can and throws the rest away.
6. So the value stored for 0.1 is not 0.1. It is very slightly more than 0.1.
7. The same is true for 0.2. Add the two slightly wrong values and the errors do not cancel. They add up.
8. The result is slightly more than 0.3, and it is a different bit pattern from the closest value to 0.3. So the comparison fails.

9. Numbers like 0.5, 0.25 and 0.75 are exact, because they are sums of powers of two. Only fractions whose denominator is a power of two are exact.

■ 5.12.2 PLAIN – a picture in your head

1. Imagine a ruler marked only in halves, quarters, eighths, sixteenths and so on, down to a very fine division.
2. Every mark on that ruler is a number the computer can hold exactly.
3. Now try to measure one tenth of the ruler's length. There is no mark there. There never will be, however fine you make the divisions.
4. So you pick the nearest mark. That is what rounding to 53 bits does.
5. Measure a tenth twice and add the two nearest marks. The two small errors both pointed the same way, so the total is a little further off.
6. Where this comparison breaks: the marks on a real ruler are evenly spaced. Floating-point marks are not.
7. Near zero the marks are extremely close together. Near very large values they are far apart. The spacing doubles every time the exponent increases by one.
8. That is what floating means. The point moves so that you always get about 16 significant decimal digits, whether the number is tiny or huge.

■ 5.12.3 PLAIN – a worked example

1. Here are the exact values a 64-bit double actually holds. These are not approximations of the stored values. They are the stored values, written out in full decimal.

```
stored for 0.1:
0.1000000000000000055511151231257827021181583404541015625

stored for 0.2:
0.2000000000000000011102230246251565404236316680908203125

exact sum of those two:
0.30000000000000000444089209850062616169452667236328125

nearest double to 0.3:
0.2999999999999999988897769753748434595763683319091796875
```

2. The sum lands on a different mark from the one 0.3 lands on. That is why the equality test is false.
3. Printed with the shortest string that round-trips, that sum shows as 0.30000000000000004. The extra digits are not noise. They are the true value.

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 == 0.3
False
>>> from decimal import Decimal
>>> Decimal(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

4. Now the same idea in hardware terms. A double has one sign bit, 11 exponent bits and 52 stored fraction bits.
5. The 53rd significant bit is implied. For normal numbers there is always a leading 1 that does not need storing, which buys one free bit of precision.
6. 0.1 in binary is 1.100110011001100... times 2 to the power minus 4, with 1001 repeating.

7. The hardware keeps 53 bits of that and rounds the rest. Rounding up here, which is why the stored value is slightly too big.

■ 5.12.4 PLAIN – what is really happening inside

1. Watch what the adder actually does when it is handed the two stored values.
2. Step 1, align. The two numbers have different exponents, minus 4 and minus 3. The smaller one is shifted right by one bit so the binary points line up.
3. That shift can push bits off the right-hand end straight away. They are not lost yet, because the hardware keeps a few extra bits for exactly this reason.
4. Step 2, add. The two 53-bit fractions go into an ordinary integer adder, the same kind built in section 5.6. There is nothing special about it.
5. Step 3, normalize. Shift the result so the leading bit is a 1 again, and adjust the exponent to match. That shift uses a barrel shifter and a priority encoder to find the leading 1.
6. Step 4, round. The result is now wider than 53 bits, so it must be cut down. The default rule is round to nearest, ties to even.
7. That rounding step is where the error is created. It is a deliberate, specified, deterministic operation, not an accident.
8. The three extra bits the hardware carries through the alignment are called guard, round and sticky. The sticky bit records whether anything non-zero fell off the end.
9. Without those three bits, results would be measurably worse. With them, the result is provably the correctly rounded value of the exact sum.
10. The honest version: the hardware is not sloppy. Every basic floating-point operation is required to give the exact result rounded once. The error you see is the minimum possible error for the format.

■ 5.12.5 TECHNICAL – the engineer's version

1. The format is IEEE 754 binary64, universally called double. It was first standardized as IEEE 754-1985, revised as IEEE 754-2008 and again as IEEE 754-2019.

Field	Bits	Meaning
Sign	1	0 positive, 1 negative
Exponent	11	biased by 1023
Fraction	52	plus 1 implied bit

2. Precision is 53 bits, which is about 15.95 decimal digits. Machine epsilon for binary64 is 2 to the power minus 52, roughly 2.22 times 10 to the power minus 16.
3. The bit patterns for the worked example, as 64-bit hexadecimal:

0.1	=	3FB999999999999A
0.2	=	3FC999999999999A
0.3	=	3FD3333333333333
0.1 + 0.2	=	3FD3333333333334

4. The two results differ in the final hexadecimal digit only. That is a difference of one unit in the last place, written 1 ulp.
5. IEEE 754 requires that add, subtract, multiply, divide and square root be correctly rounded: the result must equal the exact mathematical result rounded once to the destination format.
6. Five rounding modes are defined. The default is roundTiesToEven. The others are roundTiesToAway, roundTowardPositive, roundTowardNegative and roundTowardZero.

7. Correct rounding is not required for transcendental functions such as sine and exponential. Those are library code, and different libraries give slightly different answers. That is permitted by the standard.
8. Floating-point addition is commutative but not associative. Changing the grouping changes the result, which is why compilers may not reorder floating-point sums unless you pass `-ffast-math` and accept the consequences.
9. Practical rules for engineers: never test floating-point values with equality for computed results. Compare against a tolerance, or scale to integers, or use a decimal type.
10. For money, use integers of the smallest unit, or a decimal library. Python's decimal module, Java's `BigDecimal` and the SQL `NUMERIC` type all exist for this reason.
11. Fused multiply-add, or FMA, computes $a \times b + c$ with a single rounding at the end instead of two. It was added to x86 as FMA3 with Intel Haswell in 2013 and is standard on AArch64.
12. Tools to observe this: `printf` with `"%.20f"` in C, Python's `decimal.Decimal` constructor applied to a float, and the online notion of shortest round-trip printing implemented by `repr` in Python 3 and by `Double.toString` in Java.

■ 5.12.6 WORDS – remember these

1. Floating point — the point moves to keep precision — a sign, exponent and significand representation of the form $s \times m \times 2^e$.
2. Significand — the digits of the number — the 53-bit mantissa of binary64, with one bit implied for normal values.
3. Rounding — cutting to the bits you have — mapping an exact result to the nearest representable value under a specified rule.
4. ULP — one step on the ruler — unit in the last place, the gap between adjacent representable numbers at a given magnitude.
5. Machine epsilon — the smallest relative step — 2 to the power minus 52 for binary64, about 2.22×10^{-16} .
6. Guard, round and sticky — the three spare bits — extra precision retained during alignment so the final rounding is correct.
7. IEEE 754 — the rule book — the standard defining binary and decimal floating-point formats, operations and rounding, current edition 2019.

5.98 Common wrong ideas

1. Wrong: a gate passes the input current through to the output. Right: a gate reads its inputs and drives a fresh output from its own power supply, which is why signals do not fade along a chain.
2. Wrong: AND and OR are the basic gates. Right: in CMOS, NAND and NOR are the cheap four-transistor primitives, and AND and OR are each one of those plus an inverter, costing six transistors.
3. Wrong: XOR is just another basic gate like AND. Right: XOR has no simple series and parallel transistor form, so it costs 8 to 12 transistors and is noticeably slower, which matters because adders are mostly XOR.
4. Wrong: a computer contains a subtractor circuit. Right: it contains an adder, plus one XOR gate per bit and one control wire, and subtraction is addition of the two's complement.
5. Wrong: the carry flag and the overflow flag mean the same thing. Right: carry means the unsigned result did not fit, overflow means the signed result did not fit, and either can be set without the other.

6. Wrong: making any gate faster makes the circuit faster. Right: only the critical path matters, so speeding up a gate that is not on the longest path changes the maximum clock frequency by exactly nothing.
7. Wrong: a Karnaugh map gives the fastest circuit. Right: it gives a small two-level expression, but a shallower circuit with more gates is often faster, and synthesis tools optimize for timing, not gate count.
8. Wrong: 0.1 plus 0.2 not equalling 0.3 is a rounding bug in the language. Right: it is the specified, correct behaviour of IEEE 754 binary64, because one tenth has no exact binary representation at any finite width.
9. Wrong: division is slow because it is complicated to write. Right: it is slow because each quotient digit depends on the remainder from the previous one, so it cannot be fully parallelized the way multiplication can.
10. Wrong: universality means NAND is the best gate for every job. Right: universality says NAND is sufficient, not optimal, and real designs pick whichever library cell is cheapest for each specific function.

5.99 Chapter summary in 20 lines

1. Boolean algebra has two values, 1 and 0, and three operations, AND, OR and NOT, published by George Boole in 1847 and 1854.
2. Claude Shannon's MIT master's thesis, submitted on 10 August 1937, showed that relay switching circuits obey exactly those rules.
3. Inside a chip, 1 and 0 are two voltage bands, and the assignment of high to 1 is a convention called positive logic.
4. A CMOS gate has a PMOS pull-up network and a complementary NMOS pull-down network, and exactly one of them conducts at a time.
5. An inverter costs 2 transistors, a 2-input NAND or NOR costs 4, and an AND or OR costs 6 because it is a NAND or NOR plus an inverter.
6. Series transistors give AND-like behaviour and parallel transistors give OR-like behaviour, and CMOS gates are naturally inverting.
7. The seven gates are AND, OR, NOT, NAND, NOR, XOR and XNOR; XOR means exactly one, XNOR means the same, NAND means not both, NOR means neither.
8. NAND alone is functionally complete, needing 1 gate for NOT, 2 for AND, 3 for OR and 4 for XOR; NOR alone is complete too.
9. Fabs favour NAND because its series stack is NMOS, and electron mobility is two to three times higher than hole mobility.
10. Any truth table becomes a sum of products, and a Karnaugh map shrinks it by grouping adjacent 1s in Gray-code order.
11. De Morgan's two laws, provable in four rows each, say that not-both equals either-missing and neither equals both-missing.
12. A half adder is one XOR and one AND; a full adder is two half adders and an OR, giving sum and a majority-vote carry.
13. An 8-bit ripple-carry adder is eight full adders in a chain, and its speed is set by the carry travelling from bit 0 to bit 7.
14. Carry-lookahead computes generate and propagate for every bit at once, turning linear carry delay into logarithmic delay.
15. Subtraction is addition of the two's complement: invert every bit of B and set the adder carry-in to 1, costing eight XOR gates.

16. Carry means the unsigned answer overflowed; signed overflow is the XOR of the carry into and out of the top bit.
17. Multiplexers, decoders, priority encoders, comparators, barrel shifters and parity trees are the standard blocks a CPU is assembled from.
18. An ALU computes every operation in parallel and picks one with a multiplexer driven by the opcode, then derives the Z, C, N and V flags.
19. Multiplication is shift-and-add accelerated by Booth encoding and a Wallace tree; division is a guess-and-check loop and is the slowest common instruction, at 35 to 90 cycles for DIV r64 on Intel Skylake.
20. The critical path sets the clock, and rounding a 53-bit significand is why 0.1 plus 0.2 gives 0.30000000000000004 on every IEEE 754 machine.

How a Machine Remembers — Latches, Clocks and Memory Cells

6.0 What this chapter gives you

1. You will explain why a plain logic gate cannot remember anything.
2. You will draw an SR latch and say what all four input pairs do.
3. You will state the difference between a latch and a flip-flop in one line.
4. You will know setup time and hold time in picoseconds, and why breaking them causes metastability.
5. You will say why metastability can be made rare but never removed.
6. You will explain a clock tree, clock skew, jitter and clock gating.
7. You will build registers, shift registers and counters from flip-flops.
8. You will write a state machine as a table and a diagram, and see that a CPU control unit is one of these.
9. You will know how SRAM, DRAM and flash cells hold a bit, and why they cost such different amounts.
10. You will work out how many address lines a given capacity needs.

6.1 The problem: a gate forgets the moment its input changes

■ 6.1.1 PLAIN – in simple words

1. A logic gate looks at its inputs and drives an output, all the time.
2. Change an input and the output changes a moment later. Always.
3. So a gate has no past. It knows only what is on its wires right now.
4. A machine of gates alone therefore cannot remember.
5. The fix is short to say: wire the output back into the input.
6. Now the circuit tells itself what it was. That loop is **feedback**.
7. A good loop has two comfortable resting positions, one for 0 and one for 1. We call that **bi-stable**.

■ 6.1.2 PLAIN – a picture in your head

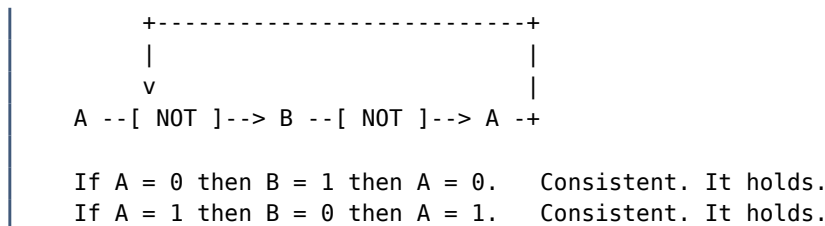
1. Think of a light switch on a wall.
2. Your finger is the input. The switch position is the output.
3. Push it up, take your finger away, and it stays up.
4. A spring inside pushes it further whichever way it has started to move.
5. That spring is feedback. It will not sit halfway. Two positions only.

Where this comparison breaks:

1. A wall switch holds its position with no power. Our circuit needs power every second or it forgets.
2. A real switch can balance on the edge for a moment. So can a real memory circuit, and that matters a great deal (section 6.4).

■ 6.1.3 PLAIN – a worked example

1. Take two NOT gates, each giving the opposite of its input.
2. Wire the first into the second, and the second back into the first.



3. Suppose A is 0. The first gate makes B equal 1, and the second makes A equal 0 again. Nothing moves.
4. Suppose A is 1. Then B is 0, which makes A equal 1. Again nothing moves.
5. Both answers are self-consistent, so that is one stored bit.
6. The bad news: no wire is left to say which bit to store. Section 6.2 fixes that.

■ 6.1.4 PLAIN – what is really happening inside

1. Each NOT gate is a pair of transistors acting as a fast switch pair.
2. Input low connects the output to the supply. Input high connects it to ground.
3. In the loop, one gate always pulls up while the other pulls down.
4. A tiny voltage wobble from a nearby wire is amplified and pushed back round the loop, so the state survives noise.
5. There is a third position, in the middle, where both gates are half on.
6. In theory the loop can balance there. In practice it falls off in well under a nanosecond, because noise always exists.
7. The honest version: while settling, the loop is not digital at all. It is an analogue amplifier with its output tied to its input.

6.1.5 TECHNICAL – the engineer's version

1. Two cross-coupled inverters form a bistable with two stable DC operating points and one metastable point between them.
2. Plotting one inverter's voltage transfer curve against the other's mirrored curve gives the **butterfly plot**, whose curves cross three times.
3. The largest square fitting inside a lobe is the **static noise margin**. For a 6T SRAM cell in a modern process, read noise margin is typically 100 to 200 mV at nominal supply, and collapses at low voltage.
4. Stability needs small-signal loop gain above 1 at the crossing point.
5. Recovery from the metastable point is exponential, with a time constant τ set by the loop gain-bandwidth product. Section 6.4 uses this.
6. The first electronic bistable was the **Eccles-Jordan trigger circuit**, built by William Eccles and Frank Jordan in 1918 from two vacuum triodes, published in the Radio Review in 1919, British patent 148,582.

■ 6.1.6 WORDS – remember these

1. Feedback — sending an output back to an input — a closed signal loop whose gain can exceed unity.
2. Bi-stable — two comfortable resting states — a circuit with two stable DC operating points.
3. Metastable point — the wobbly middle — the unstable equilibrium between the two stable points.
4. Noise margin — how much interference a bit survives — static noise margin in millivolts, from the butterfly plot.
5. Volatile — forgets without power — state held only by current from the supply rail.

6.2 The SR latch from cross-coupled NOR gates

■ 6.2.1 PLAIN – in simple words

1. Our two-inverter loop could hold a bit but we could not write one.
2. A NOR gate outputs 1 only when all inputs are 0. Any input at 1 forces the output to 0.
3. Take two NOR gates and feed each one's output into the other's input.
4. The spare inputs are our controls: S for set, R for reset.
5. The outputs are Q and Q-bar, normally opposites.
6. Raise S and Q becomes 1. Raise R and Q becomes 0.
7. Put both back to 0 and the latch keeps what it had. That is remembering.
8. Raise both at once and it misbehaves. That case is called forbidden.

■ 6.2.2 PLAIN – a picture in your head

1. Two people, each holding the other's arm down. Whoever pushes down wins.
2. S is a referee who taps the first to make them stand. R taps the second.
3. With neither referee touching anyone, the pair stays as it was.
4. If both tap at once, both try to stand and nobody holds anybody down. The arrangement is broken while that lasts.
5. If both let go at the same instant, both lunge, and luck decides.

Where this comparison breaks:

1. People are slow and roughly equal. Gates differ by picoseconds, so the race is settled by manufacturing variation, not fairness.
2. People cannot stand half up for a measurable time. Gates can, and we call that metastability.

■ 6.2.3 PLAIN – a worked example

1. All four cases of the NOR-based SR latch.

S	R	Q next	Name
0	0	no change	Hold
0	1	0	Reset
1	0	1	Set
1	1	both outputs 0	Forbidden

2. Case S=1, R=0. S forces one gate's output to 0. The other then sees two zeroes and outputs 1. Q is 1, Q-bar is 0.
3. Case S=0, R=1 is the mirror image. Q becomes 0.
4. Case S=0, R=0 from Q=1. Each gate sees the other's output holding it, so nothing moves.

5. Case $S=1, R=1$. Both gates have an input at 1, so both outputs go to 0. Q and Q-bar are now both 0, contradicting their names.
6. Drop both to 0 together and both gates try to output 1, then each sees the other's 1 and tries to go back. The result is unpredictable.

■ 6.2.4 PLAIN – what is really happening inside

1. Hold works because each output quietly holds the other gate's input.
2. Set works because a 1 on S overpowers the feedback. A NOR gate with any input at 1 must output 0, whatever the feedback says.
3. The other gate then flips, reinforcing the first. A write is a short fight that the input wins, after which the loop locks in the winner.
4. The forbidden state is not magic. The outputs are simply no longer opposites, so anything reading Q-bar as "not Q" is being lied to.
5. The danger is leaving it. Both gates start identical and race, and the loser is decided by picoseconds.
6. The NAND version flips everything. A NAND outputs 0 only when all inputs are 1, so the controls are active low, written S-bar and R-bar.
7. For the NAND latch, S-bar=0 sets, R-bar=0 resets, both at 1 holds, and both at 0 is forbidden, driving both outputs to 1.

■ 6.2.5 TECHNICAL – the engineer's version

1. The NOR form is active-high on S and R. The NAND form is active-low.
2. The NAND form dominates in practice because a CMOS NAND is smaller and faster than a NOR of equal drive. That is a CMOS fact, not a logic rule.
3. Four NAND gates give the gated SR latch, and one more inverter gives the D latch of section 6.3.
4. The forbidden combination is precisely a **non-complementary output condition**, and the failure on release is a **race condition** decided by device mismatch and edge arrival order.
5. If the release edges are more than one loop delay apart, the last released loses, deterministically. Closer than that, the latch can go metastable.
6. The data sheet for the TTL part SN74LS279, a quad latch with active-low inputs, states it plainly: both inputs low is not a valid state.
7. In Verilog an always @* block with an incomplete if and no else infers a latch by accident. Synthesis warns about it, and the warning is almost always a real bug.

■ 6.2.6 WORDS – remember these

1. SR latch — the simplest two-gate memory — a bistable set-reset element from cross-coupled NOR or NAND gates.
2. Set — store a 1 — drive Q high and let feedback hold it.
3. Hold — keep what is there — both controls inactive, feedback loop closed.
4. Forbidden state — the input pair you must not use — the non-complementary output condition, $S=R=1$ in the NOR form.
5. Race condition — two signals fighting to arrive first — an outcome decided by arrival time, not by logic values.
6. Active-low — does its job at 0 — asserted at logic low, written with a bar or a leading n, as in nRESET.

6.3 The gated D latch

■ 6.3.1 PLAIN – in simple words

1. The SR latch has two data inputs when we want one, and accepts changes at any moment.
2. The gated D latch fixes both. One data input, D. One control, enable.
3. When enable is 1, the latch copies D straight through to Q, following every move D makes.
4. When enable is 0, the latch stops looking and holds what it had.
5. Enable is a window. Open, data flows. Closed, the last value is trapped.
6. There is no forbidden input, because S and R now come from one D and can never both be 1.
7. The awkward part is the open window. While enabled the latch is see-through. That is **transparency**.

■ 6.3.2 PLAIN – a picture in your head

1. A photocopier with a page sliding underneath it.
2. Enable is the shutter. While open, the copy keeps being redrawn as the page moves.
3. Close the shutter and the copy freezes at whatever was underneath then.
4. For a clean copy the page must be still while the shutter is open and for a moment after it closes.

Where this comparison breaks:

1. A smeared photocopy is obviously wrong. A latch that catches a value mid-change gives an ordinary-looking 0 or 1 that happens to be the wrong one. Nothing looks broken.

■ 6.3.3 PLAIN – a worked example

1. A gated D latch over time. Time runs down the table.

Time	EN	D	Q after
t1	0	1	0, held
t2	1	1	1
t3	1	0	0
t4	1	1	1
t5	0	0	1, held
t6	0	1	1, held

2. At t1 enable is 0, so D is ignored and the old 0 stays.
3. From t2 to t4 enable is 1, and three changes of D give three changes of Q.
4. At t5 enable falls while D is 0, but Q keeps the value it had at the instant enable fell, which was 1.
5. Rows t2 to t4 are the problem. Q moved three times inside one enable pulse.

■ 6.3.4 PLAIN – what is really happening inside

1. Build it from the NAND SR latch, which sets on $\overline{S}=0$ and resets on $\overline{R}=0$.
2. Put two steering gates in front. Feed D into one and NOT D into the other, and feed enable into both.
3. Enable at 0 makes both steering gates output 1, which is the NAND latch's hold command. The latch is sealed.
4. Enable at 1 sends exactly one steering gate to 0, chosen by D. So $D=1$ sets and $D=0$ resets.
5. D and NOT D can never both be 1, so the forbidden case is unreachable. That is the real reason the D latch is safe.

- Chain two transparent latches on the same enable and data can run through both in one pulse. Your two-stage pipeline silently becomes one stage.
- The honest version: people say a latch “stores on the falling edge”. It does not. It stores continuously while enabled and merely stops at that edge.

■ 6.3.5 TECHNICAL - the engineer’s version

- The gated D latch is **level-sensitive**. Q is a function of D throughout the active level of the enable.
- Its timing has setup and hold measured against the closing edge, plus D to Q delay while transparent and enable to Q delay at the opening edge.
- In CMOS it is usually not gates but a transmission-gate latch: one pass gate into the storage node, one feedback inverter, one feedback pass gate.
- That costs roughly 8 to 12 transistors against about 20 to 24 for an edge-triggered flip-flop, so latches are smaller and faster.
- Latch-based pipelines allow **time borrowing**: a slow path can run past the nominal boundary into the next stage’s transparent window. Flip-flop pipelines cannot.
- The cost is much harder static timing analysis, because tools must model transparency, borrowing limits and level-sensitive loops.
- Standard cell libraries name these cells with D and L, as in DLH and DLL, meaning transparent when the enable is high or low.
- Where experts disagree: whether latch-based design is worth it. Custom CPU teams say yes, for the area and the borrowing. Most ASIC and FPGA teams say no, because verification cost outweighs the gain.

■ 6.3.6 WORDS - remember these

- D latch — one data wire, one open-or-shut wire — a level-sensitive bistable whose output follows D while enabled.
- Enable — the open-or-shut control — the level-sensitive gating signal, named EN, G or CLK.
- Transparent — see-through — a live combinational path from D to Q during the active level.
- Level-triggered — cares about the level, not the change — output is a function of D throughout the enable window.
- Time borrowing — slow work spilling into the next stage — using transparency to absorb path delay past the nominal boundary.

6.4 The D flip-flop: edge-triggered memory

■ 6.4.1 PLAIN - in simple words

- A latch is see-through while enabled, which is dangerous in a big machine.
- We want something that takes one sample at one instant and then stops looking, however long the control stays high.
- That is the **D flip-flop**, and it is edge-triggered.
- Edge-triggered means it acts on the change of the clock, not on its level.
- A rising-edge flip-flop copies D to Q the moment the clock goes 0 to 1, then ignores D until the next rising edge.
- A flip-flop takes a photograph. A latch runs a video.
- This makes it safe to wire one flip-flop’s output into the next one’s input, which is how every pipeline in every processor is built.
- The price is that D must be steady just before the edge and just after it. Those are setup time and hold time.

9. Break them and the flip-flop can hesitate. That is metastability, and it can never be fully removed.

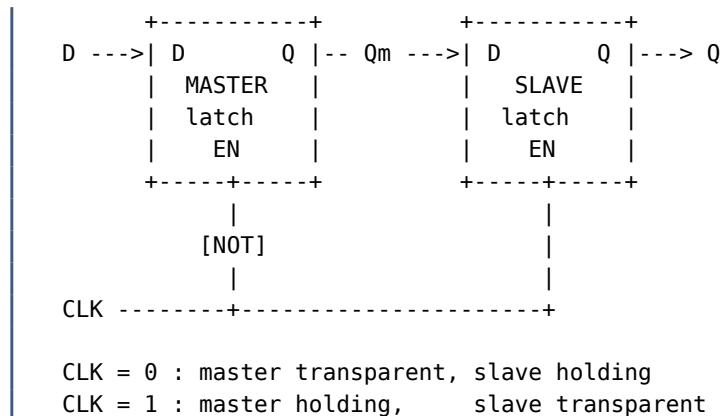
■ 6.4.2 PLAIN – a picture in your head

1. A submarine airlock with two doors and one rule: both are never open.
2. Water enters the outer room while the outer door is open, but goes no further, because the inner door is shut.
3. The outer door shuts, the inner opens, and what is in the middle moves inward. Nothing new can follow it.
4. That is exactly a master-slave flip-flop. The master fills while the clock is low, the slave accepts while the clock is high.

Where this comparison breaks:

1. Airlock doors take a second. A flip-flop's doors close in tens of picoseconds, and any overlap is what makes hold violations possible.
2. Water cannot be half in and half out. A voltage can.

■ 6.4.3 PLAIN – a worked example



1. Clock at 0. The master is transparent so Qm follows D. The slave is shut, so Q does not move.
2. Clock rises. The master seals, trapping the last value of D. The slave opens and passes that trapped value to Q.
3. Change D while the clock is still 1. The master is shut, so nothing happens.
4. Clock falls. The slave seals, keeping Q, and the master opens again to track the new D for the next edge.
5. So Q changes exactly once per cycle, at the rising edge.
6. That is the race edge-triggering solves. With plain latches a value could travel through three stages in one enable pulse, skipping two cycles of work. With flip-flops one edge moves data exactly one stage.

■ 6.4.4 PLAIN – what is really happening inside

1. Setup time is how long D must already be steady before the edge arrives.
2. Hold time is how long D must stay steady after it.
3. Together they define a small forbidden window in which D may not change.
4. The reason is physical. The master is closing a switch and handing over to a feedback loop, and that handover needs a firm push one way.
5. If D moves as the switch closes, the loop starts near the middle, where neither 0 nor 1 is winning.
6. It still recovers, because the middle is unstable, but recovery takes an unpredictable time.

7. Occasionally the next stage samples a half-formed voltage, and two different stages can then read the same wire as 0 and as 1.
8. The chance drops exponentially with the extra settling time you allow, but it never reaches zero.
9. The fix is a **synchronizer**: two or three flip-flops in a row on the same clock, whose only job is to buy the first one extra clock periods.

■ 6.4.5 TECHNICAL – the engineer’s version

1. A positive-edge-triggered D flip-flop samples D in an aperture defined by setup time t_{su} and hold time t_h , and drives Q after clock-to-output delay t_{cq} .

Part or node	t_{su}	t_h	t_{cq}
DM74LS74A, TTL	20 ns	0 ns	25 to 30 ns
Typical 7 nm cell	15 to 30 ps	5 to 20 ps	20 to 40 ps

2. The 74LS74 figures are from the Fairchild and Texas Instruments data sheets at 5 V, 25 degrees Celsius and a 15 pF load. The 7 nm figures are typical orders of magnitude, since vendor libraries are confidential.
3. Static timing analysis checks two inequalities on every path. Setup: $t_{cq} + t_{logic} + t_{skew} \leq T_{clk} - t_{su}$. Hold: $t_{cq} + t_{logic} \geq t_h + t_{skew}$.
4. Setup failures are fixed by slowing the clock. Hold failures are not, and must be fixed by adding delay in silicon.
5. Metastability resolution is exponential in the settling time available:

$$MTBF = \exp(t_r / \tau) / (T_0 * f_{clk} * f_{data})$$

t_r = settling time available after the sampling edge
 τ = resolution time constant of the flip-flop
 T_0 = effective width of the metastable aperture
 f_{clk} = sampling clock frequency
 f_{data} = rate at which the asynchronous input changes

6. Worked example at a deliberately pessimistic slow corner, with $\tau = 50$ ps, $T_0 = 1$ ps, $f_{clk} = 1$ GHz and $f_{data} = 10$ MHz:

Stages	Settling time	MTBF
1 flip-flop	400 ps	about 0.3 s
2 flip-flops	1400 ps	about 4.6 years
3 flip-flops	2400 ps	about 2.2 billion yr

7. One extra flip-flop moved the answer eight orders of magnitude, because the exponential dominates everything else.
8. Two-flop synchronizers are the convention for single-bit control signals crossing clock domains. Three are used at very high frequency or in safety-critical logic.
9. A synchronizer handles one bit. Multi-bit values crossing domains need Gray coding, a handshake or an asynchronous FIFO, because independently synchronized bits can resolve on different cycles.
10. Thomas Chaney and Charles Molnar documented this in their 1973 IEEE Transactions on Computers paper on anomalous synchronizer behaviour, and Leslie Lamport gave the impossibility argument in his 1984 note on Buridan’s principle.

11. Established fact: no purely digital circuit can eliminate metastability. Marketing claim: any product described as metastability-proof. What such products mean is a very long MTBF.
12. Tools that observe this: `report_timing` in Synopsys PrimeTime or Cadence Tempus, and clock-domain-crossing checkers such as SpyGlass CDC.

■ 6.4.6 WORDS – remember these

1. Flip-flop — memory sampling once per clock edge — an edge-triggered bistable, usually master-slave.
2. Setup time — data ready before the edge — minimum stable interval before the clock edge, t_{su} .
3. Hold time — data steady after the edge — minimum stable interval after the clock edge, t_h .
4. Metastability — hesitating between 0 and 1 — an output resting near the unstable equilibrium for an unbounded time.
5. MTBF — average time between rare failures — mean time between failures, exponential in available settling time.
6. Synchronizer — flip-flops that buy settling time — a chain of same-clock registers at a clock domain crossing.

6.5 The clock: why the whole chip marches in step

■ 6.5.1 PLAIN – in simple words

1. A clock is a wire carrying a signal that goes 0, 1, 0, 1 at a steady rate.
2. Drawn on paper it looks like a row of blocks, so we call it a square wave.
3. The moment it goes from 0 to 1 is the **rising edge**, when most flip-flops act.
4. Every flip-flop shares the same clock, so on each edge the whole machine takes one step together.
5. Between edges the gates that compute work out the answers for the next step.
6. The period must be long enough for the slowest of those. That is what limits speed.
7. One billion ticks per second is 1 GHz, so each tick lasts one nanosecond.

■ 6.5.2 PLAIN – a picture in your head

1. A large rowing boat with many rowers and one drum.
2. The drum beats and every rower pulls once. Nobody watches anyone else.
3. The beat cannot be faster than the slowest rower's stroke.
4. If the sound reaches the back later than the front, the margin shrinks. That difference is **clock skew**.
5. If the drummer is slightly irregular, that is **jitter**.
6. If some rowers rest, stop sending them the beat. That is **clock gating**.

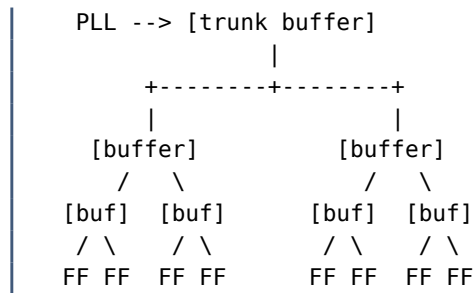
Where this comparison breaks:

1. Rowers hear one drum. On a chip the clock must be copied through thousands of buffers to reach millions of flip-flops, and every copy adds delay and its own irregularity.

■ 6.5.3 PLAIN – a worked example

1. A 3 GHz clock has a period of 0.333 nanoseconds, which is 333 picoseconds.
2. Say clock-to-Q is 30 ps and the next flip-flop needs 25 ps of setup. That leaves 278 ps for all the logic.
3. Allow 20 ps of skew and 10 ps of jitter and the logic has 248 ps.
4. At roughly 15 ps per gate stage, that is about 16 gate delays per cycle.

5. Roughly 15 to 20 gate delays per cycle has been near constant in processor design for two decades, whatever the frequency.



6. The aim of that tree is that every path from trunk to leaf is the same length, so all flip-flops see the edge at the same time.

■ 6.5.4 PLAIN – what is really happening inside

1. The clock starts as a wobbling sine wave from a quartz crystal, typically 24 or 25 MHz on a modern board.
2. A phase-locked loop multiplies it up to the frequency the chip needs and squares it off.
3. That is driven into the trunk and copied outwards by buffers, roughly doubling at each level.
4. Every buffer costs delay and power. The clock network alone commonly burns 20 to 40 per cent of a high-performance chip's dynamic power.
5. Clock gating puts an AND gate in front of a branch and switches off the beat to a block with nothing to do, saving flip-flop and buffer power together.
6. Skew is not always bad. Deliberately delaying one leaf, called **useful skew**, lends time from one pipeline stage to a slower one.
7. The honest version: a real clock edge is not vertical. It rises over tens of picoseconds, and every flip-flop decides somewhere on that slope.

■ 6.5.5 TECHNICAL – the engineer's version

1. Period T is 1 divided by frequency: 1 GHz gives 1000 ps, 3 GHz gives 333 ps, 5 GHz gives 200 ps.
2. Desktop parts now pass 6 GHz. The Intel Core i9-14900KS, launched in March 2024, was specified at 6.2 GHz maximum turbo.
3. Skew is the arrival-time difference between two flip-flops. Common sign-off targets keep global skew under about 5 per cent of the period.
4. Jitter splits into period, cycle-to-cycle and long-term jitter. A few picoseconds RMS is typical for an on-chip PLL.
5. Skew helps or hurts by direction. Positive skew from launch to capture relaxes setup and tightens hold, which is why hold violations often appear only after clock tree synthesis.
6. Distribution styles are the balanced H-tree, the buffered fishbone and the full clock mesh. Grids give the lowest skew and cost the most power, and IBM POWER designs used them.
7. Clock gating is inserted as integrated clock-gating cells, usually inferred by synthesis from an enabled register description. **Power gating** is different: it removes the supply, and loses state without retention cells.
8. Asynchronous design drops the global clock for local handshakes. It works and stays a niche. The Amulet processors at the University of Manchester in the 1990s were asynchronous implementations of the ARM architecture.
9. Tools that observe this: `report_clock_timing -type skew` in PrimeTime, and `cpupower frequency-info` on Linux for the current core clock.

■ 6.5.6 WORDS - remember these

1. Clock — the steady tick everything follows — a periodic square wave whose active edge triggers all synchronous elements.
2. Rising edge — the moment the tick goes up — the low-to-high transition, the sampling instant for a positive-edge flip-flop.
3. Clock skew — the tick arriving at different times — the spatial difference in clock arrival between two sequential elements.
4. Jitter — the tick being slightly irregular — temporal variation of edge position, in RMS or peak-to-peak picoseconds.
5. Clock tree — the branching wiring that copies the clock — the buffered distribution network from source to every leaf register.
6. Clock gating — switching the tick off to idle parts — inserting an enable into the clock path to save dynamic power.

6.6 Registers, shift registers and counters

■ 6.6.1 PLAIN - in simple words

1. One flip-flop holds one bit. For a byte, use eight side by side on the same clock. That group is a **register**.
2. A 32-bit register is 32 flip-flops. A CPU's small set of named working registers is the **register file**.
3. Pass each flip-flop's output to its neighbour instead and you get a **shift register**, whose whole row slides one place each edge.
4. A shift register turns one wire into a row of bits, one bit per tick. That is how a slow single-wire link becomes a parallel byte.
5. A **counter** is a register that adds one to itself on every clock edge.
6. Counters measure time, address memory, count events and divide clocks.

■ 6.6.2 PLAIN - a picture in your head

1. A row of eight boxes on a belt, each holding a black or white marble.
2. A register is the belt stopping and all eight boxes being refilled at once from eight chutes above.
3. A shift register is the belt moving one box right each tick, a new marble dropping in at the left and the rightmost falling off.
4. Feed one marble per tick, wait eight ticks, and a full byte is lined up. That is serial-to-parallel conversion.
5. A counter is a row of odometer wheels, the rightmost turning every tick and nudging its neighbour when it passes nine.

Where this comparison breaks:

1. On an odometer the nudge travels wheel by wheel, taking time. That is a **ripple counter**, the slow kind.
2. A good digital counter computes every digit in parallel and changes them all on one edge. That is a **synchronous counter**.

■ 6.6.3 PLAIN - a worked example

1. A 4-bit synchronous up counter, one row per clock edge.

Tick	Q3 Q2 Q1 Q0	Value
0	0 0 0 0	0
1	0 0 0 1	1
2	0 0 1 0	2
3	0 0 1 1	3
4	0 1 0 0	4
5	0 1 0 1	5
6	0 1 1 0	6
7	0 1 1 1	7
8	1 0 0 0	8
9	1 0 0 1	9
10	1 0 1 0	10
11	1 0 1 1	11
12	1 1 0 0	12
13	1 1 0 1	13
14	1 1 1 0	14
15	1 1 1 1	15
16	0 0 0 0	0, wrapped

2. Q0 changes every tick, so its frequency is half the clock. Q1 changes every second tick, so its frequency is a quarter.
3. A 4-bit counter is therefore also a divide-by-2, 4, 8 and 16 circuit, free.
4. The rule for each bit: bit n flips when all the bits below it are 1.
5. At tick 16 it rolls over. Four bits count only to 15, and wrapping is not an error, it is arithmetic modulo 16.

■ 6.6.4 PLAIN – what is really happening inside

1. In a synchronous counter every flip-flop shares the clock, with a little logic in front computing its next value.
2. For bit 0 that logic is “the opposite of what I am”. For bit 1 it is “flip me if bit 0 is 1”, and so on up the chain.
3. All of those run in parallel during the clock period, and at the edge all bits update together.
4. In a ripple counter only bit 0 sees the real clock. Bit 1 is clocked by bit 0’s output, bit 2 by bit 1’s, and so on.
5. That is cheaper in gates, but the change travels along the chain, so the top bit updates several gate delays after the bottom.
6. Going from 7 to 8, a 4-bit ripple counter can briefly show 6, then 4, then 0. Anything sampling it then reads a lie.
7. A shift register is simpler still. Each flip-flop’s D input is the previous flip-flop’s Q output, with no logic in between.

■ 6.6.5 TECHNICAL – the engineer’s version

1. A register is an n-bit array of edge-triggered flip-flops sharing a clock, and usually a reset and a load enable.
2. A **register file** is a small multi-ported array. A 64-bit RISC core has 32 architectural integer registers, and a 4-wide machine may need 8 read and 4 write ports, which is why register files use custom SRAM-like cells rather than standard flip-flops.

3. Shift registers are classed by interface: SISO, SIPO, PISO and PIPO, for serial or parallel in and out.
4. Real parts: the 74HC595 is an 8-bit serial-in, parallel-out shift register with an output latch, widely used to drive LED arrays from three microcontroller pins. The 74HC165 is its parallel-in, serial-out partner.
5. Serial-to-parallel conversion by shift register is how SPI, JTAG boundary scan and many display drivers work.
6. A **linear feedback shift register** feeds an exclusive-OR of chosen taps back to the input. A maximal-length n-bit LFSR cycles through all $2^n - 1$, non-zero states, and is used for test patterns and for scrambling in Ethernet and PCI Express.
7. Ripple counters cost the fewest gates but glitch and are limited by the chain delay. Synchronous counters cost more logic, never glitch at the outputs, and are limited only by one stage.
8. A synchronous counter in Verilog is four lines:

```
always @(posedge clk) begin
    if (rst)      count <= 4'd0;
    else if (en)  count <= count + 4'd1;
end
```

9. Tools that observe this: `perf stat` on Linux reads hardware performance counters, which are wide synchronous counters inside the CPU, and `rdtsc` on x86 reads the time-stamp counter.

■ 6.6.6 WORDS - remember these

1. Register — a row of flip-flops holding a number — an n-bit array of edge-triggered elements on a common clock.
2. Register file — the CPU's small set of named boxes — a multi-ported array of architectural registers with dedicated ports.
3. Shift register — bits sliding one place per tick — a chain of flip-flops with Q of one feeding D of the next.
4. Ripple counter — digits nudging each other along — an asynchronous counter where each stage clocks the next.
5. Synchronous counter — all digits changing together — next-state logic on a shared clock, so all bits settle on one edge.
6. LFSR — a shift register with a feedback trick that looks random — a linear feedback shift register giving a pseudo-random sequence.

6.7 Finite state machines

■ 6.7.1 PLAIN - in simple words

1. A machine with memory can be in different situations. Each is a **state**.
2. A **finite state machine** has a fixed list of states and rules for moving between them.
3. A rule says: in this state, seeing this input, move to that state on the next clock tick.
4. It also produces outputs, which are the things it makes happen.
5. Three pieces build one: a register holding the state, logic computing the next state, and logic computing the outputs.
6. If outputs depend only on the state, it is a **Moore machine**.
7. If they depend on the state and the current inputs, it is a **Mealy machine**. Moore outputs are steadier, Mealy machines need fewer states.

■ 6.7.2 PLAIN – a picture in your head

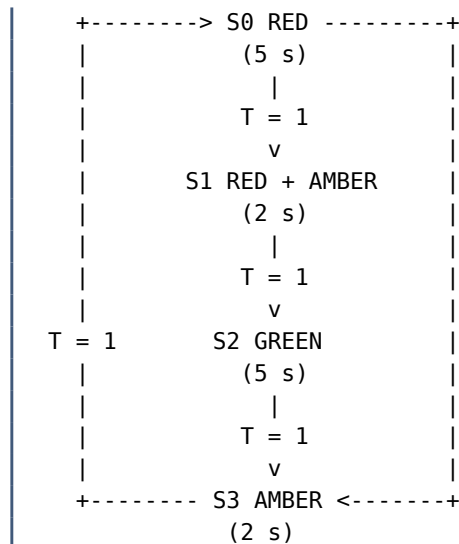
1. A board game where you move one token between numbered squares.
2. The token's square is the state, and there is only one token.
3. The painted arrows are the transitions, each with a condition written on it.
4. The clock is the drum from section 6.5. Between beats you decide, on the beat you move.
5. Squares with an instruction printed on them, like “turn the green lamp on”, give Moore outputs. Arrows with an instruction, like “ring the bell as you pass”, give Mealy outputs.

Where this comparison breaks:

1. In a board game you can pause and think. A real state machine must have its next state fully computed before the next clock edge, every time.

■ 6.7.3 PLAIN – a worked example

1. A traffic light for one direction, with one input T meaning the countdown timer has finished.



2. The state table, with the lamps written R, A and G.

State	Input T	Next state	Lamps on
S0 RED	0	S0	R
S0 RED	1	S1	R
S1 RED+AMBER	0	S1	R, A
S1 RED+AMBER	1	S2	R, A
S2 GREEN	0	S2	G
S2 GREEN	1	S3	G
S3 AMBER	0	S3	A
S3 AMBER	1	S0	A

3. This is a Moore machine. In the last column the lamps depend only on the state, never on T.
4. Four states need two bits, so the state register is two flip-flops, with S0 = 00, S1 = 01, S2 = 10 and S3 = 11.
5. To make it Mealy, add an output firing only during a move, such as a pulse reloading the timer. That pulse depends on the state and on T being 1.


```

always @(posedge clk)
  if (rst) state <= S0;
  else case (state)
    S0: state <= T ? S1 : S0;
    S1: state <= T ? S2 : S1;
    S2: state <= T ? S3 : S2;
    S3: state <= T ? S0 : S3;
  endcase

```

■ 6.7.4 PLAIN – what is really happening inside

1. On each clock edge the state register captures whatever the next-state logic worked out during the previous period.
2. The next-state logic is pure gates, seeing the current state bits and the inputs and producing the next state bits.
3. The output logic is pure gates too, and in a Moore machine it sees only the state bits.
4. So the loop is: state register drives logic, logic drives state register, clock decides when it advances. Nothing else.
5. A Moore output cannot react to an input in the same cycle. That one-tick delay is the price of its stability.
6. Now the big claim. A CPU's control unit is exactly this structure and nothing more.
7. Its state includes where it is in the instruction cycle. Its inputs include decoded instruction bits, flags and interrupt lines. Its outputs are the control wires that open registers, pick an ALU operation and start a memory access.
8. The only difference from our traffic light is size: four states here, hundreds across dozens of units in an out-of-order core.
9. The honest version: a modern CPU is not one giant state machine. It is many communicating state machines, plus large regular structures such as caches and queues that nobody describes state by state.

■ 6.7.5 TECHNICAL – the engineer's version

1. Formally an FSM is a state set, an input alphabet, an output alphabet, a transition function, an output function and an initial state.
2. Any Mealy machine converts to a Moore machine, usually at the cost of more states.
3. Encoding matters in silicon. Binary uses the fewest flip-flops, 3 bits for 8 states. One-hot uses 8, but gives the simplest next-state logic. Gray uses 3 and changes only one bit per step.
4. One-hot is the convention in FPGA design, where flip-flops are plentiful and wide logic is expensive. Binary is more common in ASICs, where flip-flop area is the cost.
5. Gray encoding is used where state bits cross a clock domain, because only one bit changes per transition, so only one bit can be sampled ambiguously. That is standard inside asynchronous FIFOs.
6. Illegal states must be handled. Safety-related designs use a full default branch forcing a return to a known state, so a single-event upset cannot wedge the machine.
7. Maurice Wilkes proposed **microprogramming** in 1951, holding the control unit's transitions in a small read-only memory rather than in fixed gates. IBM used it heavily in System/360 from 1964, which is how one architecture ran on very different hardware.
8. Modern x86 processors still contain microcode, which is why microcode updates can change processor behaviour after manufacture. That is how the 2018 Spectre and Meltdown mitigations reached shipped chips.

- Tools that observe this: `yosys -p 'fsm_detect; fsm_map'` reports inferred state machines, and commercial synthesis lists the encodings it chose.

■ 6.7.6 WORDS – remember these

- State — the situation the machine is in — one value held in the state register, from a finite set.
- Transition — the move from one situation to another — the next-state function evaluated at a clock edge.
- Moore machine — outputs depend on where you are — output is a function of the current state alone.
- Mealy machine — outputs depend on where you are and what you see — output is a function of state and inputs.
- One-hot — one flip-flop per state, only one set — an encoding of Hamming weight 1, favoured in FPGAs.
- Microcode — the control unit's own tiny program — a stored microprogram in ROM or writable control store sequencing control signals.

6.8 SRAM: the six-transistor cell

■ 6.8.1 PLAIN – in simple words

- SRAM** means static random access memory. Static means it holds its value while power is on, with no help.
- One cell is the two-inverter loop of section 6.1 plus two switches that let us reach inside.
- Each inverter is two transistors, and the two switches make six. Hence the name, the six-transistor cell.
- Both switches are opened by one wire, the **word line**.
- Outside are two wires, the **bit lines**, one carrying the value and one its opposite.
- To read, open the word line and let the cell tip the bit lines slightly apart. An amplifier notices which way.
- To write, drive the bit lines hard and open the word line. The drivers overpower the loop and flip it.
- SRAM is fast because the answer is already sitting there, and expensive because six transistors per bit is a lot of silicon.

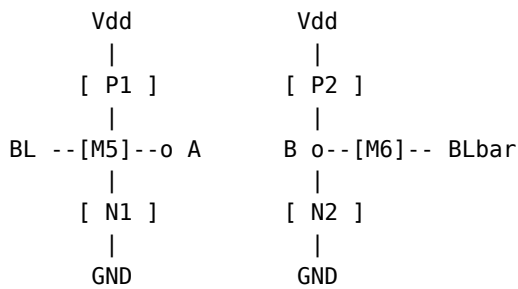
■ 6.8.2 PLAIN – a picture in your head

- A see-saw with a child sitting firmly on one end.
- Reading is feeling which end is down, without changing it.
- Writing is an adult pushing the high end down until it settles the other way.
- It never needs topping up. While gravity works, it stays.
- But a see-saw takes a lot of space, and only a few fit in the playground.

Where this comparison breaks:

- Gravity is free. An SRAM cell's gravity is a small current from the supply running for ever, and across a large cache that leakage is a real part of a chip's idle power.

■ 6.8.3 PLAIN – a worked example



WL, the word line, drives the gates of M5 and M6.
 Node A drives the gates of P2 and N2.
 Node B drives the gates of P1 and N1.
 P1 with N1 is one inverter. P2 with N2 is the other.

1. To store a 1, node A is high and node B is low. P1 holds A up, N2 holds B down, and each keeps the other in place.
2. To read, first charge both bit lines to the supply and let go. That is precharging.
3. Raise the word line. B is low, so the cell pulls BLbar down through M6 and N2, while BL stays high.
4. After a short time the bit lines differ by about 50 to 100 millivolts, and a sense amplifier drives that difference out at full logic level.
5. To write a 0, drive BL to ground and BLbar to the supply, then raise the word line. The external drivers beat the cell and node A is dragged down.
6. Sizing is the whole art. M5 and M6 must be weak enough not to disturb the cell on a read, yet the write drivers must still beat P1 and P2.

■ 6.8.4 PLAIN – what is really happening inside

1. A cache is a rectangular array of millions of these cells, sharing word lines along rows and bit lines down columns.
2. Only one word line is raised at a time, which selects one row.
3. Every column in that row dumps its value onto its own bit line pair at once, so a whole row is read in parallel, and column select logic then picks the bits you asked for.
4. SRAM is fast because the sequence is short: decode, raise a word line, let the bit lines separate a little, amplify, output.
5. There is no waiting for a capacitor to fill and no restoring afterwards, because reading did not destroy anything.
6. The honest version: reading does disturb the cell slightly. The read current lifts the low node, and in a weak cell or at low supply that can flip the bit. This is read upset, and it is why SRAM fails at low voltage before ordinary logic does.

■ 6.8.5 TECHNICAL – the engineer's version

1. The standard cell is 6T. An 8T cell adds a separate read port to remove read disturb, and 10T cells are used at very low voltage.
2. Published high-density 6T bitcell areas:

Process	Bitcell area	Volume from
TSMC N5	0.021 sq um	2020
TSMC N3E	0.021 sq um	Q4 2023

Process	Bitcell area	Volume from
TSMC N2	0.0175 sq um	H2 2025

3. Sources quote TSMC N5 as either 0.021 or 0.0199 square micrometres depending on the cell variant. The fact to keep is that SRAM barely shrank from 5 nm to 3 nm and shrank again at 2 nm.
4. Cost estimate, since SRAM is never sold by the gigabyte. A gigabyte is about 8.6 thousand million bits, so at 0.021 square micrometres per bit that is roughly 180 square millimetres of bitcell, or about 300 once decoders, sense amplifiers and error correction are added.
5. A 300 mm wafer holds about 70 700 square millimetres, and leading-edge wafer prices are publicly estimated at 16 000 to 20 000 US dollars. That puts SRAM near 100 to 200 US dollars per gigabyte, as an order of magnitude only.
6. Typical latencies in a current core, roughly 1 ns, 4 ns and 12 ns at 4 GHz:

Level	Technology	Typical latency
L1 data cache	SRAM	4 to 5 cycles
L2 cache	SRAM	12 to 20 cycles
L3 cache	SRAM	40 to 60 cycles

7. The Intel 3101, released in 1969, was Intel's first product: a 64-bit bipolar SRAM. The Intel 1101 of the same year was a 256-bit MOS SRAM.
8. SRAM leakage is dominated by subthreshold and gate leakage and scales badly, so designs use multiple threshold voltages and array power gating.
9. Tools that observe this: `lscpu` and `getconf -a | grep CACHE` print cache sizes, and `perf stat -e cache-misses` counts misses against them.

■ 6.8.6 WORDS - remember these

1. SRAM — memory that holds itself up while powered — static RAM built from cross-coupled inverter cells.
2. 6T cell — the six-transistor bit — two inverters plus two access transistors sharing one word line.
3. Word line — the wire that selects a row — the row line driving the gates of the access transistors.
4. Bit line — the wire carrying the value in and out — one of a complementary pair, precharged before each read.
5. Sense amplifier — hears a whisper and shouts it — a differential amplifier resolving a small bit line split into full logic.
6. Read upset — a read accidentally changing the bit — a cell flip caused by read current lifting the storage node.

6.9 DRAM: one transistor and one capacitor

■ 6.9.1 PLAIN - in simple words

1. DRAM means dynamic random access memory. Dynamic means it does not hold itself up and must be topped up constantly.
2. One cell is a single transistor acting as a switch, plus a tiny capacitor, which is a bucket for charge.
3. Charge in the bucket means 1. An empty bucket means 0.
4. Two components instead of six is why DRAM is so much cheaper and denser than SRAM. That is the entire reason it exists.

5. The bucket leaks, and left alone it empties in a fraction of a second.
6. So the chip walks through every row, reads it and writes it straight back. That is **refresh**, and it never stops while power is on.
7. Worse, reading a cell empties the bucket, so every read is automatically followed by a write-back of what was found.

■ 6.9.2 PLAIN – a picture in your head

1. A wall of thousands of small buckets, each with a slow drip.
2. In under a tenth of a second a full bucket becomes ambiguous.
3. A worker walks the wall topping up every bucket that still looks full, and must get round before the first goes bad. That deadline is the refresh interval.
4. Checking a bucket means tipping it into a measuring tray, so you must then pour it back.
5. The tray starts half full, so a bucket tips it slightly up or down. That tiny difference is what the sense amplifier measures.

Where this comparison breaks:

1. Water can be at any level. A cell that has half leaked is not half a bit, it is a bit the sense amplifier will read wrongly. No partial credit.

■ 6.9.3 PLAIN – a worked example

1. A modern DRAM cell capacitor is around 6 to 10 femtofarads and the supply is about 1.1 volts.
2. Charge equals capacitance times voltage, so 10 femtofarads times 1.1 volts is 11 femtocoulombs.
3. Dividing by the charge on one electron gives roughly 69 000 electrons for a stored 1. That is the whole physical difference between 1 and 0 in your main memory.
4. Now the refresh arithmetic for DDR4. The standard refresh window is 64 milliseconds up to 85 degrees Celsius.
5. A DDR4 device needs 8192 refresh commands to cover everything, so one command must go out on average every 64 divided by 8192 milliseconds, which is 7.8 microseconds. That figure is tREFI.
6. Above 85 degrees the window halves to 32 milliseconds and tREFI halves to 3.9 microseconds.
7. DDR5 tightened this. Its normal window is 32 milliseconds up to 85 degrees, giving a tREFI of 3.9 microseconds, and 16 milliseconds from 85 to 95.
8. So the familiar answer of 64 milliseconds is right for DDR through DDR4 and out of date for DDR5. Both figures are in the JEDEC standards.

■ 6.9.4 PLAIN – what is really happening inside

1. Cells sit in a grid. A row is selected by a word line, a column is read out on a bit line.
2. Precharge. Every bit line is driven to exactly halfway between supply and ground, then left floating.
3. Activate. The chosen word line rises, connecting every cell in that row to its bit line.
4. The cell's tiny charge tips the much larger bit line by tens of millivolts, and the cell is now drained. That is the destructive read.
5. Sense. The sense amplifier is a cross-coupled pair, exactly like section 6.1, which latches onto whichever side is higher and drives the pair fully apart.
6. That same act refills the cell through the still-open word line, so the write-back is automatic and free.
7. The whole row now sits in the sense amplifiers, together called the **row buffer**, typically 1 or 2 kilobytes.

8. Column select. Only now do we pick the bytes we asked for and send them off the chip.
9. Another request in the same row skips straight to that last step. That is a **row buffer hit**, and it is much faster. A different row means precharging and activating again.
10. Because the address arrives in two halves, the control signals are called **RAS**, row address strobe, and **CAS**, column address strobe. The names are from the 1970s and the signals still exist.

■ 6.9.5 TECHNICAL – the engineer’s version

1. Robert Dennard invented the one-transistor, one-capacitor cell at IBM’s Thomas J. Watson Research Center in 1966, filed in 1967, and was granted United States patent 3,387,286 in 1968.
2. The Intel 1103, launched in October 1970, was a 1 kilobit PMOS DRAM and the first commercially successful one. It ended magnetic core memory.
3. Address multiplexing latches the row address on the falling edge of RAS and the column address on the falling edge of CAS. This halved the pin count, and is why the terms survive in modern timing parameters.
4. The four numbers on any memory module label:

Symbol	Meaning	Typical DDR5
CL	CAS to data out	40 to 46 cycles
tRCD	activate to CAS	39 cycles
tRP	precharge time	39 cycles
tRAS	row active minimum	76 to 84 cycles

5. A row buffer hit costs CL alone. A row conflict costs tRP plus tRCD plus CL. At DDR5-6400 those are about 12.5 ns and about 37 ns, before queueing.
6. Above the cell: cells form subarrays, subarrays form **banks**, banks form **bank groups**, chips accessed together form a **rank**, and a controller port is a **channel**. DDR5 splits each DIMM into two 32-bit sub-channels.
7. Banks let the controller keep several rows open and overlap activation with data transfer, which is where most real bandwidth comes from.
8. The DDR family, with JEDEC publication years:

Generation	Standard year	Voltage	Data rate MT/s
SDR SDRAM	1993	3.3 V	66 to 133
DDR, JESD79	2000	2.5 V	200 to 400
DDR2, JESD79-2	2003	1.8 V	400 to 1066
DDR3, JESD79-3	2007	1.5 V	800 to 2133
DDR4, JESD79-4	2012	1.2 V	1600 to 3200
DDR5, JESD79-5	2020	1.1 V	3200 to 8800

9. DDR4 parts reached the market in 2014, two years after the standard. DDR5 was published on 14 July 2020, and JEDEC raised its ceiling to 8800 MT/s in an update published in April 2024 that also added anti-rowhammer features.
10. DDR6 is in development. Reporting through 2025 and 2026 points to a 2027 timeframe and an 8800 to 17 600 MT/s range. No standard is published yet, so treat that as expectation, not fact.
11. Now the point of the section. DDR-400 at CL3 in 2000 gave a CAS latency of 15 ns and 3.2 GB/s per 64-bit module. DDR5-6400 at CL40 gives 12.5 ns and 51.2 GB/s. Bandwidth rose sixteenfold, latency improved about twenty per cent, in twenty-five years.

12. The reason is physics, not laziness. Bandwidth is bought with more banks, wider buses and faster signalling. Latency is set by charging a long, heavy bit line through one small transistor, and that has not changed.
13. DDR5 added a same-bank refresh command, REFsb, beside the all-bank REFab. For a 16 Gb device Micron quotes tRFC of 295 ns for REFab and 130 ns for REFsb, and reports 6 to 9 per cent throughput gain from using REFsb.
14. Every DDR5 chip carries on-die ECC, 8 parity bits over 128 data bits. That covers internal cell errors and is not the same as module-level ECC.
15. Tools that observe this: `dmidecode -t memory` prints module type, speed and rank, and Intel MLC measures achieved bandwidth and idle latency.

■ 6.9.6 WORDS - remember these

1. DRAM — memory that must be topped up — dynamic RAM using a one-transistor, one-capacitor cell.
2. Refresh — the constant topping up — periodic read and restore of every row inside the retention window, paced by tREFI.
3. Destructive read — looking at it empties it — charge sharing from cell to bit line, requiring an automatic restore.
4. Row buffer — the row currently held open — the sense amplifier array holding one activated row, typically 1 to 2 kilobytes.
5. RAS and CAS — the two halves of an address — row address strobe and column address strobe, from multiplexed addressing.
6. Bank — an independently openable section — a subarray group with its own row buffer, allowing overlapped access.
7. Rank — a set of chips answering together — devices on a module selected by one chip select and sharing the data bus.
8. Channel — one road between controller and memory — an independent memory controller port with its own command and data buses.

6.10 Non-volatile cells: from mask ROM to flash

■ 6.10.1 PLAIN - in simple words

1. Everything so far forgets when power goes. Now we want cells that remember without it.
2. **Mask ROM** has the bits built in during manufacture by the shape of one layer. It can never be changed.
3. **PROM** leaves the factory blank with a tiny fuse at every bit. You blow selected fuses with current, once, and a blown fuse cannot be repaired.
4. **EPROM** stores charge instead of blowing metal. Its package has a quartz window: shine ultraviolet light through it for about twenty minutes and every bit returns to blank.
5. **EEPROM** does the same erase with voltage rather than light, one byte at a time.
6. **Flash** is EEPROM that erases in large blocks instead of by byte. That one restriction made it far smaller and far cheaper.
7. All three of the last ones store a bit as electrons trapped on a small island of conductor with no wires attached to it.

■ 6.10.2 PLAIN - a picture in your head

1. A hollow in the top of a hill, ringed by a high ridge, with no path in.
2. Marbles in the hollow are the trapped electrons, and present or absent is the stored bit.

3. Nothing takes them out, because the ridge is too high. They sit for years.
4. To put marbles in you either fire them hard enough to fly over the ridge, which is hot-electron injection, or tilt the hill steeply so they slip across a thin part, which is tunnelling.
5. To read you notice that the hill's shape is slightly different when the pile is there.
6. Every firing chips rock off the ridge, and after enough thousands of times marbles leak out on their own. That is wear.

Where this comparison breaks:

1. Marbles are in or out. Electrons are quantum objects and tunnelling is a probability, not a rolling motion. The ridge is an energy barrier of roughly 3.2 electron volts between silicon and silicon dioxide.

■ 6.10.3 PLAIN – a worked example

```

      control gate
      =====
      oxide layer
      -----
      floating gate      <- charge lives here
      -----
      tunnel oxide, 8 to 10 nm thick
      =====
      source | channel | drain
      =====
      substrate
  
```

1. A floating-gate transistor is an ordinary transistor with a second gate buried in the insulator, connected to nothing.
2. With no electrons on it, a moderate voltage on the control gate switches the transistor on. Call that erased.
3. Trap electrons there and their negative charge repels the control gate's influence, so the same voltage no longer switches it on. Call that programmed.
4. Reading is applying a fixed voltage and seeing whether current flows. The threshold has shifted, typically by two to four volts in older parts.
5. Control the charge finely and you get several distinct levels, and so more than one bit per cell.

Cell type	Bits per cell	Levels	Typical endurance
SLC	1	2	50k to 100k cycles
MLC	2	4	about 10k cycles
TLC	3	8	1k to 3k cycles
QLC	4	16	a few hundred

■ 6.10.4 PLAIN – what is really happening inside

1. NOR flash programs by **channel hot-electron injection**: a large voltage across the channel accelerates electrons until some jump the barrier.
2. That is fast per byte but draws heavy current, so it cannot be done to many cells at once.
3. NAND flash uses **Fowler-Nordheim tunnelling** for both writing and erasing. A high field across a very thin oxide lets electrons tunnel through.
4. Tunnelling draws almost no current, so thousands of cells can be programmed together. That is exactly why NAND is used for bulk storage and NOR is not.

5. The 15 to 20 volts needed are made on the chip by charge pumps, from an ordinary 3.3 volt supply.
6. Erase pulls electrons back off, and only a whole block at a time, typically several megabytes.
7. That block-erase rule causes nearly everything odd about SSDs. You cannot overwrite in place, so you write elsewhere, remember where, and collect the garbage later.
8. Retention comes from the barrier height, so it is always quoted with a temperature, because heat helps electrons escape.
9. The honest version: most flash sold today is not floating-gate at all. Modern 3D NAND mostly uses **charge trap** cells, where electrons sit in a layer of silicon nitride full of defects rather than on a conductive island.

■ 6.10.5 TECHNICAL - the engineer's version

1. The floating-gate MOSFET was invented by Dawon Kahng and Simon Sze at Bell Labs in 1967.
2. Dov Frohman at Intel turned it into a product. The Intel 1702 of 1971 was the first EPROM, holding 2048 bits as 256 words of 8 bits, erased by ultraviolet light through a quartz window.
3. That erase uses light at 253.7 nm, typically for 20 to 30 minutes. Ordinary glass blocks that wavelength, which is why the window had to be quartz.
4. EEPROM based on Fowler-Nordheim tunnelling was patented by Siemens in 1974. Intel's 2816, a 16 kilobit EEPROM developed by a team including George Perlegos, arrived in 1978.
5. Fujio Masuoka invented flash at Toshiba in 1980. NOR flash was presented in 1984 and NAND flash at the IEEE International Electron Devices Meeting in San Francisco in 1987. Intel shipped the first commercial NOR flash in 1988.
6. NOR gives random byte-level read and is used for boot code executed in place. NAND is read and written a page at a time and erased a block at a time.
7. Since Samsung's V-NAND in 2013, NAND has been built vertically. Layer counts passed 200 in 2022 and current products sit in the 200 to 300 plus range. Stacking, not lithographic shrinking, is now the density driver.
8. Retention is standardized. JEDEC JESD218 requires a client SSD to hold data 1 year at 30 degrees Celsius and an enterprise SSD 3 months at 40, at the end of rated endurance. Fresh cells do far better.
9. Embedded MRAM has replaced embedded flash at 22 nm and below at TSMC, Samsung and GlobalFoundries, because flash cell physics does not scale below about 28 nm.
10. Tools that observe this: `smartctl -a /dev/nvme0` and `nvme smart-log` report NAND wear as percentage used, and `flashrom` reads and writes NOR devices.

■ 6.10.6 WORDS - remember these

1. Mask ROM — bits built in at the factory — contents fixed by a photomask layer during fabrication.
2. PROM — write once, never again — one-time programmable memory using fusible links or antifuses.
3. EPROM — erasable by ultraviolet light — floating-gate memory in a package with a quartz window, erased in bulk.
4. EEPROM — erasable by voltage, byte by byte — electrically erasable floating-gate memory using tunnelling.
5. Flash — EEPROM that erases in big blocks — block-erasable floating-gate or charge-trap memory, NOR or NAND organized.
6. Floating gate — an island of conductor holding electrons — an isolated gate in the oxide that shifts the transistor threshold voltage.
7. Tunnelling — electrons crossing a barrier they should not — Fowler-Nordheim conduction through thin oxide under a high field.

- Endurance — how many rewrites it survives — rated program and erase cycles before the oxide degrades out of specification.

6.11 Memory as a grid: addresses, word lines and bit lines

■ 6.11.1 PLAIN – in simple words

- A memory chip is a rectangle of cells laid out in rows and columns.
- Horizontal wires run across the rows. Each is a word line, and raising one selects that whole row.
- Vertical wires run down the columns. Each is a bit line, carrying one cell's value in or out.
- An **address** is a number saying which cell or group of cells you want.
- The circuit that turns an address into exactly one raised word line is the **address decoder**.
- A decoder with n inputs selects 2^n rows, because n binary digits make that many patterns.
- So with n address lines you reach 2^n locations, and to reach L locations you need n equal to the base-2 logarithm of L , rounded up.

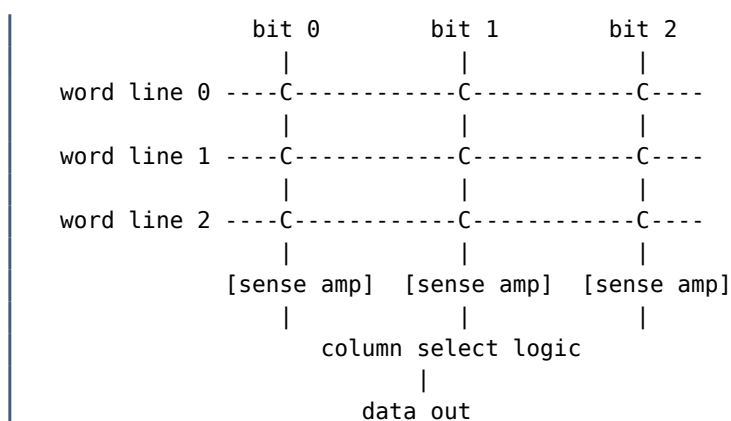
■ 6.11.2 PLAIN – a picture in your head

- A block of post office boxes arranged in a grid on a wall.
- Every box has a number, and that number is the address.
- A clerk takes the number, walks to one row, then along it to one box.
- The decoder is the clerk. Number in, one box selected, nothing else.
- The clerk does not read the letter. Reading is a separate job done by the sense amplifiers on the bit lines.

Where this comparison breaks:

- A clerk walks, so distant boxes take longer. A decoder is a tree of gates and takes the same time for any address. That is exactly what “random access” means: every location costs the same.

■ 6.11.3 PLAIN – a worked example



- Forward. How many locations can 16 address lines reach? Two to the power 16 is 65 536.
- Backward. How many lines does a 1 gibibyte byte-addressable memory need? One gibibyte is 2^{30} bytes, so 30 lines.

Capacity, byte-addressed	Address lines	Locations
1 KiB	10	1 024

Capacity, byte-addressed	Address lines	Locations
64 KiB	16	65 536
4 MiB	22	4 194 304
1 GiB	30	1 073 741 824
16 GiB	34	17 179 869 184

3. Every doubling of capacity costs exactly one more address line.
4. A historical note this explains. The original IBM PC used an Intel 8088 with 20 address lines, reaching 2 to the power 20 bytes, which is 1 MiB. That is where the famous 640 KiB usable limit came from, once the top 384 KiB was reserved.

■ 6.11.4 PLAIN – what is really happening inside

1. A decoder with 20 inputs and a million outputs, built directly, would be absurd. Real designs never do that.
2. First trick, split the address: some bits pick a row, the rest pick a column within the row.
3. A one megabit array becomes 1024 rows by 1024 columns, needing one 10-to-1024 row decoder and one 10-to-1024 column selector.
4. That is also why arrays are roughly square. A square minimizes total wire length, and wire length is delay.
5. Second trick, predecoding: decode in two stages, for instance two 5-to-32 decoders whose outputs are combined in pairs.
6. Third trick, used in DRAM: send row bits and column bits down the same pins at different times. That is the RAS and CAS scheme of section 6.9, halving the address pins.
7. Real chips also carry spare rows and columns. If test finds a bad one, fuses redirect its address to a spare, so almost every memory chip you have used contains repaired defects.

■ 6.11.5 TECHNICAL – the engineer’s version

1. An n-to-2-to-the-n decoder asserts exactly one output per input code. Built flat it needs 2 to the power n gates of n inputs each, so it is always built hierarchically, with predecoding feeding NAND word line drivers.
2. Array organization is quoted as rows by columns by bits per word, plus a column multiplexer ratio. A 4:1 column mux means four bit line pairs share one sense amplifier, trading speed for area.
3. Sub-arrays, called mats, keep word lines and bit lines short. A large cache is dozens of mats with hierarchical decoding. The academic tool CACTI models exactly this trade-off.
4. Cache address terminology is the same grid indexed cleverly: low bits are the byte offset in a line, middle bits are the set index driving the decoder, high bits are the tag that is compared.
5. Redundancy repair uses laser fuses or electrical antifuses set at wafer test. DDR4 and DDR5 also support post-package repair, retiring a failing row in the field.
6. The formulas worth memorizing:

```

locations      = 2 ** address_lines
address_lines  = ceil(log2(locations))
total_bits     = locations * word_width

```

7. Tools that observe this: `lsmem` and `/proc/iomem` show the physical address layout on Linux, and `cpuid` reports the physical address width a processor implements, commonly 46 to 52 bits rather than the full 64.

■ 6.11.6 WORDS – remember these

1. Address — the number naming a location — a binary code decoded to select one row and one column of an array.
2. Address decoder — turns a number into one selected row — an n-to-2-to-the-n selector, built with predecoding.
3. Word line — the horizontal select wire — the row line driven high to connect a row of cells to their bit lines.
4. Bit line — the vertical data wire — the column line carrying charge or current between a cell and a sense amplifier.
5. Random access — every location costs the same — access latency independent of the address, unlike tape or a spinning disk.
6. Redundancy repair — spare rows replacing broken ones — fuse or antifuse remapping of failing addresses to redundant elements.

6.12 Volatile, non-volatile, and memory versus storage

■ 6.12.1 PLAIN – in simple words

1. **Volatile** means the contents vanish when power goes. SRAM and DRAM are volatile.
2. **Non-volatile** means they survive. Flash, ROM and disks are non-volatile.
3. That is a property of the cell, and it is not the same as the difference between memory and storage, though people mix the two up constantly.
4. **Memory** is what the processor reaches directly, one byte at a time, with ordinary load and store instructions.
5. **Storage** is what it reaches by asking a device to move a block, through operating system calls like read and write.
6. The real dividing line is how the processor talks to it, not whether it forgets.
7. Memory is measured in nanoseconds. Storage is microseconds for an SSD and milliseconds for a spinning disk.
8. The line is genuinely blurring, because some devices are non-volatile yet sit on the memory bus and are addressed a byte at a time.

■ 6.12.2 PLAIN – a picture in your head

1. Memory is the desk you work at. Storage is the filing cabinet in the corner.
2. Anything on the desk you touch instantly, without asking anyone.
3. Anything in the cabinet you must fetch, and you fetch a whole folder, not one line of one page.
4. At the end of the day the cleaners clear the desk and leave the cabinet.
5. Persistent memory is a desk the cleaners have been told to leave alone. Still a desk, with desk speed and desk habits, but it survives the night.

Where this comparison breaks:

1. A desk that survives the night sounds purely good. It creates a hard new problem: if power fails halfway through an update you are left with a half-updated permanent structure, and most programs are not written for that.

■ 6.12.3 PLAIN – a worked example

1. Watch what happens when you save a file.
2. Your editor holds the text in DRAM: fast, byte-addressable, gone if power drops.

3. You press save, and the operating system copies the bytes into a page cache, still in DRAM.
4. It then issues a block write to the SSD, whose controller collects it into a page and eventually programs it into NAND cells.
5. Only after that program completes is the data non-volatile.
6. Lose power between steps 3 and 5 and the file is lost, even though the save appeared to succeed. That is why `fsync` exists and why databases care about it so much.
7. With persistent memory on the memory bus the sequence is shorter: a store instruction plus a cache line flush and a fence makes the data durable, with no block layer at all.

■ 6.12.4 PLAIN - what is really happening inside

1. Byte-addressable means the address bus can name any single byte, and load and store instructions work on it directly.
2. Block-addressable means the smallest unit that can move is a sector or page, historically 512 bytes and now usually 4096.
3. That difference is not a detail. It is why file systems, page caches, drivers and interrupt handling exist at all.
4. Non-volatile memory on the memory bus removes that stack. You map the device into your address space and use pointers.
5. The catch is the CPU caches. A store lands in a cache, not in the persistent device, and until it is flushed out it is still volatile.
6. So persistent memory programming needs explicit instructions to push cache lines out and to order those pushes.
7. The honest version: both “persistent memory is just fast storage” and “it is just memory that survives” are wrong. It is a third thing, with the addressing of memory and the durability rules of storage.

■ 6.12.5 TECHNICAL - the engineer's version

1. The three-way comparison, with figures for August 2026:

Property	SRAM	DRAM	NAND flash
Devices per bit	6 transistors	1 tr + 1 cap	1 cell, 1 to 4 bits
Read latency	1 to 10 ns	45 to 90 ns	30 to 100 us
Cost per GB	100 to 200 USD, est	16 to 18 USD	about 0.08 USD
Volatility	volatile	volatile, refreshed	non-volatile

2. Read the cost row carefully. SRAM is never sold by the gigabyte, so that figure is an estimate from silicon area and public wafer price estimates, as worked out in section 6.8.
3. The DRAM figure must be dated. TrendForce reported conventional DRAM contract prices up 90 to 95 per cent in the first quarter of 2026. Retail DDR5 in August 2026 was 16 to 18 US dollars per gigabyte, against 2 to 4 before.
4. The NAND figure is a drive-level price, around 76 US dollars per terabyte for consumer Gen 4 NVMe in mid 2026, including controller, cache and packaging. Raw NAND is cheaper.
5. So the ratios matter more than the absolute numbers. SRAM to DRAM is roughly ten to one, and DRAM to NAND roughly two hundred to one. Those ratios have been broadly stable for two decades, and they are why the memory hierarchy has the shape it does.
6. Persistent memory history, told honestly. Intel and Micron announced 3D XPoint in July 2015. Intel shipped it as Optane DC Persistent Memory in DIMM form from April 2019 and announced the wind-down of the Optane business in July

2022. The product is discontinued. The idea is not.

7. NVDIMM-N remains available: ordinary DRAM plus flash plus a supercapacitor, copying DRAM to flash on power loss. It is byte-addressable at DRAM speed and persistent across power cuts.
8. CXL, Compute Express Link, attaches memory over a PCI Express physical layer. CXL 2.0 arrived in 2020, 3.0 in 2022 and 3.1 in late 2023. Expanders add capacity at 150 to 250 ns rather than a local DIMM's 80 ns.
9. Persistent memory programming uses the x86 CLWB and CLFLUSHOPT instructions with SFENCE, and on Linux the DAX mode of ext4 and xfs with the `ndctl` and `daxctl` tools.
10. Where experts disagree: whether byte-addressable persistence will become mainstream. One camp points to Optane's commercial failure. The other points to CXL, to embedded MRAM shipping in volume, and to the argument that the software problem, not the device, was Optane's real barrier.

■ 6.12.6 WORDS – remember these

1. Volatile — forgets without power — state maintained only while the supply rail is present.
2. Non-volatile — remembers without power — state held by trapped charge, magnetization or physical structure.
3. Memory — what the processor reaches directly — byte-addressable storage on the load-store path, latency in nanoseconds.
4. Storage — what the processor asks a device for — block-addressable media behind a controller and a driver stack.
5. Persistent memory — non-volatile but used like memory — byte-addressable non-volatile media mapped into the address space.
6. CXL — a newer way to hang memory off a processor — Compute Express Link, a cache-coherent protocol layered on PCI Express.

6.98 Common wrong ideas

1. Wrong: a latch and a flip-flop are the same thing. Right: a latch is transparent while its enable is active, a flip-flop samples only at an edge.
2. Wrong: a flip-flop stores whatever is on D while the clock is high. Right: it stores what was on D during the setup window before the edge.
3. Wrong: metastability is a bug a good engineer avoids. Right: it is a physical certainty at any asynchronous boundary, and the only defence is buying settling time to make it astronomically rare.
4. Wrong: two flip-flops in a synchronizer guarantee correctness. Right: they give a long MTBF for one bit. Multi-bit crossings still need Gray coding, a handshake or a FIFO.
5. Wrong: a faster clock always means a faster machine. Right: useful work per cycle is set by logic depth, and roughly 15 to 20 gate delays per cycle has been near constant for two decades.
6. Wrong: DRAM refresh is like topping up a battery. Right: refresh is a full read and rewrite of a row through the same sense amplifiers as a normal read, because reading DRAM destroys the stored charge.
7. Wrong: DRAM has got much faster over the years. Right: bandwidth rose about sixteenfold from DDR-400 to DDR5-6400 while CAS latency improved only from about 15 ns to about 12.5 ns.
8. Wrong: RAM is memory and an SSD is storage because one forgets and one does not. Right: the line is byte-addressable load and store versus block transfer. Persistent memory is non-volatile and is still memory.

9. Wrong: flash stores bits in tiny capacitors like DRAM. Right: flash shifts a transistor's threshold voltage using charge trapped on an isolated gate or in a nitride layer, and needs no power to hold it.
10. Wrong: SRAM is faster because it sits closer to the CPU. Right: it is faster because the value is already held by a driven inverter pair, so there is no charge sharing, sensing or restore step.

6.99 Chapter summary in 20 lines

1. A logic gate has no past. Its output depends only on its inputs right now.
2. Wiring an output back to an input creates feedback, and feedback creates two stable resting states. That is one stored bit.
3. Two cross-coupled inverters are the ancestor of every memory cell, back to the Eccles-Jordan trigger circuit of 1918.
4. Two cross-coupled NOR gates give the SR latch, with set, reset, hold and one forbidden combination.
5. Generating S and R from a single D input removes the forbidden case and gives the gated D latch, transparent while enabled.
6. Transparency is a hazard, because data can race through several stages inside one enable pulse.
7. Two latches on opposite clock phases give a master-slave D flip-flop, which samples once per edge and cannot be raced through.
8. A flip-flop needs D steady for a setup time before the edge and a hold time after it, from tens of nanoseconds in 1970s TTL to tens of picoseconds today.
9. Violating that window can leave the output hovering between 0 and 1. That is metastability, and it can be made rare but never removed.
10. MTBF rises exponentially with settling time, which is why one extra flip-flop in a synchronizer moves the answer by many orders of magnitude.
11. The clock is a square wave whose rising edge makes the whole chip take one step together, distributed through a buffered tree.
12. Clock skew, jitter and the tree's own power are the costs of marching in step, and clock gating is the first way to cut that power.
13. Flip-flops side by side make a register, chained make a shift register, and with next-value logic make a counter.
14. Synchronous counters update every bit on one edge. Ripple counters are cheaper and briefly show values nobody intended.
15. A state register plus next-state logic plus output logic is a finite state machine, Moore if outputs depend on state alone, Mealy if on inputs too.
16. A CPU control unit is exactly that, only larger, and since Wilkes proposed microprogramming in 1951 much of it has lived in microcode.
17. SRAM holds a bit in six transistors as a live inverter loop, so it is fastest and dearest per bit and is used for registers and caches.
18. DRAM holds a bit as about 69 000 electrons on a tiny capacitor. It is dense and cheap, it leaks, it must be refreshed within 64 ms for DDR4 or 32 ms for DDR5, and reading it destroys it.
19. Flash holds a bit as charge trapped on an isolated gate, placed and removed by tunnelling or hot-electron injection, surviving years without power at the cost of limited rewrite endurance.
20. Every memory is a grid of word lines and bit lines behind a decoder, n address lines reach 2 to the power n locations, and memory differs from storage by byte-addressing, not by volatility.



PART B

Turning Bits Into Meaning

How switches become numbers, numbers become text,
pictures on a screen and sound in the air.

Numbers and Meaning — Binary, Text and Unicode

7.0 What this chapter gives you

1. You will be able to explain why computers count in twos, and convert any number to binary and back by two different methods.
2. You will be able to read hexadecimal on sight, and say where you meet it in real work: memory addresses, colour codes, MAC addresses, error codes.
3. You will be able to say what a byte is, why “octet” exists, and what “64-bit CPU” means about registers and address space.
4. You will be able to explain two’s complement, give the exact range of any signed integer size, and name three real disasters caused by overflow.
5. You will be able to encode a decimal fraction into IEEE 754 float32 bits by hand, decode a pattern back, and say why 0.1 plus 0.2 is not 0.3.
6. You will be able to trace text encoding from Morse code through ASCII to Unicode, and write the UTF-8 bytes for an emoji by hand.
7. You will be able to explain endianness, parity, checksums, CRC32 and ECC memory, and say what each one can and cannot catch.
8. You will be able to build a Huffman tree, explain LZ77, and state Shannon entropy in one formula.
9. You will understand the deepest idea here: a pattern of bits has no meaning until something decides how to read it.

7.1 Why base 2

■ 7.1.1 PLAIN – in simple words

1. A computer is made of switches. A switch is either on or off. There is no “half on” a machine can read back reliably.
2. So a computer can store only two marks. Call them 0 and 1.
3. To store bigger numbers, line up many switches in a row.
4. Two switches give four patterns: 00, 01, 10, 11. Three give eight. Each new switch doubles the count.
5. Counting with only two marks is called **binary**, or base 2.
6. Our normal counting uses ten marks, 0 to 9. That is base 10, or decimal, and the only reason for ten is that people have ten fingers.
7. A machine has two voltage levels, so it uses two marks.

■ 7.1.2 PLAIN – a picture in your head

1. Picture a row of eight light switches on a wall.
2. Each has a price tag above it. From the right: 1, 2, 4, 8, 16, 32, 64, 128. Each tag is double the one to its right.
3. To show a number, flip up the switches whose tags add to it. For 5, flip up 4 and 1. That is 00000101. For 156, flip up 128, 16, 8 and 4.
4. Every number from 0 to 255 has exactly one pattern, and no number has two.
5. Reading the wall is just adding the tags of the switches that are up.

Where this comparison breaks

1. Real memory does not hold a switch position. It holds a small charge or a magnetic direction that fades and must be refreshed or rewritten.
2. Real switches are also not exactly on or off. They sit above or below a voltage threshold, and the circuit rounds them to a clean 0 or 1.

■ 7.1.3 PLAIN – a worked example

Counting from 0 to 32 in binary, five bits then six.

Dec	Binary	Dec	Binary
0	00000	17	10001
1	00001	18	10010
2	00010	19	10011
3	00011	20	10100
4	00100	21	10101
5	00101	22	10110
6	00110	23	10111
7	00111	24	11000
8	01000	25	11001
9	01001	26	11010
10	01010	27	11011
11	01011	28	11100
12	01100	29	11101
13	01101	30	11110
14	01110	31	11111
15	01111	32	100000
16	10000		

1. Notice 32 needs a sixth bit. Five bits stop at 31.
2. N bits hold 2 to the power N patterns, from 0 to (2 to the N) minus 1.

The place values in one byte:

Bit position	Place value	Power of 2
7 (leftmost)	128	2 ⁷
6	64	2 ⁶
5	32	2 ⁵
4	16	2 ⁴

Bit position	Place value	Power of 2
3	8	2^3
2	4	2^2
1	2	2^1
0 (rightmost)	1	2^0

Method one, decimal to binary by repeated division. Convert 156.

```

156 / 2 = 78 remainder 0  <- lowest bit
 78 / 2 = 39 remainder 0
 39 / 2 = 19 remainder 1
 19 / 2 = 9 remainder 1
 9 / 2 = 4 remainder 1
 4 / 2 = 2 remainder 0
 2 / 2 = 1 remainder 0
 1 / 2 = 0 remainder 1  <- highest bit

Read the remainders bottom to top: 1 0 0 1 1 1 0 0
156 decimal = 10011100 binary

```

Method two, decimal to binary by subtraction. Same number, 156.

```

Largest place value not bigger than 156 is 128. Use it.
 156 - 128 = 28 bit 7 = 1
Is 64 <= 28? No. bit 6 = 0
Is 32 <= 28? No. bit 5 = 0
Is 16 <= 28? Yes. 28 - 16 = 12 bit 4 = 1
Is 8 <= 12? Yes. 12 - 8 = 4 bit 3 = 1
Is 4 <= 4? Yes. 4 - 4 = 0 bit 2 = 1
Is 2 <= 0? No. bit 1 = 0
Is 1 <= 0? No. bit 0 = 0

Result: 10011100

```

Binary back to decimal is easier still. Take 10011100 and add the place values of the bits that are 1: $128 + 16 + 8 + 4 = 156$.

- Both directions agree, as they must. Division scales better to large numbers; subtraction is easier to do in your head for one byte.

■ 7.1.4 PLAIN - what is really happening inside

- Inside a chip a 1 is a wire held near the supply voltage and a 0 is a wire held near ground. Modern logic cores run at roughly 0.7 to 1.2 volts, so the whole gap is under one volt.
- The circuit never measures the exact voltage. It compares it with a threshold and snaps the answer to a clean 0 or 1.
- That snapping is why binary survives noise. A wire at 0.9 volts and one at 1.05 volts are both read as 1, and both come out identical.
- A base-10 circuit would need ten bands in the same one volt gap, about 0.1 volts each, and noise would ruin it.
- So base 2 is not a preference. It is the widest safety margin one wire can give you.
- The honest version: some devices do store more than two levels. NAND flash stores 3 or 4 bits per cell using 8 or 16 charge levels. It works because a memory cell is read slowly and carefully, not billions of times a second.

■ 7.1.5 TECHNICAL – the engineer’s version

1. Binary is a positional numeral system with radix 2. The digit at position i carries weight 2^i , counted from 0 at the least significant bit. An n -bit unsigned field represents integers in $[0, 2^n - 1]$ inclusive.
2. Repeated division produces the radix-2 expansion least significant digit first. The subtraction method is greedy expansion against descending powers of 2; it is correct because binary place values are super-increasing, each power exceeding the sum of all lower powers.
3. Gottfried Wilhelm Leibniz described binary arithmetic in a 1703 paper for the Paris Academy, long before any machine used it.
4. Claude Shannon’s 1937 MIT master’s thesis, “A Symbolic Analysis of Relay and Switching Circuits”, first tied Boolean algebra to switching hardware.
5. Noise margin is the real engineering reason for radix 2. With two levels the margin is roughly half the supply rail; with ten levels it is one eighteenth, before any process variation is counted.
6. Multi-level storage exists where read latency may be large: NAND flash SLC (1 bit per cell), MLC (2), TLC (3), QLC (4).

Bits	Patterns	Max unsigned value
4	16	15
8	256	255
10	1024	1023
16	65536	65535
32	4294967296	4294967295

7. Observe it with `python3 -c "print(bin(156))"` or `echo 'obase=2; 156' | bc`.

■ 7.1.6 WORDS – remember these

1. Bit — one on-or-off mark — a binary digit, the smallest unit of information, one element of the set $\{0, 1\}$.
2. Binary — counting with only 0 and 1 — base 2 positional notation with place weights 2^i .
3. Base or radix — how many different digits a system uses — the number of distinct symbols per position in a positional numeral system.
4. Place value — the worth of a digit’s slot — the weight r^i applied to the digit at position i in radix r .
5. Most significant bit — the leftmost, biggest-value bit — the bit with the highest positional weight, often written MSB.
6. Least significant bit — the rightmost, worth 1 — the bit of weight 2^0 , often written LSB.

7.2 Hexadecimal and octal

■ 7.2.1 PLAIN – in simple words

1. Binary is correct but painful to read. 11010110 is easy to mistype.
2. Engineers group the bits in fours and give each group one short name.
3. Four bits have sixteen patterns, so we need sixteen digit names: 0 to 9 and then A, B, C, D, E, F. A means ten, F means fifteen.
4. This is **hexadecimal**, usually shortened to hex. So 11010110 becomes 1101 and 0110, which is D and 6, written 0xD6.
5. The 0x in front is just a label saying “the rest is hex”.

- Hex is not a different kind of number. It is a shorter way of writing the same bits.
- There is also **octal**, which groups bits in threes and uses digits 0 to 7. It is mostly historical, but it survives in file permissions on Linux and macOS.

■ 7.2.2 PLAIN – a picture in your head

- Think of a long phone number with no spaces: 919876543210. Hard to read, easy to lose your place, easy to copy wrong.
- Now group it: 91 98765 43210. Same number, far easier to handle.
- Hex does exactly this to bits, but the grouping is fixed at four and each group gets a single character instead of a space.
- One hex digit is always exactly four bits. Two hex digits are always exactly one byte. That rule never bends.

Where this comparison breaks

- Phone number grouping is a display habit and can vary by country.
- Hex grouping is arithmetic. Because 16 is 2 to the fourth power, each hex digit maps to four bits with no carrying between groups. Decimal grouping has no such property, which is why decimal to binary needs real division.

■ 7.2.3 PLAIN – a worked example

The complete conversion table. Learn this and hex becomes automatic.

Hex	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Worked conversion, hex to binary to decimal. Take 0xB4.

```
0xB4 -> B    4
      -> 1011 0100
      -> 10110100
```

Now add the place values:

$128 + 0 + 32 + 16 + 0 + 4 + 0 + 0 = 180$

0xB4 = 10110100 binary = 180 decimal

Worked conversion, decimal to hex. Take 3000.

```
3000 / 16 = 187 remainder 8 -> digit 8
187 / 16 = 11 remainder 11 -> digit B
11 / 16 = 0 remainder 11 -> digit B

Read bottom to top: 0xBB8
Check: 11 x 256 + 11 x 16 + 8 = 2816 + 176 + 8 = 3000
```

Octal, three bits per digit, for Unix permissions.

```
chmod 755 script.sh

7 = 111 = read, write, execute (owner)
5 = 101 = read, no write, execute (group)
5 = 101 = read, no write, execute (everyone else)
```

1. The three bits are, in order, read (4), write (2), execute (1).
2. So 6 is read plus write, 4 is read only, 0 is nothing.

■ 7.2.4 PLAIN – what is really happening inside

1. The machine never stores hex. Memory holds bits and nothing else. Hex exists only where a human reads or types a value.
2. A debugger, a hex editor and an error message all convert bits to hex just before printing, and hex back to bits just after you type.
3. Hex won over decimal for this job because of alignment. Byte, nibble and word boundaries all fall on hex digit boundaries.
4. A 32-bit value is always exactly 8 hex digits and a 64-bit value always exactly 16, so you can count the digits and know the size. In decimal the same value takes 1 to 10 digits, so the width is invisible.
5. Octal survives from machines whose word sizes divided by three, such as the 12-bit PDP-8 and the 36-bit PDP-10, where four bits per digit did not fit.

■ 7.2.5 TECHNICAL – the engineer's version

1. Hexadecimal is radix 16, octal is radix 8. Both are power-of-two radices, so conversion to and from binary is pure regrouping, never division.
2. A 4-bit group is a **nibble** (also spelled nybble). One hex digit is one nibble. Two nibbles form one octet.
3. Notation is language-specific: 0x1F in C, C++, Java, Python, Go, Rust and JavaScript; 1Fh or \$1F in assemblers; 16#1F# in Ada; #x1F in Lisp. Octal is 0o755 in Python 3 and Rust, and a bare leading 0 in C, which is a well-known source of bugs.
4. Where you meet hex in real work:

Context	Example value	What it is
Memory address	0x7fee3b04a20	Pointer on x86-64
Colour	#FF8800	R=255 G=136 B=0
MAC address	00:1A:2B:3C:4D:5E	48-bit NIC address
Windows error	0x80070005	HRESULT, access denied

5. File magic numbers, the first bytes that identify a format:

Format	First bytes (hex)	As text
PNG	89 50 4E 47 0D 0A 1A 0A	.PNG....
PDF	25 50 44 46	%PDF
ELF binary	7F 45 4C 46	.ELF
ZIP / JAR	50 4B 03 04	PK..
Java class	CA FE BA BE	(none)

6. JPEG files start FF D8 FF. GIF files start 47 49 46 38, which reads “GIF8”. The file command on Unix works mainly from a database of these patterns, held in /usr/share/file/magic or similar.
7. In a MAC address the first three octets are the Organizationally Unique Identifier assigned to a vendor by the IEEE, and the last three are chosen by that vendor.
8. Tools that show hex: xxd, hexdump -C, od -t x1, objdump -d, and the x/16xb command inside GDB.

■ 7.2.6 WORDS – remember these

1. Hexadecimal — base 16, digits 0 to F — radix-16 positional notation, four bits per digit, standard for displaying binary data.
2. Nibble — half a byte — a 4-bit group, exactly one hexadecimal digit.
3. Octal — base 8, digits 0 to 7 — radix-8 notation, three bits per digit, used for Unix permission modes and legacy word sizes.
4. Magic number — the fingerprint at the start of a file — a fixed byte sequence at a known offset used to identify a file format.
5. Prefix 0x — a label meaning “hex follows” — a lexical marker for hexadecimal integer literals in C-family languages.

7.3 Bits, bytes and words

■ 7.3.1 PLAIN – in simple words

1. One bit is one 0 or 1, and almost useless alone. Eight bits make one **byte**, which holds 256 different values.
2. A byte is the smallest chunk memory hands out. You cannot ask for “bit number 5” directly; you ask for the byte and pick the bit yourself.
3. A byte is enough for one English letter, or a number from 0 to 255.
4. A **word** is how many bits the processor works on at once. Old processors used 8-bit words, then 16, then 32. Today it is 64.
5. “A 64-bit computer” means the processor handles 64 bits at a time and can point at a huge amount of memory.
6. Networking people say **octet** instead of byte, because “byte” has not always meant exactly eight bits.

■ 7.3.2 PLAIN – a picture in your head

1. Think of a warehouse of identical boxes on numbered shelves. Each box is a byte and each shelf number is a memory address.
2. The forklift is the processor, and its fork width is the word size. A 64-bit forklift picks up eight boxes in one trip; a 32-bit one picks up four.
3. The address space is how many shelf labels the building can print. With 32-bit labels you get about 4.29 billion; with 64-bit labels, 18.4 billion billion.

Where this comparison breaks

1. A real forklift wastes no effort carrying fewer boxes. A real 64-bit CPU moving one byte still uses a full 64-bit path, and pointers get bigger, which uses more cache. Wider is not free.
2. Real memory addresses are not all backed by RAM. Large regions are mapped to devices, or to nothing at all.

■ 7.3.3 PLAIN – a worked example

The bits inside one byte, with their weights.

bit	7	6	5	4	3	2	1	0	
value	128	64	32	16	8	4	2	1	
Byte	1	0	1	1	0	1	1	0	= 0xB6
	128		32	16		4	2		= 182

1. The largest value in one byte is 11111111, which is 255, or 0xFF.
2. Two bytes side by side make 16 bits, holding 0 to 65535.
3. Four bytes make 32 bits, holding 0 to 4,294,967,295.

Why a 32-bit machine stopped near 4 GB:

```

2^32 = 4,294,967,296 distinct addresses
Each address names one byte
So the total space named = 4,294,967,296 bytes
                           = 4 GiB exactly

```

1. That 4 GiB is not only RAM. The graphics card aperture, the BIOS ROM and other device windows are carved out of the same range.
2. That is why 32-bit Windows machines with 4 GB installed typically reported about 3.2 to 3.5 GB usable. The rest of the addresses were spoken for.

■ 7.3.4 PLAIN – what is really happening inside

1. Inside the processor are small named storage slots called registers. On a 64-bit CPU each general register is 64 bits wide, and so is the adder, so one add instruction handles 64 bits in one step.
2. Separately, the address the CPU puts on the memory path decides how much memory it can name. These two widths are usually equal but need not be: some 8-bit chips had 16-bit addresses, some 32-bit chips 36-bit ones.
3. A pointer is a plain integer holding an address, so on a 64-bit system it occupies 8 bytes. Moving to 64-bit doubles every pointer, and programs full of pointers use noticeably more memory afterwards.
4. The honest version: no shipping x86-64 chip wires all 64 address bits. Current designs use 48, giving 256 TiB; newer server parts support 57, giving 128 PiB. The unused top bits must copy the highest used bit, a rule called canonical form.

■ 7.3.5 TECHNICAL – the engineer’s version

1. The term **byte** was coined by Werner Buchholz at IBM in July 1956 during the Stretch project, which shipped as the IBM 7030. The spelling was deliberately changed from “bite” so it could not be misread as “bit”.
2. A byte was not always 8 bits. The CDC 6600 used 60-bit words with 6-bit characters. The PDP-10 used 36-bit words with programmable byte sizes of 6, 7, 8 or 9 bits. Some early machines used 9-bit bytes.
3. Because of that history, IETF and ITU specifications say **octet** when they mean exactly 8 bits. RFC documents use octet throughout for this reason.
4. The 8-bit byte became universal with the IBM System/360, announced 7 April 1964, which fixed 8-bit bytes and 32-bit words.
5. Word size across the generations:

Word	Example chip	Year
8-bit	Intel 8080	1974
16-bit	Intel 8086	1978
32-bit	Intel 80386	1985
64-bit	DEC Alpha 21064	1992
64-bit x86	AMD Opteron	2003

6. Address space arithmetic:

Address bits	Bytes addressable	Common name
16	65,536	64 KiB
32	4,294,967,296	4 GiB
48	281,474,976,710,656	256 TiB
64	1.8446744×10^{19}	16 EiB

7. Physical Address Extension (PAE), introduced with the Intel Pentium Pro in 1995, widened physical addresses to 36 bits, allowing 64 GiB of RAM on a 32-bit machine. Each individual process was still limited to a 4 GiB virtual address space, so it helped servers running many processes and did little for a single large application.
8. x86-64 canonical addressing sign-extends bit 47 into bits 48 to 63. Intel 5-level paging, shipped in Ice Lake server parts from 2019 and supported in Linux since kernel 4.14, extends this to 57 bits.
9. Observe with `getconf LONG_BIT`, `uname -m`, `lscpu` (look for “Address sizes”), and `cat /proc/cpuinfo | grep "address sizes"` on Linux.

■ 7.3.6 WORDS – remember these

1. Byte — eight bits — the smallest individually addressable unit of memory on virtually all modern architectures.
2. Octet — a guaranteed eight bits — the term used in IETF and ITU standards because “byte” was historically machine-dependent.
3. Word — the chunk size a CPU works on naturally — the native register and datapath width of an architecture.
4. Register — a slot inside the processor — the fastest storage in the machine, named rather than addressed, typically 64 bits wide today.

5. Address space — the set of memory locations that can be named — the range of addresses expressible in the architecture's pointer width.
6. Canonical address — a valid pointer shape on x86-64 — an address whose bits 63 to 48 are all copies of bit 47.

7.4 Negative numbers

■ 7.4.1 PLAIN – in simple words

1. A byte holds a pattern of 0s and 1s. Nothing in it says “minus”, so we must agree on a rule reserving some patterns for negatives.
2. The first idea was to use the leftmost bit as a minus sign. That is **sign-magnitude**. It gives two zeros, plus zero and minus zero, and needs special rules for adding mixed signs.
3. The second idea was to flip every bit to negate. That is **one's complement**. It also gives two zeros and needs an extra carry step.
4. The third idea won everything: flip every bit, then add 1. That is **two's complement**. There is exactly one zero, and normal addition just works, because the circuit never needs to know the signs.
5. Every processor you will use today stores signed integers this way.
6. The cost is a lopsided range. In one byte you get -128 to +127, one more negative than positive.

■ 7.4.2 PLAIN – a picture in your head

1. Picture a clock face with 256 marks instead of 12. Going past the top wraps to the start, so 255 plus 1 is 0.
2. Now agree that marks past halfway are read as negatives. The mark we would call 255 we call -1; 254 becomes -2.
3. Under that reading, “add -1” and “add 255” are the same movement. The hand lands in the same place either way.
4. That is the whole trick: subtracting is adding, if you go far enough forward around the dial. The circuit never thinks about signs. It steps around and throws away whatever falls off the top.

Where this comparison breaks

1. A clock hand moving past 12 is harmless. A number wrapping past its top is often a serious bug, because the program keeps running with a value that is wildly wrong rather than stopping.
2. Also, a clock has no “unsigned versus signed” reading. In a computer, the same bits are read one way or the other depending on the declared type, and mixing the two in one expression causes real defects.

■ 7.4.3 PLAIN – a worked example

Making -5 in eight bits, by all three methods.

```

Start with +5:           00000101

SIGN-MAGNITUDE
  set the top bit:       10000101  (reads as "minus five")
  problem: 10000000 is "minus zero", a second zero

ONE'S COMPLEMENT
  flip every bit:        11111010
  problem: 11111111 is "minus zero", a second zero

```

TWO'S COMPLEMENT

```
flip every bit:      11111010
add one:            11111011  <- this is -5
as an unsigned byte that pattern is 251
```

Now check that plain addition works with no special cases.

```
  00000101      (+5)
+ 11111011      (-5)
-----
1 00000000
```

The 1 falls off the left end and is discarded.
What remains is 00000000, which is zero. Correct.

Wraparound at the top of the range.

```
  01111111      (+127, the largest signed byte)
+ 00000001      (+1)
-----
10000000      which as signed is -128, not +128
```

1. Adding 1 to the biggest positive number gives the most negative number.
2. This is **overflow**. The answer did not fit, so it wrapped.

The exact ranges you should know by heart:

Bits	Unsigned max	Signed min	Signed max
8	255	-128	127
16	65,535	-32,768	32,767
32	4,294,967,295	-2,147,483,648	2,147,483,647
64	1.8446744e19	-9.2233720e18	9,223,372,036,854,775,807

1. The 64-bit unsigned maximum written out is 18,446,744,073,709,551,615.
2. The 64-bit signed minimum written out is -9,223,372,036,854,775,808.

■ 7.4.4 PLAIN – what is really happening inside

1. The adder inside a CPU adds two n-bit patterns and produces an n-bit result plus a carry-out bit, the same way every time.
2. To compute A minus B, the CPU negates B and adds. Many designs feed the inverted B into the adder with carry-in set to 1, performing “flip and add one” in a single pass.
3. So one adder does both jobs. That is the real reason two’s complement won: fewer transistors.
4. The CPU sets status flags after each arithmetic instruction, and there are two separate overflow signals:
 1. The carry flag says the result did not fit as an unsigned number.
 2. The overflow flag says it did not fit as a signed number.
5. The processor does not know which reading you intended. It sets both and leaves the program to check. Most programs never check, and the wrong value continues silently.
6. To test whether a pattern is negative, look at the top bit alone. To read the magnitude of a negative, apply the same operation again: flip and add one to 11111011 gives 00000101, which is 5.

■ 7.4.5 TECHNICAL – the engineer’s version

1. Two’s complement represents value v in n bits as the residue $v \bmod 2^n$. Interpretation: the MSB carries weight $-2^{(n-1)}$, all other bits carry the usual positive weights.
2. Range is $[-2^{(n-1)}, 2^{(n-1)} - 1]$. The asymmetry is unavoidable: 2^n patterns cannot be split evenly around zero if zero takes one pattern.
3. Consequence to remember: negating the minimum value overflows. In 32-bit arithmetic, `-INT_MIN` is `INT_MIN`, and `abs(-2147483648)` returns `-2147483648`. This is a genuine source of security bugs.
4. In C and C++, signed integer overflow is **undefined behaviour**. Unsigned overflow is defined and wraps modulo 2^n . Compilers exploit the undefined case for optimization, so an overflowing signed loop can be transformed in ways that surprise you. In Java, signed overflow is defined to wrap. In Rust, it panics in debug builds and wraps in release builds by default.
5. Historical use: the EDSAC (1949) used two’s complement. The IBM System/360 (1964) used it for binary integers and made it the mainstream choice. One’s complement persisted in the CDC 6600 (1964) and UNIVAC 1100 series; sign-magnitude persisted in the IBM 7090 series.

Three real overflow failures.

6. **Year 2038.** Unix time is seconds since 00:00:00 UTC on 1 January 1970, traditionally stored in a signed 32-bit `time_t`. The last representable second is 2,147,483,647, which is 03:14:07 UTC on 19 January 2038. The next increment wraps to -2,147,483,648, which reads as 20:45:52 UTC on 13 December 1901. The fix is a 64-bit `time_t`; Linux has supported 64-bit `time_t` on 32-bit architectures since kernel 5.6 (2020). The bug already bites early: AOLserver crashed in May 2006 on a timeout one billion seconds in the future, and a Microsoft Exchange antimalware update failed in January 2022 when a version number was read as a Unix timestamp.
7. **YouTube, 2 December 2014.** The view counter for the video “Gangnam Style” reached 2,147,483,647, the maximum signed 32-bit integer. Google’s public statement said the company “never thought a video would be watched in numbers greater than a 32-bit integer”. The counter was moved to 64-bit, which raises the ceiling to 9,223,372,036,854,775,807. Note the honest caveat: press coverage described the site as “breaking”, but what the public saw was largely a display Easter egg, and reports differ on exactly when the underlying storage was widened.
8. **Ariane 5 Flight 501, 4 June 1996.** About 37 seconds after main engine ignition, both Inertial Reference Systems failed within milliseconds of each other. The cause was a conversion of the horizontal bias value, BH, from a 64-bit floating point number to a 16-bit signed integer. The value exceeded 32,767 and raised an Operand Error that was not handled. The code was Ariane 4 alignment software kept running after liftoff “for commonality reasons” although it served no purpose on Ariane 5. The launcher broke up and was destroyed about 39 seconds after ignition. The inquiry board was chaired by Professor Jacques-Louis Lions and reported on 19 July 1996. The financial loss is widely reported as roughly 370 million US dollars; the inquiry report itself states no figure, so treat that number as approximate.
9. Detection tools: `-fsanitize=signed-integer-overflow` and `-fsanitize=unsigned-integer-overflow` in Clang and GCC, `__builtin_add_overflow` in GCC and Clang, `checked_add` in Rust, `Math.addExact` in Java.

■ 7.4.6 WORDS – remember these

1. Two’s complement — flip the bits and add one to negate — the standard signed integer encoding where the MSB carries weight $-2^{(n-1)}$.

2. One's complement — flip the bits to negate — an older encoding with two representations of zero and end-around carry addition.
3. Sign-magnitude — a separate sign bit plus the plain value — an encoding with two zeros, still used for the sign bit in IEEE 754 floats.
4. Overflow — the answer was too big to fit — the condition where an arithmetic result falls outside the representable range of the type.
5. Wraparound — the number rolls over the top back to the bottom — modular reduction of the result by 2^n , defined for unsigned types in C.
6. Unix time — seconds counted since the start of 1970 — POSIX `time_t`, seconds since the epoch 1970-01-01T00:00:00Z, ignoring leap seconds.

7.5 Fractions: fixed point and floating point

■ 7.5.1 PLAIN – in simple words

1. Bits count whole things. To store 3.75 you need an agreement about where the point goes.
2. The simple way is **fixed point**: decide once that the last two digits are after the point, and never change it. Storing money in paise or cents is fixed point. You store 1075 and everyone knows it means 10.75.
3. Fixed point is exact and fast, but the range is narrow. One setting must cover both tiny and huge values, and it cannot.
4. The other way is **floating point**: store the digits, and store separately where the point sits. That is scientific notation, like 6.02 times 10 to the 23. Two small numbers describe one enormous one.
5. Floating point gives a gigantic range at a price. Most values are stored slightly wrong, rounded to the nearest one the format can hold.
6. The rounding is tiny but it accumulates, and it is why 0.1 plus 0.2 does not come out as exactly 0.3.

■ 7.5.2 PLAIN – a picture in your head

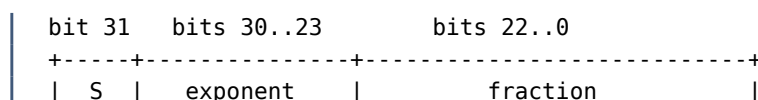
1. Imagine a metre ruler marked every millimetre. That is fixed point: every measurement is a whole number of millimetres, every gap the same size. But you cannot measure a bacterium and you cannot measure a road.
2. Now imagine an instrument that reports “3 point something, times ten to the power something”.
3. Near 1 it can tell 1.001 from 1.002. Near a million it can only tell 1,000,000 from 1,000,001,000. The steps grow with the number.
4. That is floating point. Precision is relative, not absolute. You get about the same number of significant digits at every scale.

Where this comparison breaks

1. A ruler has no equivalent of infinity, of “not a number”, or of a negative zero. Floating point has all three, and they behave in specific ways.
2. And a ruler's marks are evenly spaced everywhere. Floating point steps are even only within one power of two, and double at every power boundary.

■ 7.5.3 PLAIN – a worked example

The layout of a 32-bit float, called `binary32` or `float32`.



```

+-----+-----+-----+
1 bit    8 bits    23 bits
value = (-1)^S x 1.fraction x 2^(exponent - 127)

```

1. S is the sign. 0 means positive, 1 means negative.
2. The exponent field stores the real exponent plus 127. That offset is called the **bias**, and it lets the field stay an unsigned number.
3. The fraction stores only the digits after the point. The leading 1 is not stored, because a normalized binary number always starts with 1. That free bit is called the **hidden bit** or implicit leading one.

Encoding 0.15625 step by step.

```

Step 1: convert 0.15625 to binary by repeated doubling.
0.15625 x 2 = 0.3125 -> digit 0
0.3125 x 2 = 0.625 -> digit 0
0.625 x 2 = 1.25 -> digit 1, keep 0.25
0.25 x 2 = 0.5 -> digit 0
0.5 x 2 = 1.0 -> digit 1, keep 0.0, stop

0.15625 = 0.00101 in binary

Step 2: normalize to 1.something
0.00101 = 1.01 x 2^-3

Step 3: bias the exponent
-3 + 127 = 124 = 01111100

Step 4: take the fraction bits after the leading 1
01 followed by 21 zeros
= 010000000000000000000000

Step 5: assemble
S=0 exp=01111100 frac=010000000000000000000000

0 01111100 010000000000000000000000
= 0011 1110 0010 0000 0000 0000 0000 0000
= 0x3E200000

```

Encoding -6.25 step by step.

```

Step 1: 6 = 110 in binary. 0.25 = 0.01 in binary.
6.25 = 110.01

Step 2: normalize
110.01 = 1.1001 x 2^2

Step 3: bias
2 + 127 = 129 = 10000001

Step 4: fraction after the leading 1
1001 followed by 19 zeros
= 100100000000000000000000

Step 5: sign is 1 because the number is negative
1 10000001 100100000000000000000000
= 1100 0000 1100 1000 0000 0000 0000 0000
= 0xC0C80000

```

Decoding 0x41200000 back to a number.

```
0x41200000 = 0100 0001 0010 0000 0000 0000 0000 0000

sign      = 0                      -> positive
exponent  = 10000010 = 130         -> 130 - 127 = 3
fraction  = 010000000000000000000000
           = 0.25 (only the 2-2 bit is set)

value = +1.25 x 23 = 1.25 x 8 = 10.0
```

1. All three results can be confirmed in one line of Python: `import struct; struct.pack('>f', 0.15625).hex()` prints `3e200000`.

■ 7.5.4 PLAIN – what is really happening inside

1. When the exponent field is neither all zeros nor all ones, the value is **normal** and the hidden leading 1 applies. That is the ordinary case.
2. When the exponent field is all zeros, the hidden bit is 0 instead of 1 and the exponent is treated as -126. These are **subnormal** or denormal numbers, and they fill the tiny gap between the smallest normal number and zero. Without them, the gap around zero would be larger than the gap between the smallest two normal numbers, which breaks the rule that “a minus b is zero only if a equals b”.
3. When the exponent field is all zeros and the fraction is also all zeros, the value is zero. The sign bit still applies, so there is a positive zero and a **negative zero**. They compare as equal, but 1 divided by +0 gives positive infinity and 1 divided by -0 gives negative infinity.
4. When the exponent field is all ones and the fraction is all zeros, the value is **infinity**, positive or negative by the sign bit.
5. When the exponent field is all ones and the fraction is not zero, the value is **NaN**, meaning not a number. It is produced by 0 divided by 0, by infinity minus infinity, and by the square root of a negative number.
6. NaN has a strange and useful property: it is not equal to anything, including itself. `x != x` is a valid test for NaN, and many libraries implement `isnan` exactly that way.
7. Now the famous problem. In binary, 0.1 is a repeating fraction, exactly the way 1/3 is repeating in decimal. It is 0.0001100110011001100... forever.
8. The format has to stop somewhere, so it stores the nearest value it can. The stored value is very slightly too big.
9. 0.2 is likewise stored slightly too big. Their sum, rounded again, lands on a value slightly above the nearest stored value of 0.3.
10. So the comparison fails, not because the machine is broken, but because it is doing exactly what a finite binary format must do.

■ 7.5.5 TECHNICAL – the engineer’s version

1. IEEE 754 was approved in 1985. William Kahan of UC Berkeley led the effort; the initial proposal was drafted in 1978 by Kahan, Jerome Coonen and Harold Stone. Kahan received the ACM Turing Award in 1989. The Intel 8087 coprocessor shipped in 1980 implementing the draft, five years before approval. The standard was revised as IEEE 754-2008 and IEEE 754-2019, and adopted internationally as ISO/IEC 60559.
2. Format parameters:

Format	S / E / M bits	Bias	Approx decimal digits
binary16	1 / 5 / 10	15	3.3
bfloat16	1 / 8 / 7	127	2.4
binary32	1 / 8 / 23	127	7.2
binary64	1 / 11 / 52	1023	15.9

3. Ranges and step sizes:

Format	Largest finite	Machine epsilon
binary16	65,504	$2^{-10} = 9.77e-4$
bfloat16	$3.39e38$	$2^{-7} = 7.81e-3$
binary32	$3.4028235e38$	$2^{-23} = 1.1921e-7$
binary64	$1.7976931e308$	$2^{-52} = 2.2204e-16$

4. **Machine epsilon** is the distance from 1.0 to the next representable value above it. It equals 2^{-p} where p is the number of stored fraction bits. It is the correct constant to use in a relative tolerance test, not an arbitrary value like 0.000001.
5. The exact stored values behind the famous example, in binary64:

```
0.1 stores as
0.1000000000000000055511151231257827021181583404541015625
0.2 stores as
0.200000000000000001102230246251565404236316680908203125
their sum rounds to
0.30000000000000000444089209850062616169452667236328125
but the nearest double to 0.3 is
0.299999999999999988897769753748434595763683319091796875

so (0.1 + 0.2) == 0.3 is false, and 0.1 + 0.2 prints as
0.30000000000000004
```

6. Rounding modes defined by the standard: round to nearest with ties to even (the default), round toward zero, round toward positive infinity, round toward negative infinity, and round to nearest with ties away from zero. Ties-to-even avoids the small upward bias that ties-away introduces.
7. Money must not use binary floating point. Reasons: 0.01 is not exactly representable, sums are order-dependent, and financial rules require specific decimal rounding. Use one of:
 1. Integers in minor units, such as paise or cents, with explicit scaling.
 2. A decimal type: `decimal.Decimal` in Python, `BigDecimal` in Java, `System.Decimal` in .NET, `NUMERIC` or `DECIMAL` in SQL.
 3. The IEEE 754-2008 decimal formats `decimal32`, `decimal64` and `decimal128`, which are hardware-supported on IBM POWER and z/Architecture.
8. **bfloat16** was developed at Google Brain and used in Tensor Processing Units from TPU v2 onward. It is binary32 with the low 16 fraction bits removed, so conversion to and from float32 is a truncate or a zero-fill. Established fact: it keeps float32's exponent range, so it does not underflow on the very small gradient values that appear in deep networks, which is where binary16 struggles and needs loss scaling. Hardware support is now broad, including Intel Cooper Lake (2020), Armv8.6-A, and NVIDIA GPUs from the Ampere generation (2020).
9. Active research rather than settled practice: 8-bit formats. The E4M3 and E5M2 shapes proposed jointly by NVIDIA, Arm and Intel in 2022, and now carried in the Open Compute Project microscaling specifications, are in production use for inference and increasingly for training, but best

practice is still moving. Treat any specific claim about FP8 training quality as version-dependent and dated.

10. Inspect bit patterns with `struct.pack` in Python, `printf "%a"` in C (which prints hexadecimal floating point), `frexp` and `ldexp` in the C library, or `numpy.float32(x).view(numpy.uint32)`.

■ 7.5.6 WORDS – remember these

1. Fixed point — the point never moves — an integer with an implicit constant scale factor, usually a power of two or ten.
2. Floating point — the point moves with the exponent — a sign, an exponent and a significand encoding a value as significand times $\text{base}^{\text{exponent}}$.
3. Mantissa or significand — the digits of the number — the fractional field plus the implicit leading bit for normal values.
4. Bias — a fixed offset added to the exponent — the constant $2^{(k-1)} - 1$ that lets a k-bit exponent field stay unsigned.
5. Subnormal — very small numbers below the normal range — values with a zero exponent field and no implicit leading one, giving gradual underflow.
6. NaN — a result that is not a number — an exponent field of all ones with a non-zero significand, unequal to every value including itself.
7. Machine epsilon — the smallest step above 1.0 — 2^{-p} for p stored fraction bits; $1.19\text{e-}7$ for binary32 and $2.22\text{e-}16$ for binary64.

7.6 Text before Unicode

■ 7.6.1 PLAIN – in simple words

1. A computer stores numbers. To store letters, someone must publish a table saying which number means which letter.
2. That table is called a **character encoding**. It is only an agreement.
3. The first such agreements were for telegraphs, not computers.
4. Morse code gave each letter a pattern of short and long signals, with the common letters kept short. E is one dot.
5. Baudot code gave every letter exactly five on-off marks, so machines could handle it without a human ear.
6. IBM built its own table, EBCDIC, for its big machines.
7. Then in 1963 a committee published **ASCII**, which used 7 bits and 128 codes, and it slowly won.
8. ASCII covered English and nothing else. There was no space for accents, no Devanagari, no Chinese, no Arabic.
9. So every country invented its own extension using the spare eighth bit. Those extensions were called **code pages**.
10. A file did not say which code page it used. Open it with the wrong one and you got garbage. That mess is what Unicode was created to end.

■ 7.6.2 PLAIN – a picture in your head

1. Imagine a phone keypad where each button number stands for a letter, and two friends each keep their own written key.
2. If both keys are identical, messages arrive fine.
3. If one friend swaps a few entries, most of the message still reads, but some words come out as nonsense.

- Now imagine a hundred friends with a hundred slightly different keys, and no way to write on the envelope which key you used.
- That is exactly the code page era. Text arrived as bare numbers with no label saying how to read them.

Where this comparison breaks

- In the keypad picture, both sides know a key exists. In the real problem, most software silently assumed its own local default and never asked.
- And guessing was often close enough to look right, which was worse than failing loudly, because the corruption was saved back to disk.

■ 7.6.3 PLAIN – a worked example

The clever parts of the ASCII layout.

Character	Decimal	Hex	Binary (7 bit)
space	32	0x20	0100000
'0'	48	0x30	0110000
'9'	57	0x39	0111001
'A'	65	0x41	1000001
'Z'	90	0x5A	1011010
'a'	97	0x61	1100001
'z'	122	0x7A	1111010

- The digits sit at 0x30 to 0x39, so the low four bits are the digit value. To turn the character '7' into the number 7, mask with 0x0F.
- 'A' is 65 and 'a' is 97. The difference is 32, which is a single bit, bit 5.

```
'A' = 1000001
'a' = 1100001
      ^
      this one bit is the whole difference

To lowercase: c OR 0x20
To uppercase: c AND 0xDF
To flip case: c XOR 0x20
```

- That is why case conversion on ASCII is one instruction, not a lookup.
- Control characters occupy 0 to 31. They line up with letters too. The Ctrl key clears bits 6 and 5, so Ctrl-I is 9 (tab), Ctrl-M is 13 (carriage return), Ctrl-J is 10 (line feed), Ctrl-C is 3 (end of text).
- DEL is 127, all seven bits set. On paper tape you deleted a character by punching every hole, which no other code could be mistaken for.
- Uppercase comes before lowercase, and digits before both, so sorting by byte value roughly matches alphabetical order for plain English.

■ 7.6.4 PLAIN – what is really happening inside

- A text file is a sequence of bytes. There is no letter anywhere in it.
- When you open the file, a program picks a decoding table, looks up each byte, and asks a font for a shape to draw.
- Three separate things are involved and people confuse them constantly:

1. The **code**, meaning which number stands for which character.
2. The **encoding**, meaning how that number is packed into bytes.
3. The **font**, meaning what the shape looks like on screen.
4. For 7-bit ASCII the code and the encoding are almost the same thing, because every character fits in one byte with a spare top bit.
5. That spare top bit is where the trouble started. It doubled the table from 128 to 256 entries, and nobody agreed on the second half.
6. Russian text encoded in KOI8-R and read as Windows-1252 produces Latin letters and punctuation. Nothing crashes. The bytes are all valid. Only the meaning is lost.
7. The honest version: some code pages were also **stateful**, using shift codes to switch between banks of characters. Baudot did this with LTRS and FIGS, and some Asian encodings did it with escape sequences. In a stateful encoding you cannot decode a byte without knowing the state, so you cannot start reading in the middle of a file.

■ 7.6.5 TECHNICAL – the engineer’s version

1. Morse code was developed by Samuel Morse and Alfred Vail through the late 1830s; the famous line “What hath God wrought” was sent on 24 May 1844. It is a variable-length code with frequency-weighted symbols, an early instance of the idea Huffman later formalized. It is not binary: it has dot, dash, and three distinct gap lengths.
2. Baudot code was devised by Emile Baudot around 1870 and patented in 1874. Five bits give 32 combinations, extended by LTRS and FIGS shift codes. Donald Murray revised it in 1901; the result became International Telegraph Alphabet No. 2, standardized by the CCITT in 1930. The unit “baud” is named after Baudot.
3. EBCDIC (Extended Binary Coded Decimal Interchange Code) was created by IBM for the System/360, announced 7 April 1964. It is 8-bit and derived from punched-card BCD. Its letters are not contiguous: A to I occupy 0xC1 to 0xC9, J to R occupy 0xD1 to 0xD9, S to Z occupy 0xE2 to 0xE9. Code that tests `c >= 'A' && c <= 'Z'` is wrong on EBCDIC.
4. ASCII was published as ASA X3.4-1963 on 17 June 1963 by the American Standards Association. Work began at the first meeting of the X3.2 subcommittee on 6 October 1960. Bob Bemer was a leading contributor. Revisions followed as USAS X3.4-1967 and ANSI X3.4-1986. The international equivalents are ECMA-6 and ISO/IEC 646.
5. Common 8-bit code pages:

Encoding	Year	Covers
IBM CP437	1981	IBM PC, box drawing
ISO 8859-1	1987	Western Europe
Windows-1252	1990s	Latin-1 plus 0x80-0x9F
KOI8-R	RFC 1489, 1993	Russian Cyrillic
Shift-JIS	1982	Japanese
ISCII (IS 13194)	1991	Indian scripts

6. ISO 8859-15 arrived in 1999 mainly to add the euro sign, which ISO 8859-1 had no room for. That single missing character forced a new standard, which tells you how badly the fixed 256-slot model had run out.
7. Windows-1252 is not ISO 8859-1, although it is often labelled as such. It places printable characters in the C1 control range 0x80 to 0x9F, including the curly quotes and the em dash. Mislabelled Windows-1252 content is the single most common source of stray question marks on the web.

8. Tools: `iconv -f WINDOWS-1252 -t UTF-8`, `file -i`, `chardet` and `charset-normalizer` in Python, `enca`.

■ 7.6.6 WORDS – remember these

1. Character encoding — the table linking numbers to letters — a mapping from a character repertoire to byte sequences.
2. ASCII — the 1963 English 7-bit table — ASA X3.4-1963, 128 code positions, 33 control characters and 95 printable characters.
3. Code page — a national extension of the top 128 byte values — a vendor or national single-byte character set covering 0x80 to 0xFF.
4. Control character — a code that means an action, not a shape — ASCII 0x00 to 0x1F plus 0x7F, used for framing, flow control and terminals.
5. EBCDIC — IBM’s rival table — an 8-bit punched-card-derived encoding used on IBM mainframes, with non-contiguous alphabetic ranges.
6. Stateful encoding — you must know what mode you are in — an encoding whose byte meanings depend on preceding shift or escape sequences.

7.7 Unicode and UTF-8

■ 7.7.1 PLAIN – in simple words

1. Unicode’s idea is simple: give every character in every writing system its own permanent number, once, for everyone.
2. That number is called a **code point**. It is written as U+ then hex, for example U+0041 for the letter A.
3. Unicode assigns the numbers. It does not say how to store them in bytes. That is a separate decision.
4. There are three main storage schemes: UTF-8, UTF-16 and UTF-32.
5. **UTF-8** uses one to four bytes per character. English text takes one byte per letter, exactly like ASCII.
6. That backward compatibility is why UTF-8 took over the world. Every old ASCII file was already a valid UTF-8 file, unchanged.
7. **UTF-16** uses two bytes for most characters and four for the rest.
8. **UTF-32** uses four bytes for everything, always.
9. Once you use these, the old question “which code page is this file?” disappears. There is one table for the whole planet.

■ 7.7.2 PLAIN – a picture in your head

1. Think of a global postal system where every building on Earth gets one permanent number, and nobody ever reuses a number.
2. Unicode is that numbering scheme. U+0915 always means the Devanagari letter ka, in every country, forever.
3. Now think of how you write that number on an envelope. You could always use a fixed seven-digit box, wasting space on short numbers.
4. Or you could use a variable scheme where short numbers get short writing and long numbers get more digits, with a marker telling the reader how many digits to expect.
5. That second scheme is UTF-8. The first few bits of the first byte announce the length of the whole character.

Where this comparison breaks

1. Postal numbers name one thing each. A visible character on screen is often several code points glued together, and the glue is itself a code point.
2. And postal numbers are never reordered. Unicode has to handle text that runs right-to-left, and combining marks that attach to the character before them, so visual order and stored order differ.

■ 7.7.3 PLAIN – a worked example

The UTF-8 rules in one table.

Code point range	Bytes	Bit template
U+0000 to U+007F	1	0xxxxxxx
U+0080 to U+07FF	2	110xxxxx 10xxxxxx
U+0800 to U+FFFF	3	1110xxxx 10xxxxxx 10xxxxxx
U+10000 to U+10FFFF	4	11110xxx 10xxxxxx x2 more

1. The full 4-byte template is 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx, which carries $3 + 6 + 6 + 6 = 21$ bits.

Character one: the English letter A, U+0041.

```
0x41 = 65, which is under 128, so one byte.
Result: 41
```

Character two: the Devanagari letter ka, U+0915.

```
0x915 in binary, padded to 16 bits:
0000 1001 0001 0101

U+0915 is in the 3-byte range, template 1110xxxx 10xxxxxx 10xxxxxx
That carries  $4 + 6 + 6 = 16$  bits, so use all 16.

Split 0000100100010101 as 0000 | 100100 | 010101

byte 1 = 1110 0000 = 0xE0
byte 2 = 10 100100 = 0xA4
byte 3 = 10 010101 = 0x95

Result: E0 A4 95
```

Character three: the grinning face emoji, U+1F600.

```
0x1F600 in binary, padded to 21 bits:
0 0001 1111 0110 0000 0000
= 000011111011000000000

U+1F600 is in the 4-byte range, carrying  $3 + 6 + 6 + 6 = 21$  bits.

Split as 000 | 011111 | 011000 | 000000

byte 1 = 11110 000 = 0xF0
byte 2 = 10 011111 = 0x9F
byte 3 = 10 011000 = 0x98
byte 4 = 10 000000 = 0x80

Result: F0 9F 98 80
```

2. Confirm any of these with `python3 -c "print('A'.encode('utf-8').hex())"`.

■ 7.7.4 PLAIN – what is really happening inside

1. Look again at the templates. Every continuation byte starts with the bits
10. No leading byte ever starts with 10.
2. So if you drop into the middle of a UTF-8 stream, you can find a character boundary by walking backwards until you hit a byte that does not start with
10. You never scan back more than three bytes.
3. That property is called **self-synchronising**, and it is the reason UTF-8 beat the alternatives. A corrupted or truncated stream loses one character, not the rest of the file.
4. Second property: a byte below 0x80 is always a real ASCII character and never part of a multi-byte sequence.
5. That means old code searching for the byte 0x2F (a slash) or 0x00 (a string terminator) still works correctly on UTF-8 text without changes. This is what “file system safe” meant in the original name.
6. Third property: sorting UTF-8 byte strings gives the same order as sorting by code point. That is not true of UTF-16.
7. UTF-16 handles characters above U+FFFF with a pair of 16-bit units called a **surrogate pair**. Two thousand and forty-eight code points, U+D800 to U+DFFF, are permanently reserved for this and are never characters.
8. The arithmetic: subtract 0x10000 from the code point, leaving 20 bits. The top 10 bits go into 0xD800 plus that value, the bottom 10 into 0xDC00 plus that value.
9. For U+1F600: 0x1F600 minus 0x10000 is 0xF600. The high half is 0xD800 plus 0x3D, which is 0xD83D. The low half is 0xDC00 plus 0x200, which is 0xDE00.
10. This is why a JavaScript or Java string reports a length of 2 for a single emoji. Those languages count 16-bit units, not characters.

■ 7.7.5 TECHNICAL – the engineer’s version

1. Unicode 1.0 was published in October 1991, following Joe Becker’s 1988 draft “Unicode 88”. The Unicode Consortium was incorporated in January 1991. Unicode 2.0, in July 1996, broke the original 16-bit assumption and added surrogates. Unicode 17.0 was released on 9 September 2025 and contains 159,801 assigned characters.
2. The code space is U+0000 to U+10FFFF: 1,114,112 code points in 17 planes of 65,536 each. Plane 0 is the Basic Multilingual Plane. Plane 1 is the Supplementary Multilingual Plane, holding emoji and historic scripts. Plane 2 holds CJK extensions. Planes 15 and 16 are private use.
3. The ceiling of U+10FFFF is not a natural limit. It exists because that is the largest value UTF-16 surrogate pairs can express. UTF-8’s own structure could have reached U+1FFFFFF or, in the original 1992 design, six bytes and 31 bits. RFC 3629 cut it back to four bytes to match UTF-16.
4. UTF-8 was designed by Ken Thompson and Rob Pike in September 1992. Pike’s account states it was worked out “on a placemat in a New Jersey diner”, the scheme was described to the X/Open committee on 8 September 1992, and the whole of Plan 9 was converted within days. The original name was FSS-UTF, for File System Safe UCS Transformation Format. Pike and Thompson presented it at the USENIX Winter 1993 conference in San Diego.
5. The current specification is RFC 3629, published November 2003 by Francois Yergeau, which is also STD 63. It obsoletes RFC 2279 (1998), which obsoleted RFC 2044 (1996).
6. RFC 3629 forbids **overlong encodings**: a code point must use the shortest template that fits. Accepting overlongs was a real security hole, exploited against Microsoft IIS in 2001 to smuggle a

directory-traversal slash past a filter that checked for 0x2F only.

7. Encoding sizes for one character:

Character	UTF-8	UTF-16BE	UTF-32BE
A (U+0041)	41	00 41	00 00 00 41
e-acute (U+00E9)	C3 A9	00 E9	00 00 00 E9
ka (U+0915)	E0 A4 95	09 15	00 00 09 15
emoji (U+1F600)	F0 9F 98 80	D8 3D DE 00	00 01 F6 00

8. Byte order marks. U+FEFF at the start of a stream signals encoding and byte order: FE FF for UTF-16BE, FF FE for UTF-16LE, 00 00 FE FF for UTF-32BE, FF FE 00 00 for UTF-32LE. In UTF-8 it appears as EF BB BF. The Unicode standard permits but does not recommend a UTF-8 BOM; it breaks shell scripts, PHP output, and many CSV parsers. Note the ambiguity: FF FE alone is UTF-16LE, but FF FE 00 00 is UTF-32LE, so a decoder must look further.

9. Combining characters and normalization, defined in Unicode Standard Annex #15. The character e-acute has two spellings:

precomposed:	U+00E9	UTF-8:	C3 A9
decomposed:	U+0065 U+0301	UTF-8:	65 CC 81

They render identically. They are not equal as byte strings and not equal under a naive == comparison.

10. The four normalization forms are NFC (canonical composition), NFD (canonical decomposition), NFKC and NFKD (compatibility forms, which also fold the ligature fi to “fi” and the circled digit one to “1”). NFC is the web default; the W3C Character Model recommends it. Compatibility forms lose information and must never be used for round-tripping.
11. macOS HFS+ historically stored filenames in a variant of NFD, while Linux stores whatever bytes it is given. That mismatch broke real Git repositories containing accented or Indic filenames; `git config core.precomposeunicode true` exists for exactly this reason.
12. String length has at least four correct answers. For the four-person family emoji, built from four people joined by three zero-width joiners (U+200D):

Unit	Count
UTF-8 bytes	25
UTF-16 code units	11
Code points	7
Grapheme clusters	1

13. Grapheme cluster rules live in Unicode Standard Annex #29 and change between versions. Unicode 15.1, released September 2023, added rule GB9c so that Indic conjunct sequences such as the Devanagari cluster in “namaste” group correctly. So even “how many characters is this” depends on which Unicode version your library implements. Python’s `len` counts code points, Go’s `len` counts bytes, Java’s `String.length` counts UTF-16 units, Swift’s `count` counts grapheme clusters. Four languages, four different answers, all defensible.
14. Tools: `unicode` and `uname` on Linux, `python3 -c "import unicodedata; print(unicodedata.name(ch))"`, `iconv`, `hexdump -C`, and the ICU library.

■ 7.7.6 WORDS – remember these

1. Code point — the permanent number for a character — an integer in U+0000 to U+10FFFF assigned by the Unicode Standard.
2. UTF-8 — one to four bytes per character, ASCII-compatible — the variable width encoding defined by RFC 3629 and STD 63.
3. Surrogate pair — two halves standing for one big character — a high surrogate in U+D800 to U+DBFF followed by a low surrogate in U+DC00 to U+DFFF, encoding a code point above U+FFFF in UTF-16.
4. BOM — an invisible marker at the file start — U+FEFF used to signal encoding and byte order.
5. Mojibake — text turned to garbage by the wrong table — the result of decoding bytes with an encoding other than the one used to produce them.
6. Grapheme cluster — what a person calls one character — a sequence of code points treated as one unit by the rules in Unicode Annex #29.
7. Normalization — putting equal-looking text into one standard spelling — the NFC, NFD, NFKC and NFKD transformations of Unicode Annex #15.

7.8 Endianness

■ 7.8.1 PLAIN – in simple words

1. A number bigger than one byte must be split across several bytes.
2. There are two sensible orders to write them in, and both are in use.
3. **Big-endian** writes the biggest part first, the way we write numbers.
4. **Little-endian** writes the smallest part first, backwards from how we read.
5. Neither is better. They are two conventions that both work.
6. The trouble comes when a big-endian machine sends bytes to a little-endian machine and nobody converts. The number arrives scrambled.
7. So network protocols pick one order and everyone must obey it. They picked big-endian, and it is called network byte order.
8. Nearly every computer you own runs little-endian internally, so it converts on the way out and on the way in.

■ 7.8.2 PLAIN – a picture in your head

1. Think of writing the date. Some people write 13/08/2026, day first. Others write 2026-08-13, year first.
2. Both write the same date. Both are unambiguous once you know the rule.
3. Give a form filled in one way to a reader expecting the other, and 08/13 becomes an impossible 13th month, or worse, a plausible wrong date.
4. Endianness is the same problem at the byte level, and computers do not notice that a date is impossible. They just use the wrong number.

Where this comparison breaks

1. Date formats are ambiguous forever. Byte order is fixed by the machine and by the protocol specification, so it is always knowable.
2. Also, little-endian is not simply “backwards”. It has a real advantage: the low byte sits at the lowest address, so reading a 32-bit value as an 8-bit value needs no address adjustment.

■ 7.8.3 PLAIN – a worked example

Store the 32-bit value 0x12345678 at address 0x1000.

BIG-ENDIAN (most significant byte at the lowest address)

```
address: 0x1000 0x1001 0x1002 0x1003
byte:    12      34      56      78
         ^ biggest part first
```

LITTLE-ENDIAN (least significant byte at the lowest address)

```
address: 0x1000 0x1001 0x1002 0x1003
byte:    78      56      34      12
         ^ smallest part first
```

1. The number is identical in both. Only the layout in memory differs.
2. A real dump from a little-endian machine looks like this:

```
$ python3 -c "import sys; sys.stdout.buffer.write(
    (0x12345678).to_bytes(4,'little'))" | hexdump -C
00000000  78 56 34 12                                |xV4.|
```

3. If you send those four bytes to a big-endian reader without conversion, it reads 0x78563412, which is 2,018,915,346 instead of 305,419,896.

■ 7.8.4 PLAIN – what is really happening inside

1. The CPU does not store a number “the wrong way round”. It stores it the only way its memory interface is wired.
2. Registers have no endianness. A 64-bit register is just 64 bits. Endianness only exists when a multi-byte value touches byte-addressed memory, a file, or a wire.
3. When a program writes a 32-bit integer, the store instruction splits it and places the bytes in the architecture’s fixed order.
4. When it reads it back on the same machine, the load reverses that split perfectly. So within one machine, endianness is invisible.
5. It becomes visible the moment bytes cross a boundary: a network socket, a file shared between machines, or a memory-mapped device.
6. That is why protocol code is full of conversion calls. On a little-endian host `htonl` swaps the bytes. On a big-endian host the same function does nothing at all and compiles away.
7. This is also why casting a byte buffer straight into a struct pointer is a portability bug, even though it usually works on the machine you tested on.

■ 7.8.5 TECHNICAL – the engineer’s version

1. The terms come from Danny Cohen’s memo “On Holy Wars and a Plea for Peace”, Internet Experiment Note 137, dated 1 April 1980, later published in IEEE Computer in October 1981. Cohen borrowed Big-Endians and Little-Endians from Jonathan Swift’s Gulliver’s Travels of 1726, where the dispute is over which end of a boiled egg to open.
2. Architecture byte orders:

Architecture	Byte order
x86, x86-64	Little-endian

Architecture	Byte order
ARM (in practice)	Little-endian
RISC-V	Little-endian
SPARC, m68k, z/Arch	Big-endian

3. ARM, PowerPC, MIPS and RISC-V are bi-endian, meaning the mode is selectable. In practice ARM Linux, Android, iOS and macOS all run little-endian, and Linux on POWER moved to little-endian with the ppc64le port around POWER8 in 2014.
4. Network byte order is big-endian, fixed by the Internet Protocol specification, RFC 791 (September 1981), and restated in RFC 1700. The conversion functions are `htons`, `htonl`, `ntohs`, `ntohl` in `<arpa/inet.h>`, plus `htobe32` and friends in `<endian.h>` on Linux.
5. File formats also pick a side. PNG, JPEG, Java class files and most internet formats are big-endian. ZIP, GIF, BMP and RIFF or WAV are little-endian. TIFF is explicitly both: the file starts with “II” for Intel little-endian or “MM” for Motorola big-endian.
6. There is also **middle-endian**, seen on the PDP-11, which stored 32-bit values as two 16-bit words in big-endian order with each word little-endian inside. It is why “PDP-endian” is a term.
7. Bit order within a byte is a separate question, handled by the hardware serializer rather than by software.
8. Observe with `lscpu | grep -i "byte order"`, `sysctl hw.byteorder` on macOS, `python3 -c "import sys; print(sys.byteorder)"`, and the C macro `__BYTE_ORDER__` in GCC and Clang.

■ 7.8.6 WORDS – remember these

1. Endianness — which end of a number goes first in memory — the byte ordering convention for multi-byte scalar values.
2. Big-endian — biggest byte at the lowest address — most significant byte first, the order used by network protocols.
3. Little-endian — smallest byte at the lowest address — least significant byte first, used by x86-64, ARM and RISC-V.
4. Network byte order — the order the internet insists on — big-endian, fixed by RFC 791 for all IP header fields.
5. Byte swap — reversing the order — the `bswap` instruction on x86 or the `REV` instruction on ARM, one cycle in hardware.

7.9 Checking that data is intact

■ 7.9.1 PLAIN – in simple words

1. Bits get flipped. Cables pick up noise, memory cells lose charge, disks develop weak spots, cosmic rays strike chips.
2. A flipped bit is silent. Nothing complains. The data is simply wrong.
3. So we send extra information alongside the data, computed from the data itself, and check that it still matches.
4. The simplest is a **parity bit**: one extra bit making the number of 1s even. If it comes out odd, something changed.
5. A **checksum** is slightly better: add all the bytes and send the total.
6. A **CRC** is much stronger. It uses division rather than addition, and it catches whole runs of damaged bits.

7. None of these stop a deliberate attacker, who can simply recompute the check value. For that you need a **cryptographic hash**, designed so that nobody can find a second message with the same value.
8. **ECC memory** goes further still. It stores enough extra bits to spot a single flipped bit and repair it on the fly.

■ 7.9.2 PLAIN – a picture in your head

1. A shop counts a delivery of boxes and writes the total on the paperwork.
2. If one box goes missing, the count disagrees and the error is caught. That is a checksum.
3. If one box goes missing and someone adds a different box, the count still matches. The check passed and the delivery is still wrong.
4. Now suppose the paperwork records not just the count but the weight, the volume, and a code derived from the serial numbers in order.
5. Now swapping one box for another almost certainly disagrees somewhere. That is roughly what a CRC does.
6. And suppose there is enough redundant paperwork to work out exactly which box is missing and reorder it automatically. That is ECC.

Where this comparison breaks

1. A shop clerk can think and investigate. A CRC only ever says “matches” or “does not match”. It never says which bit changed or how to fix it.
2. And the paperwork analogy suggests deliberate fraud is hard. With a CRC it is trivially easy, because the formula is public and reversible.

■ 7.9.3 PLAIN – a worked example

Parity on one byte.

```
Data byte: 1 0 1 1 0 1 0 0    number of 1s = 4, which is even
Even parity bit = 0    -> sent as 10110100 0
```

Now flip one bit in transit:

```
Received: 1 0 1 1 0 1 1 0    number of 1s = 5, which is odd
The parity bit says it should be even -> error detected
```

Now flip two bits:

```
Received: 1 0 1 1 1 1 1 0    number of 1s = 6, still even
Parity says fine. The error is missed.
```

1. Parity detects any odd number of flipped bits and misses every even number.
2. It can never repair anything, because it does not know which bit moved.

A real CRC32 value you can reproduce:

```
$ python3 -c "import zlib; print(hex(zlib.crc32(b'123456789')))"
0xcbf43926
```

3. That value, 0xCBf43926 for the nine ASCII digits “123456789”, is the standard published check value for CRC-32. Any correct implementation produces it. It is the first thing to test when you write one.

■ 7.9.4 PLAIN – what is really happening inside

1. A CRC treats the whole message as one enormous binary number.
2. It divides that number by a fixed constant and keeps only the remainder. The remainder is the CRC value.
3. The division is not ordinary division. It uses XOR instead of subtraction, so there are no carries and no borrows.
4. In practice you do it by shifting the message through a register, and every time a 1 falls off the top you XOR the register with the constant.
5. The constant is called the **polynomial**, because mathematicians describe the same operation as polynomial division over the field with two elements. The name is historical; you can implement it knowing only XOR and shift.
6. The strength comes from that division. Adding bytes lets errors cancel out. Division spreads every input bit's influence across the whole remainder.
7. ECC memory works differently. It computes several overlapping parity bits, each covering a different subset of the data bits.
8. When a bit flips, the pattern of which parity checks fail points directly at the position of the flipped bit. Flip it back and the data is repaired.
9. This is a Hamming code. Adding one more overall parity bit lets it also detect, without correcting, a double flip. That combination is called SECDED, single error correct, double error detect.

■ 7.9.5 TECHNICAL – the engineer's version

1. Parity is used in UART serial framing, written as 8N1 for eight data bits, no parity, one stop bit, or 7E1 for seven data bits, even parity, one stop bit. Hamming distance 2: it detects one error, corrects none.
2. The Internet checksum, specified in RFC 1071, is the 16-bit one's complement of the one's complement sum of 16-bit words. It protects the IPv4 header, TCP and UDP. It is weak: it cannot detect a reordering of 16-bit words, and it misses many pairs of compensating errors.
3. The CRC idea was published by W. Wesley Peterson in 1961. CRC-32 as used in Ethernet (IEEE 802.3), PNG, gzip and ZIP uses the polynomial 0x04C11DB7 in normal notation, or 0xEDB88320 reflected, with an initial value of all ones and a final XOR of all ones.
4. Guaranteed properties of a 32-bit CRC with a well-chosen polynomial:
 1. Detects all single-bit and all double-bit errors within its bound.
 2. Detects any odd number of bit errors, if the polynomial includes the factor $(x + 1)$.
 3. Detects all burst errors up to 32 bits long.
 4. Misses longer bursts with probability 2^{-32} , about one in 4.3 billion.
5. Common CRC variants:

Name	Width	Polynomial	Used in
CRC-32	32	0x04C11DB7	Ethernet, PNG, zip
CRC-32C	32	0x1EDC6F41	iSCSI, ext4, Btrfs
CRC-16-CCITT	16	0x1021	XMODEM, Bluetooth
CRC-8	8	0x07	SMBus, 1-Wire

6. CRC-32C has a dedicated x86 instruction, `crc32`, added with SSE4.2 in the Intel Nehalem generation (2008), which is why modern filesystems chose it.
7. Cryptographic hashes solve a different problem: collision resistance against an adversary. Named here and covered properly in the security chapter: MD5 (1992, collisions demonstrated in 2004),

SHA-1 (1995, collision demonstrated by Google and CWI Amsterdam in February 2017, the SHAttered attack), the SHA-2 family including SHA-256 (2001), SHA-3 based on Keccak (standardized 2015), and BLAKE3 (2020). A CRC is not a hash and must never be used where tampering is possible.

8. ECC DRAM adds 8 check bits per 64 data bits, so a registered ECC DIMM is 72 bits wide rather than 64. Standard SECDED corrects one bit per 64-bit word and detects two. Chipkill, called Advanced ECC or SDDC by different vendors, arranges the code so that a whole failed DRAM device is correctable. The underlying theory is Richard Hamming's 1950 paper "Error Detecting and Error Correcting Codes" in the Bell System Technical Journal.
9. DDR5, specified in JEDEC JESD79-5 published July 2020, mandates on-die ECC inside the DRAM chip. Be careful: on-die ECC protects the array, not the bus or the connector. It is not a substitute for full ECC, and marketing material sometimes blurs the two.
10. Real failure rates, from "DRAM Errors in the Wild: A Large-Scale Field Study" by Schroeder, Pinheiro and Weber, published at SIGMETRICS 2009, covering Google's fleet from January 2006 to June 2008:

Measure	Value
DIMMs with a correctable error	8.2% per year
Correctable errors per DIMM	about 4,000 per year
DIMMs with an uncorrectable fault	0.22% per year
Machines with any error	about one third per year

11. That study overturned the belief that memory errors were rare and caused mainly by cosmic rays. Errors correlated strongly with specific DIMMs, pointing at hard faults rather than random strikes.
12. Observe ECC events with `edac-util -v, ras-mc-ctl --summary, dmesg | grep -i edac`, or `mcelog` on Linux.

■ 7.9.6 WORDS – remember these

1. Parity bit — one extra bit making the count of 1s even or odd — a distance-2 code detecting any odd number of bit errors.
2. Checksum — add everything up and compare — a weak integrity value, often a one's complement sum as in RFC 1071.
3. CRC — a remainder from carry-less division — a cyclic redundancy check, the remainder of the message polynomial divided by a generator polynomial over GF(2).
4. ECC memory — memory that repairs single flipped bits — DRAM with SECDED Hamming coding, 8 check bits per 64 data bits.
5. SECDED — fix one error, spot two — single error correct, double error detect, the standard server DRAM protection level.
6. Collision resistance — nobody can forge a match — the property that distinguishes a cryptographic hash from a checksum or CRC.

7.10 Compression

■ 7.10.1 PLAIN – in simple words

1. Real data repeats itself. English text is full of “the”. Photographs have large areas of near-identical sky. Compression finds the repetition and writes it down once instead of many times.
2. It also exploits unevenness. In English ‘e’ is far more common than ‘z’, so giving both the same eight bits is wasteful.
3. Short codes for common things and long codes for rare things shrink the total. That is the second big idea.
4. **Lossless** compression returns the exact original bytes. ZIP, PNG and gzip are lossless.
5. **Lossy** compression throws away detail people are unlikely to notice and cannot give the original back. JPEG and MP3 are lossy.
6. Never use lossy compression on text, code or a bank record. One changed character can change the meaning completely.
7. And no scheme shrinks everything. Compressing an already compressed file usually makes it very slightly larger.

■ 7.10.2 PLAIN – a picture in your head

1. Think of packing a suitcase. Folding clothes flat removes trapped air. The clothes are unchanged; you removed only the wasted space. That is lossless.
2. Now think of leaving three of your five shirts at home. The suitcase is far lighter, but those shirts are gone. That is lossy.
3. Now think of shorthand. A stenographer writes one squiggle for a whole common word and spells out the rare ones. That is entropy coding.
4. And think of a note saying “same as page 4, lines 2 to 9”. That is dictionary compression: pointing back instead of repeating.

Where this comparison breaks

1. A suitcase can always be packed a bit tighter with effort. Data has a hard mathematical floor, the entropy, below which no lossless scheme can go.
2. And unfolding a suitcase is easy either way. Decompression cost varies a great deal between schemes, which is often the deciding factor.

■ 7.10.3 PLAIN – a worked example

Run-length encoding. Replace each run with a count and a value.

```
Input (53 characters):  
WWWWWWWWWWBWWWWWWWWWWBWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWB  
  
Output (15 characters):  
12W1B12W3B24W1B  
  
53 down to 15, a saving of about 72 percent.
```

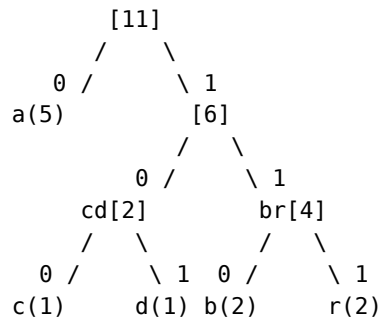
1. But run-length encoding on text with no runs makes it twice as long, since every single character becomes a count plus a character.

Huffman coding, worked completely on the word “abracadabra”.

```
Symbol counts: a=5 b=2 r=2 c=1 d=1 total 11 symbols  
  
Step 1: join the two smallest, c(1) and d(1) -> node cd = 2  
Step 2: smallest are now b(2), r(2), cd(2).
```

join b(2) and r(2)	-> node br = 4
Step 3: join cd(2) and br(4)	-> node X = 6
Step 4: join a(5) and X(6)	-> root = 11

The finished tree, with 0 for left and 1 for right:



The code table read off the tree:

Symbol	Count	Code	Bits
a	5	0	1
c	1	100	3
d	1	101	3
b	2	110	3
r	2	111	3

Encoding the whole word:

a	b	r	a	c	a	d	a	b	r	a
0	110	111	0	100	0	101	0	110	111	0

= 01101110100010101101110 (23 bits)

Plain ASCII would need $11 \times 8 = 88$ bits.

A fixed 3-bit code for 5 symbols would need $11 \times 3 = 33$ bits.

Huffman needs 23 bits.

- No code is a prefix of any other code, so the decoder never needs separators. It walks down the tree, emits a symbol at each leaf, and returns to the root.

■ 7.10.4 PLAIN - what is really happening inside

- Huffman coding builds the tree from the bottom by always joining the two least frequent items. Rare symbols therefore end up deepest, with the longest codes. This is provably optimal for whole-bit codes.
- LZ77 works on a completely different principle. It keeps a window of the recently seen bytes and, whenever the upcoming text has appeared before, it emits a back-reference instead of the text.
- A back-reference is a pair: how far back to look, and how many bytes to copy. For example, “go back 3 bytes and copy 9”.
- The length may exceed the distance. Copying is done one byte at a time, so “back 3, copy 9” on “abc” produces “abcabcabc”. That is how LZ77 also does run-length encoding for free.
- DEFLATE is simply both together. First LZ77 replaces repeats with back-references. Then Huffman coding compresses the resulting stream of literals, lengths and distances.
- That two-stage design is why gzip, ZIP and PNG all behave alike: they are the same algorithm with different containers around it.

7. The mathematical floor is **entropy**. For a source where symbol x has probability $p(x)$, the entropy in bits per symbol is:

$$H = - \sum \text{over all } x \text{ of } p(x) \times \log_2 p(x)$$

8. In words: it is the average number of bits genuinely needed per symbol. No lossless method can average fewer than H bits per symbol over the long run.
9. For “abracadabra” the entropy is 2.04 bits per symbol, so 22.4 bits for the whole word. Huffman achieved 23 bits. It came within one bit of the theoretical floor, which is exactly the guarantee Huffman coding offers.

■ 7.10.5 TECHNICAL – the engineer’s version

- David A. Huffman published “A Method for the Construction of Minimum-Redundancy Codes” in the Proceedings of the IRE in September 1952, written as a term paper for Robert Fano’s information theory class at MIT. Huffman coding is optimal among prefix codes with integer code lengths; it is never worse than $H + 1$ bits per symbol.
- Arithmetic coding and its modern relative asymmetric numeral systems, or ANS, published by Jarek Duda from 2009, break the whole-bit restriction and reach the entropy bound more closely. ANS is what makes Zstandard fast.
- Jacob Ziv and Abraham Lempel published “A Universal Algorithm for Sequential Data Compression” in IEEE Transactions on Information Theory in May 1977, giving LZ77. Their 1978 follow-up gave LZ78, from which LZW derives.
- DEFLATE was created by Phil Katz for PKZIP 2.0 in 1993 and specified by L. Peter Deutsch in RFC 1951, May 1996. The related containers are RFC 1950 for zlib and RFC 1952 for gzip. DEFLATE uses a 32 KiB sliding window and match lengths of 3 to 258 bytes.
- Claude Shannon defined entropy in “A Mathematical Theory of Communication”, Bell System Technical Journal, July and October 1948. The source coding theorem in that paper is the statement that H is a hard lower bound.
- Modern general-purpose compressors:

Name	Year	Basis
DEFLATE	1993	LZ77 plus Huffman
bzip2	1996	Burrows-Wheeler plus Huffman
LZMA / xz	2001	LZ77 plus range coding
Brotli	2015	LZ77, Huffman, static dict
Zstandard	2016	LZ77 plus ANS

- Brotli is specified in RFC 7932 (2016) and includes a built-in 120 KiB dictionary of common web strings. Zstandard, created by Yann Collet at Facebook, is specified in RFC 8878 (February 2021) and is now the default in Linux kernel modules, Btrfs and many package managers.
- The counting argument for why universal compression is impossible: there are 2^n distinct inputs of length n and only $2^n - 1$ possible outputs shorter than n bits in total across all shorter lengths. A lossless injective map cannot fit them all. Therefore any scheme that shrinks some inputs must expand others. DEFLATE’s stored block mode caps the expansion at about 5 bytes per 65,535-byte block.
- Lossless versus lossy, with real formats: lossless are PNG, FLAC, ALAC, gzip, ZIP, TIFF with LZW; lossy are JPEG, WebP in lossy mode, MP3, AAC, Opus, H.264, H.265, AV1. Image and

audio codecs are covered in Chapters 8 and 9; the compression machinery underneath them is what you have just read.

10. Measure with `gzip -9 -c file | wc -c, zstd -19, xz -9e`, and `ent` for a quick entropy estimate of a file.

■ 7.10.6 WORDS – remember these

1. Lossless — you get the exact bytes back — a reversible encoding, as in DEFLATE, PNG and FLAC.
2. Lossy — some detail is discarded forever — an irreversible encoding tuned to human perception, as in JPEG and MP3.
3. Run-length encoding — write “twelve whites” instead of twelve whites — a code replacing each maximal run with a count and a value.
4. Huffman coding — short codes for common symbols — an optimal prefix code built bottom-up by repeatedly merging the two least frequent nodes.
5. Prefix code — no code is the start of another — a uniquely decodable code requiring no separators between symbols.
6. LZ77 — point back to text you already sent — sliding-window dictionary compression emitting (distance, length) back-references.
7. Entropy — the true information content — $H = -\sum p(x) \log_2 p(x)$ bits per symbol, the lower bound on lossless compression.

7.11 How a number becomes a thing

■ 7.11.1 PLAIN – in simple words

1. Here is the deepest idea in this chapter, and it is short: a pattern of bits means nothing at all by itself.
2. The same 32 bits can be a number, a fraction, four letters, a colour or a machine instruction.
3. Nothing in memory says which. There is no tag, no label, no type stored alongside the data.
4. The meaning comes entirely from what the program decides to do with it. If the program treats those bits as a number, they are a number. If it treats them as text, they are text.
5. That is why file formats begin with magic numbers, why network protocols have header fields, and why programming languages have types. Each is a way of writing down, somewhere, what the bits are supposed to mean.

■ 7.11.2 PLAIN – a picture in your head

1. Think of the digits 0812. On a door they are a house number. On a form they are a date. On a receipt they are a price. On a lock they are a code.
2. The digits never change. Where you find them, and what you were expecting, decides what they mean.
3. Now remove all context. Someone hands you a slip of paper with 0812 and walks away. You genuinely cannot know.
4. Raw memory is that slip of paper. Every byte, every time.

Where this comparison breaks

1. A person seeing 0812 on a receipt can notice the context is wrong and ask.
2. A program cannot. It applies whatever interpretation it was written with, silently, and produces a confident wrong answer. That failure mode is called **type confusion** and it is a common security vulnerability class.

■ 7.11.3 PLAIN – a worked example

One pattern, five readings. The bits are 0x41424344.

The 32 bits:

0100 0001 0100 0010 0100 0011 0100 0100

Read as a 32-bit unsigned integer	-> 1,094,861,636
Read as a 32-bit signed integer	-> 1,094,861,636
Read as an IEEE 754 float32	-> 12.141422
Read as four ASCII bytes	-> A B C D
Read as R,G,B,A colour bytes	-> 65, 66, 67, 68
Read as two 16-bit big-endian integers	-> 16706 and 17220

1. The float reading works out as sign 0, exponent 0x82 which is 130, giving 2^3 , and a fraction of 0.5176778, so 1.5176778 times 8 is 12.141422.
2. The colour reading gives a very dark, slightly blue grey. In web notation the first three bytes are the colour #414243.
3. If the same four bytes are read on a little-endian machine as one 32-bit integer loaded from memory in the other order, the value becomes 0x44434241, which is 1,145,258,561. A sixth reading, from byte order alone.

You can reproduce all of it:

```
import struct
p = 0x41424344
b = struct.pack('>I', p)
print(struct.unpack('>I', b)[0]) # 1094861636
print(struct.unpack('>f', b)[0]) # 12.141422271728516
print(b.decode('ascii')) # ABCD
print(list(b)) # [65, 66, 67, 68]
```

■ 7.11.4 PLAIN – what is really happening inside

1. Memory is a flat array of bytes with addresses. It carries no type information whatsoever.
2. The type lives in the compiled code, not in the data. When the compiler saw `int x`, it emitted an integer load instruction for every use of `x`.
3. If the same address is later read through a float pointer, the CPU issues a floating point load, and the floating point unit interprets the exponent and fraction fields of whatever is there.
4. Neither instruction checks anything. Both succeed. Only one is meaningful.
5. A **cast** in C changes nothing in memory. It changes which instruction the compiler emits.
6. A C union deliberately overlaps two types at one address, which is the controlled version of the same trick.
7. This is also why the same bytes can be executed. Jump to an address and the CPU fetches those bytes into the instruction decoder instead of a register. Attacks that redirect execution into data exploit exactly this.
8. The honest version: some systems do tag data in hardware. The Burroughs B5000 and Lisp machines tagged words; CHERI, developed at Cambridge and Arm and shipped in the Arm Morello prototype in 2022, tags pointers with capability metadata. These are real but exceptional. Mainstream hardware stores untagged bits.

■ 7.11.5 TECHNICAL – the engineer's version

1. Type is a static property of a program, not a runtime property of memory, on all mainstream hardware. C's object model calls the intended reading the **effective type** of the storage.

2. Reading an object through a pointer of an incompatible type violates the strict aliasing rule in C and C++ and is undefined behaviour. The legal way to reinterpret bits is `memcpy`, or `std::bit_cast` in C++20, both of which compile to nothing on real targets.
3. `-fno-strict-aliasing` disables the compiler's use of that rule. The Linux kernel builds with it, because too much existing code depends on aliasing.
4. Type confusion is CWE-843 and appears regularly in browser and interpreter vulnerabilities, where an object is allocated as one class and later accessed as another.
5. Because meaning is external, every serious data format carries it explicitly. Examples already met in this chapter: file magic numbers, the TIFF "II" or "MM" tag, HTTP Content-Type headers, and Unicode byte order marks.
6. The same principle explains why the encoding of a text file cannot be determined with certainty. Charset detectors such as `charset-normalizer` are statistical guesses, not measurements. Any file of bytes below 0x80 is simultaneously valid ASCII, valid UTF-8, valid Latin-1 and valid Windows-1252.
7. Inspect the same bytes several ways with `xxd`, `od -t x1 -t c -t d4`, `objdump -D -b binary -m i386:x86-64`, and `numpy.ndarray.view`.

■ 7.11.6 WORDS – remember these

1. Interpretation — the decision about what bits mean — the effective type applied to a region of storage by the code that accesses it.
2. Cast — telling the compiler to read bits differently — a conversion that may reinterpret or may genuinely convert, depending on the language.
3. Type confusion — using data as the wrong kind of thing — CWE-843, accessing an object through an incompatible type, a common exploit class.
4. Magic number — a format's fingerprint — a fixed byte signature at a known offset used to identify content when no external label exists.
5. Strict aliasing — the rule that objects of different types do not overlap — a C and C++ assumption that permits optimization and makes type punning through pointers undefined.

7.98 Common wrong ideas

1. Wrong: computers use binary because it is more efficient. Right: they use it because a switch has two reliable states. Binary is the widest noise margin you can get from one wire, not the densest coding.
2. Wrong: hexadecimal is a different kind of number that computers understand. Right: hex is only a way of writing binary for humans. The machine stores bits; hex appears when something prints them.
3. Wrong: a byte has always been 8 bits. Right: the CDC 6600 used 6-bit characters and the PDP-10 used programmable 6, 7, 8 or 9-bit bytes. That is precisely why standards say "octet".
4. Wrong: a 64-bit CPU can address 16 exbibytes of memory. Right: the architecture defines 64-bit pointers, but shipping x86-64 chips implement 48 address bits, giving 256 TiB, or 57 bits with 5-level paging.
5. Wrong: floating point numbers are approximate because computers are imprecise. Right: every `float32` and `float64` value is an exact binary number. The error comes from rounding a decimal input to the nearest representable binary value, which is a property of the format, not a hardware defect.
6. Wrong: use a small tolerance like 0.000001 to compare floats. Right: use a relative tolerance based on machine epsilon, 1.19e-7 for `float32` and 2.22e-16 for `float64`, scaled to the magnitude being

compared.

7. Wrong: UTF-8 is a 1-byte encoding and Unicode is a 2-byte encoding. Right: Unicode is a numbering scheme with no byte width at all. UTF-8 uses 1 to 4 bytes and UTF-16 uses 2 or 4.
8. Wrong: a Unicode character is one code point. Right: what a reader calls one character is a grapheme cluster, which may be many code points. The family emoji is 7 code points and 25 UTF-8 bytes.
9. Wrong: a CRC or a checksum proves the data was not tampered with. Right: it detects accidental corruption only. Anyone can alter data and recompute a CRC. Tamper resistance needs a cryptographic hash or a MAC.
10. Wrong: compression can shrink any file if the algorithm is clever enough. Right: by a simple counting argument, any lossless scheme that shrinks some inputs must expand others. Shannon entropy sets a hard floor.

7.99 Chapter summary in 20 lines

1. Computers use base 2 because a wire is reliably above or below a threshold, and nothing in between can be read back safely.
2. N bits hold 2^N patterns, giving unsigned values from 0 to 2^N minus 1.
3. Convert decimal to binary by repeated division by 2, or by greedy subtraction of descending powers of 2. Both give the same answer.
4. Hexadecimal is binary written four bits at a time. Two hex digits are always exactly one byte, which is why engineers read memory in hex.
5. A byte is 8 bits today, but was not always, which is why standards say octet. A word is the CPU's natural width, now 64 bits.
6. A 32-bit address space names 4,294,967,296 bytes, exactly 4 GiB, which is the wall 32-bit systems hit.
7. Two's complement won because flipping the bits and adding one makes subtraction into addition, so one adder circuit does both jobs.
8. Signed 32-bit tops out at 2,147,483,647. That number caused the Year 2038 problem and the YouTube view counter overflow of 2 December 2014.
9. Ariane 5 Flight 501 was destroyed on 4 June 1996 because a 64-bit float was converted into a 16-bit signed integer that could not hold it.
10. IEEE 754, approved in 1985 under William Kahan, stores a sign bit, a biased exponent and a fraction with an implicit leading one.
11. In float32 the bias is 127, so 0.15625 encodes as 0x3E200000 and -6.25 encodes as 0xC0C80000.
12. All-ones and all-zeros exponents are reserved, giving zero, negative zero, subnormals, infinities and NaN, which is unequal to itself.
13. 0.1 plus 0.2 is not 0.3 because 0.1 is a repeating binary fraction, so both inputs and the sum are rounded to different nearby values. Money must use integer minor units or a decimal type.
14. ASCII, published as ASA X3.4-1963 on 17 June 1963, used 7 bits, put digits at 0x30 so masking gives their value, and separated 'A' from 'a' by one bit, bit 5.
15. The spare eighth bit produced the code page era, where the same bytes meant different characters in different countries, with no label in the file.
16. Unicode gives every character one permanent code point from U+0000 to U+10FFFF, across 17 planes; version 17.0 of 9 September 2025 holds 159,801 characters.
17. UTF-8, designed by Ken Thompson and Rob Pike in September 1992 and now specified by RFC 3629, is 1 to 4 bytes, ASCII-compatible and self-synchronising, which is why it won.

18. Endianness decides which byte of a multi-byte value sits at the lowest address; network byte order is big-endian, while x86-64 and ARM are little-endian.
19. Parity detects odd numbers of flipped bits, CRC32 catches all bursts up to 32 bits, ECC memory repairs one bit per 64 and detects two, and none of them resist a deliberate attacker.
20. Compression works because data repeats and is uneven; Huffman gives short codes to common symbols, LZ77 points back at earlier text, DEFLATE is both, and Shannon entropy is the floor none of them can pass.

Pixels, Screens and How You See an Image

8.0 What this chapter gives you

1. You will be able to say what light is, and why your eye sees a slice of it.
2. You will be able to explain why three colours can fake millions.
3. You will be able to say what a pixel is, and why it is not one dot.
4. You will be able to work out the PPI of any screen from two numbers.
5. You will be able to say when extra pixels stop helping.
6. You will be able to explain how an LCD blocks light and an OLED makes it.
7. You will be able to describe a framebuffer, its stride and double buffering.
8. You will be able to do the bandwidth sum for 4K at 60 Hz and 120 Hz.
9. You will be able to explain tearing, vsync and adaptive sync.
10. You will be able to explain how a finger becomes two numbers.

Drawing the picture, by the graphics chip, is Chapter 22. This chapter runs from light, through the panel, to the pixels in memory and out to your eye.

8.1 Light and the eye

■ 8.1.1 PLAIN – in simple words

1. Light travels as a wave, and the distance from one crest to the next is its wavelength.
2. Your eye reacts to about 380 to 700 nanometres, a nanometre being a billionth of a metre. Short waves look violet, long waves red.
3. At the back of the eye sits the retina. Rods work in dim light and report no colour; cones report colour.
4. You have only three kinds of cone, so every colour you see is your brain comparing three numbers.

■ 8.1.2 PLAIN – a picture in your head

1. Imagine three people watching one lamp through tinted glasses, one bluish, one greenish, one reddish.
2. None can name a colour. Each shouts one number: how bright it looks.
3. Those three numbers are the whole message your brain gets.
4. So a different lamp giving the same three numbers cannot be told apart.

Where this comparison breaks: the glasses overlap heavily, since green and red cones peak only about 30 nanometres apart, and the retina turns the three numbers into difference signals before they leave the eye.

■ 8.1.3 PLAIN – a worked example

1. A pure beam at 580 nanometres looks yellow. Red at 630 plus green at 530 on one spot can look identical, though physically they differ.
2. Below are illustrative cone responses scaled to 100, not measurements.

Light source	S cone	M cone	L cone
Pure 580 nm	0	60	85
630 nm + 530 nm	0	60	85

3. Two lights matching this way are **metamers**. Not an illusion: it follows from having only three sensors.

■ 8.1.4 PLAIN – what is really happening inside

1. A photon strikes a cone, a molecule changes shape, and the cell's voltage changes. That is the signal.
2. Whether a photon is caught depends on wavelength, and each cone type has its own catching curve.
3. Once caught, the wavelength is forgotten. The cell only says "I caught one". This is univariance.
4. The retina combines the three into light-dark, red-green and blue-yellow channels.
5. About 6 million cones feed about 1 million nerve fibres per eye, so heavy compression happens inside the eye.

■ 8.1.5 TECHNICAL – the engineer's version

1. The CIE, the international body for light measurement, quotes 380 nm to 780 nm; useful sensitivity is roughly 400 nm to 700 nm.
2. The honest version: there is no hard edge. Sensitivity falls smoothly, and at high intensity people detect out to about 310 nm and 1100 nm. Any stated boundary is a convention.

Cone	Name	Peak (nm)	Share of cones
S	Short wave	about 420	about 2%
M	Medium	about 530	roughly 30%
L	Long wave	about 560	roughly 65%

3. The L to M ratio ranges from about 1.1:1 to 16:1 with no difference in colour naming. Rods number 90 to 120 million per retina and cones 6 to 7 million; sources differ, so treat these as approximate.
4. The CIE 1931 standard observer defines colour matching over 2 degrees, the 1964 supplementary observer over 10. The CIE 1924 photopic curve peaks at 555 nm and the scotopic curve at 507 nm, a shift called the Purkinje effect.
5. Trichromatic theory came from Thomas Young in 1802 and Hermann von Helmholtz in the 1850s; cone pigments were measured directly only in the 1960s.
6. Tools: a spectroradiometer measures the full spectral power distribution, while a colorimeter such as an X-Rite i1Display measures only tristimulus values, which is why cheap meters get fooled by new panel types.

■ 8.1.6 WORDS – remember these

1. Wavelength — how long one wave is — crest-to-crest distance in nanometres.
2. Retina — the light-sensing sheet in the eye — neural tissue holding the photoreceptors.
3. Rod — the dim-light cell — scotopic photoreceptor, peak 507 nm, no colour.
4. Cone — the colour cell — photopic photoreceptor in S, M and L types.
5. Metamerism — different lights looking identical — distinct spectra, equal tristimulus values.
6. Tristimulus — three numbers for a colour — CIE X, Y, Z from the standard observer.

8.2 What a pixel actually is

■ 8.2.1 PLAIN – in simple words

1. A pixel is the smallest part of a picture the screen controls separately. The word is short for picture element.
2. It is not one dot but a group of separate lights, called sub-pixels, usually one red, one green and one blue.
3. Yellow means red on, green on, blue off. White is all on, black is all off.
4. Your eye cannot separate them, so it blends them into one colour. A pixel is a control unit, not an object you could pick up.

■ 8.2.2 PLAIN – a picture in your head

1. Think of a stadium crowd holding coloured cards to make a giant picture.
2. On the pitch you see individual people with red, green and blue cards.
3. From the far end the people vanish and one smooth image appears. Only your ability to separate the parts changed.
4. Each block of three neighbours is one pixel.

Where this comparison breaks: cards are up or down, but sub-pixels have hundreds of brightness levels, and in some layouts a pixel borrows a sub-pixel from its neighbour.

■ 8.2.3 PLAIN – a worked example

1. Look closely at white on a monitor, through a drop of water acting as a lens. You see repeating vertical stripes.

one pixel	one pixel	one pixel
+---+---+---+	+---+---+---+	+---+---+---+
R G B	R G B	R G B
+---+---+---+	+---+---+---+	+---+---+---+

RGB stripe: 3 sub-pixels per pixel, side by side

2. A 1920 by 1080 monitor has 2,073,600 pixels, so 6,220,800 driven sub-pixels.
3. On a 24-inch 1080p panel one pixel is about 0.277 mm wide, so one sub-pixel is about 0.092 mm. Thinner than a hair.

■ 8.2.4 PLAIN – what is really happening inside

1. Each sub-pixel has its own switch, and drivers set a voltage deciding how much light passes or is emitted.
2. The three land on the same tiny patch of retina, and your cones add up all light in that patch without reporting where it came from.
3. Software can exploit this. Lighting only the red sub-pixel puts a sliver of edge one third of a pixel to the left.

4. That is sub-pixel anti-aliasing. Anti-aliasing means smoothing a jagged staircase edge with in-between brightness levels.
5. It triples horizontal text accuracy at the cost of faint colour fringes.

■ 8.2.5 TECHNICAL – the engineer’s version

Layout	Sub-pixels per pixel	Typical use
RGB stripe	3 full	Monitors, TVs
RGBW	4, W adds brightness	LG WOLED TVs
PenTile RGBG	2 per pixel, shared	AMOLED phones
QD-OLED	3 in a triangle	Samsung panels

1. PenTile came from Clairvoyante, founded by Candice Brown Elliott; Samsung bought the intellectual property in 2008 and formed Nouvoyance.
2. PenTile RGBG uses one red and one blue per two greens, exploiting peak luminance sensitivity near green, so a “2560 by 1440” PenTile panel has fewer than 2560 x 1440 x 3 sub-pixels.
3. The honest version: makers count PenTile panels by pixel address, not sub-pixel count. Both are defensible; marketing picks the larger.
4. Sub-pixel rendering was announced by Microsoft as ClearType at COMDEX in 1998. It needs the physical stripe order, so it fails on rotated monitors and on PenTile.
5. Apple removed it in macOS 10.14 Mojave, September 2018, arguing Retina panels no longer need it. Users on non-Retina external monitors disagreed, and that disagreement is still live.
6. Fill factor is the pixel area that actually emits or transmits; the rest is transistors, wiring and black matrix. LCDs are typically 50% to 70%.
7. Tools: a 200x USB microscope resolves sub-pixels, `xrandr --verbose` reports geometry, and `fontconfig’s rgba` setting selects the stripe order.

■ 8.2.6 WORDS – remember these

1. Pixel — the smallest controllable picture part — one addressable sample in the framebuffer and panel matrix.
2. Sub-pixel — one coloured light inside a pixel — an individually driven cell with its own transistor.
3. RGB stripe — red, green, blue side by side — the conventional vertical geometry of desktop LCDs.
4. PenTile — a layout that shares sub-pixels — RGBG or RGBW with under three sub-pixels per address.
5. Anti-aliasing — smoothing jagged edges — filtering to reduce spatial aliasing.
6. Fill factor — how much of a pixel lights up — emissive area over total pixel area.

8.3 Resolution and PPI

■ 8.3.1 PLAIN – in simple words

1. Resolution is a count. “1920 by 1080” means 1920 columns and 1080 rows, so 2,073,600 pixels. It says nothing about sharpness.
2. PPI is a density: pixels per inch. It says how tightly packed they are.
3. A phone and a television can share a resolution and look nothing alike, one packing it into 6 inches and the other over 55.
4. What decides sharpness is neither alone. It is how big one pixel looks from where you sit.

■ 8.3.2 PLAIN – a picture in your head

1. Think of a mosaic of square tiles. Resolution is the number of tiles; density is how small each one is.
2. Nose against the wall, you see every tile edge. Walk back and the edges disappear.
3. That distance depends on tile size, not on how many tiles the mural has.
4. A billboard of huge tiles looks perfect from the road and terrible from arm's length. Nothing about it changed.

Where this comparison breaks: tiles have grout lines and pixels usually do not, and your eye blurs slightly from its own lens imperfections, hiding the grid before your neurons get involved.

■ 8.3.3 PLAIN – a worked example

1. Take the diagonal in pixels, then divide by the diagonal in inches.

```
diagonal_px = sqrt(width_px^2 + height_px^2)
PPI         = diagonal_px / diagonal_inches
24 in 1080p : sqrt(1920^2+1080^2)=2202.9 ; /24 = 91.8 PPI
27 in 1440p : sqrt(2560^2+1440^2)=2937.2 ; /27 = 108.8 PPI
6.8 in phone: sqrt(3120^2+1440^2)=3436.3 ; /6.8= 505.3 PPI
```

2. Samsung publishes 505 PPI for the Galaxy S24 Ultra, which matches.

Screen	Resolution	Size	PPI
Office monitor	1920 x 1080	24 in	91.8
4K monitor	3840 x 2160	27 in	163.2
4K television	3840 x 2160	55 in	80.1
Galaxy S24 Ultra	3120 x 1440	6.8 in	505.3

3. The television has four times the pixels of the office monitor and a lower density.

■ 8.3.4 PLAIN – what is really happening inside

1. Your eye measures angle, not size.
2. A good eye just separates two lines about one arcminute apart. An arcminute is one sixtieth of a degree, the definition behind 20/20 vision.
3. So the question is whether one pixel covers less than one arcminute of your view.
4. Shortcut: the PPI you need is about 3438 divided by viewing distance in inches, so 287 PPI at 12 inches, 143 at 24 inches, 34 at 100 inches.
5. This is why a 505 PPI phone and an 80 PPI television both look sharp. Each beats the requirement at its own distance.

■ 8.3.5 TECHNICAL – the engineer's version

1. Angular density is measured in pixels per degree, PPD. The classical 1 arcminute limit corresponds to 60 PPD.

```
PPD = PPI * distance_inches * tan(1 degree)
     = PPI * distance_inches * 0.017455
```

Display	PPI	Distance	PPD
24 in 1080p	91.8	24 in	38.5

Display	PPI	Distance	PPD
27 in 1440p	108.8	24 in	45.6
27 in 4K	163.2	24 in	68.4
Phone at 505 PPI	505.3	12 in	105.8

- Apple introduced the term Retina with the iPhone 4 on 7 June 2010: 960 by 640 on 3.5 inches, 326 PPI, claiming 300 PPI at 10 to 12 inches beats the eye's limit.
- Raymond Soneira of DisplayMate argued the limit is nearer 0.6 arcminute, implying about 477 PPI at 12 inches. Others defended 1 arcminute. Experts genuinely disagree, and acuity varies between people.
- Ashraf, Chapiro and Mantiuk published in Nature Communications on 27 October 2025 a measured foveal achromatic limit of 94 PPD, individuals reaching 120 PPD, with chromatic limits lower at 89 PPD red-green and 53 PPD yellow-violet. So 60 PPD is a floor, not a ceiling.
- PPI is display pixel density; DPI is printer ink dot density. Printer dots are binary, so a 1200 DPI inkjet needs many dots per halftone cell and delivers perhaps 200 to 300 PPI of real detail.
- The honest version: software calls display scaling "DPI" anyway. Windows does, and CSS fixes one inch at 96 CSS pixels. That is a convention, not a measurement.
- Tools: `xdpinfo | grep resolution` on X11, `system_profiler SPDisplaysDataType` on macOS, and `edid-decode` for the panel's physical size.

■ 8.3.6 WORDS – remember these

- Resolution — how many pixels — horizontal and vertical sample counts of a raster.
- PPI — how tightly packed the pixels are — pixels per linear inch of diagonal.
- PPD — how big a pixel looks from where you sit — pixels per degree of visual angle.
- Arcminute — one sixtieth of a degree — the unit of visual acuity, the 20/20 reference.
- DPI — printer dot density — dots per inch of binary ink, not the same as PPI.
- EDID — the panel's identity card — Extended Display Identification Data, giving size and timings.

8.4 Colour

■ 8.4.1 PLAIN – in simple words

- A screen stores each pixel as three numbers: red, green, blue. Most use 8 bits each, giving 256 levels from 0 for off to 255 for full.
- $256 \times 256 \times 256 = 16,777,216$ combinations, the famous 16.7 million colours.
- Better screens use 10 bits each, about 1.07 billion combinations. More levels does not mean more colourful; it means smoother steps.
- A colour space is a written agreement on exactly which red, green and blue you meant. Without it the same numbers look different everywhere.

■ 8.4.2 PLAIN – a picture in your head

- Think of three paint tins, red, green and blue, each with 256 notches.
- But "red" only helps if we both own the same tin of red paint.
- A colour space is the label on the tin. sRGB is the small tin everybody has, DCI-P3 is bigger and more vivid, Rec.2020 is huge and no consumer screen fully owns it.
- Sending sRGB numbers to a P3 screen without saying so is like following a recipe written for another tin. Everything comes out too strong.

Where this comparison breaks: paint mixes by subtracting light and screens by adding it. The notches are also not evenly spaced in brightness, which is the next idea.

■ 8.4.3 PLAIN – a worked example

1. Mid grey is written 128, 128, 128. You would expect half of white's brightness. It emits about 21%.
2. The reason is gamma: light out is roughly $(\text{value} / 255)$ to the power 2.2, and 0.502 to the power 2.2 = 0.216.
3. This is deliberate. Your eye sees differences in dark tones far better than bright ones, so gamma spends the 256 codes where you can see them.

3840 x 2160 pixels	= 8,294,400 pixels
8,294,400 x 3 bytes (24 bpp)	= 24,883,200 B = 23.7 MiB
at 60 frames per second	= 1.49 GB per second
at 4 bytes per pixel	= 33,177,600 B = 31.6 MiB

4. One second of raw 4K at 60 Hz is about 1.5 gigabytes. That is why every video file you have opened is compressed.

■ 8.4.4 PLAIN – what is really happening inside

1. Each number becomes a voltage on one sub-pixel through a lookup table, because the panel's own response is not a straight line either.
2. Gamma appears twice: encoding gamma when the image is stored, decoding gamma at the display. They are meant to cancel.
3. White point sets what 255, 255, 255 looks like, stated as colour temperature in kelvin. Lower is warmer and orange, higher is bluer.
4. If white is wrong, every colour is wrong, because everything is measured relative to white.
5. HDR changes the deal: instead of "255 is as bright as this screen goes", a code means an absolute number of candelas per square metre.

■ 8.4.5 TECHNICAL – the engineer's version

Space	Standard	Year	Note
sRGB	IEC 61966-2-1	1999	HP and MS, 1996
Rec.709	ITU-R BT.709	1990	Same primaries
Adobe RGB	Adobe de facto	1998	Wider green
DCI-P3	SMPTE RP 431-2	2011	Cinema projection
Rec.2020	ITU-R BT.2020	2012	UHD, very wide

1. sRGB primaries in CIE 1931 xy: red (0.6400, 0.3300), green (0.3000, 0.6000), blue (0.1500, 0.0600), white D65 at (0.3127, 0.3290), whose nominal correlated colour temperature is 6504 K.
2. The sRGB transfer function is piecewise, not a pure power law:

```

if C_srgb <= 0.04045: C_lin = C_srgb / 12.92
else:                  C_lin = ((C_srgb + 0.055) / 1.055) ^ 2.4

```

3. The exponent is 2.4, but the linear toe near black makes the whole curve approximate 2.2. "sRGB gamma is 2.2" is a useful lie, not the specification.
4. Bit depths give 16,777,216 codes at 8-bit and 1,073,741,824 at 10-bit. Many consumer "10-bit" panels are 8-bit plus frame rate control, which dithers over time.

5. HDR curves: PQ, standardized as SMPTE ST 2084 in 2014, is absolute and defined to 10,000 cd/m². HLG, from ARIB STD-B67 and ITU-R BT.2100, is relative and backward compatible with SDR broadcast.
6. VESA DisplayHDR tiers state peak luminance: 400, 600, 1000, 1400, plus True Black 400, 500 and 600. Marketing claim to distrust: an “HDR” badge with no tier number promises only that the panel accepts an HDR signal.
7. Tools: ArgyllCMS dispcal and colprof build ICC profiles and dispwin loads them; ColorSync Utility inspects them on macOS.

■ 8.4.6 WORDS – remember these

1. Colour space — which red you meant — defined primaries, white point and transfer function.
2. Gamma — the bend between stored number and light — a nonlinear transfer function, nominally 2.2.
3. White point — the agreed shade of white — a chromaticity coordinate, usually D65 at 6504 K.
4. Bit depth — how many steps per colour — bits per component, 8, 10 or 12.
5. Gamut — the colours a screen can hit — the triangle enclosed by the display primaries.
6. PQ — the HDR curve with absolute brightness — the Perceptual Quantizer of SMPTE ST 2084.

8.5 How a display makes light, part 1: LCD

■ 8.5.1 PLAIN – in simple words

1. An LCD does not make light. It blocks light. Behind the screen is a flat lamp that is always on: the backlight.
2. Liquid crystal flows like a liquid but its molecules line up like a crystal.
3. A voltage twists or untwists them, so light either passes or is stopped, and a coloured filter then makes it red, green or blue.
4. Because the lamp is always on, LCD black is a lamp behind a nearly closed shutter. Some light always leaks.

■ 8.5.2 PLAIN – a picture in your head

1. Picture a window with two polarizing sunglass lenses, one rotated 90 degrees from the other. Together they block almost everything.
2. Now put a twisting layer between them that can rotate light by 90 degrees.
3. Twist on: light passes the first lens, is rotated, and lines up with the second. Bright.
4. Twist off: light passes the first unchanged and is stopped by the second. Dark. That twisting layer is the liquid crystal.

Where this comparison breaks: crossed polarizers reject far more than sunglasses, and the twist is not on or off but varies smoothly, which is how you get 256 grey levels instead of two.

■ 8.5.3 PLAIN – a worked example

1. Here is the stack, from lamp to eye.

```

backlight (LED bar or sheet)
|
diffuser + brightness films
|
rear polarizer (passes vertical waves only)
|
glass + TFT layer (one transistor per sub-pixel)

```

```

      |
liquid crystal (twist set by voltage)
      |
colour filter (R, G or B dye)
      |
front polarizer (passes horizontal waves only)
      |
v your eye

```

2. Blocking is never perfect. A good IPS panel leaks about 1 part in 1000, a contrast ratio near 1000:1, so if white is 300 candelas per square metre, black is about 0.3, not zero.
3. In a dark room that leak looks dark grey. It is the biggest weakness of LCD.

■ 8.5.4 PLAIN - what is really happening inside

1. Every sub-pixel has its own thin-film transistor, a TFT, and a small storage capacitor.
2. The panel is driven one row at a time: a gate line switches on that row's transistors while source drivers put the right voltage on every column.
3. Each capacitor holds its voltage until that row comes round again, one frame later. This is active matrix addressing.
4. The molecules take time to rotate, and that settling time is the panel's response time.
5. Drive voltage is reversed in polarity every frame, because constant DC would chemically damage the liquid crystal.
6. Local dimming attacks black level from behind, splitting the backlight into independently dimmed zones.

■ 8.5.5 TECHNICAL - the engineer's version

1. Liquid crystals were discovered by the Austrian botanist Friedrich Reinitzer in 1888 in cholesteryl benzoate; Otto Lehmann named the phase.
2. George Heilmeyer, Louis Zanon and Lucian Barton at RCA Laboratories demonstrated dynamic scattering in 1964, and RCA announced the first liquid crystal displays in 1968.
3. The twisted nematic effect was patented by Wolfgang Helfrich and Martin Schadt at Hoffmann-La Roche, filed 4 December 1970 and published in Applied Physics Letters on 15 February 1971. James Ferguson filed independently in the United States and the dispute ended in shared royalties.

Type	Contrast	Response	Viewing angle
TN	700-1000:1	fastest	poor, 170/160
IPS	1000-1500:1	medium	178/178
VA	2000-5000:1	slowest	178/178

4. TechSpot's aggregate measurements across many models give average contrast of about 872:1 for TN, 1037:1 for IPS and 2898:1 for VA.
5. TN is twisted nematic, IPS in-plane switching, VA vertical alignment. IPS rotates crystals in the panel plane, so off-axis colour shift is small. VA has black crush, IPS has off-angle glow, TN has vertical gamma shift.
6. Mini-LED shrinks the backlight LEDs so hundreds to thousands of dimming zones fit; high-end 2025 and 2026 monitors quote 1,000 to 5,000 zones.
7. The limit is blooming. With 2,000 zones on a 3840 by 2160 panel each zone covers about 4,000 pixels, so a white cursor on black lights a halo.
8. Tools: `ddcutil detect` and `ddcutil getvcp 10` read and set backlight level over DDC/CI.

■ 8.5.6 WORDS – remember these

1. Backlight — the lamp behind the picture — edge-lit or direct-lit LED array with diffuser films.
2. Polarizer — a filter passing light waving one way — a film selecting one linear polarization state.
3. Liquid crystal — a material that twists light under voltage — a nematic phase, field-controlled.
4. TFT — the tiny switch at each sub-pixel — thin-film transistor in amorphous silicon, LTPS or IGZO.
5. Contrast ratio — how much brighter white is than black — white luminance over black luminance.
6. Local dimming — dimming the lamp behind dark parts — zone-wise backlight modulation.

8.6 How a display makes light, part 2: OLED

■ 8.6.1 PLAIN – in simple words

1. An OLED has no backlight. Every sub-pixel is its own tiny lamp of organic, meaning carbon-based, material that glows when current flows.
2. For black, the sub-pixel switches off and emits nothing. That is true black, so contrast is effectively unlimited.
3. Switching a light on beats rotating a molecule, so OLED reacts far faster than LCD.
4. The cost is wear. Areas showing the same bright thing for years dim faster than the rest, and that uneven wear is burn-in.

■ 8.6.2 PLAIN – a picture in your head

1. An LCD is a stage lit by one floodlight, each actor holding a shutter.
2. An OLED is a stage where every actor carries their own torch, and switches it off to be invisible; on the LCD stage the floodlight still spills.
3. But every torch has a battery that runs down with use.
4. An actor lit in the same spot every night grows dimmer, and you see their outline even in dark scenes.

Where this comparison breaks: the sub-pixel does not run out like a battery. It degrades chemically, and blue degrades fastest, so worn areas shift colour as well as dimming.

■ 8.6.3 PLAIN – a worked example

1. Show full-screen black in a dark room. The LCD shows a faint grey rectangle; the OLED is indistinguishable from the wall.
2. Contrast ratio: LCD $300 / 0.3 = 1000:1$. OLED $300 / 0.0005 = 600,000:1$, and instruments often just report infinity.
3. Now show a small white square at full brightness. The OLED pushes it to 1000 candelas per square metre or more.
4. Make the square fill the screen and brightness drops sharply, often to a quarter. That is the automatic brightness limiter protecting the panel from heat and current. LCDs do not do this.

■ 8.6.4 PLAIN – what is really happening inside

1. Each OLED sub-pixel is a sandwich: a cathode, several thin organic layers and a transparent anode.
2. Current injects electrons from one side and holes, meaning missing electrons, from the other; they meet in the emissive layer and release a photon whose colour depends on that layer's chemistry.
3. Brightness follows current, so OLED circuits control current, not voltage. That is a real design difference from LCD.

4. Lowering current shifts colour slightly, so many phones instead flash the panel on and off very fast and vary how long it stays on. That is PWM, pulse width modulation.
5. Averaged over time your eye reads that flicker as lower brightness. Some people are sensitive to it and report headaches.
6. Panels also track use per sub-pixel, quietly boost worn ones, and run pixel refresh cycles when idle.

■ 8.6.5 TECHNICAL - the engineer's version

1. The efficient thin-film OLED was demonstrated by Ching Wan Tang and Steven Van Slyke at Eastman Kodak, published as "Organic Electroluminescent Diodes" in Applied Physics Letters volume 51, pages 913 to 915, in 1987. Both entered the National Inventors Hall of Fame in 2018.

Variant	How colour is made	Typical use
RGB OLED	Separate R, G, B	Phones, tablets
WOLED	White stack + filters	LG TVs
QD-OLED	Blue stack + QD layer	Samsung panels
Tandem	Stacked emitter decks	2024+ tablets

2. QD-OLED uses a blue emitter with quantum dots, nanocrystals that convert blue photons to pure red or green by size, avoiding WOLED's filter losses.
3. Tandem OLED stacks two or more emissive units in series, roughly doubling brightness at the same current density; Apple shipped it in the iPad Pro in May 2024.
4. Micro-LED replaces organics with inorganic LEDs, giving OLED blacks without burn-in. As of 2026 it is confined to very large modular walls and small wearables, because transferring millions of dice remains unsolved at cost. Treat consumer micro-LED as active engineering, not a shipping product.
5. PWM frequency is an implementation detail: older AMOLED phones used 240 Hz or 480 Hz, while 2024 and 2025 models advertise 1440 Hz, 2160 Hz and 3840 Hz. Whether PWM causes harm is contested: flicker below about 100 Hz has established effects, but evidence above 1000 Hz is weak and mostly self-reported.

Property	LCD (IPS/VA)	OLED
Black level	0.1-0.3 cd/m ²	0.0000 cd/m ²
Contrast	1000-5000:1	effectively inf.
Response	1-8 ms	0.03-0.2 ms
Full white	holds brightness	limiter kicks in
Burn-in risk	none	real, cumulative
Viewing angle	good to poor	excellent

6. Tools: a phone camera at 960 fps is the cheapest PWM detector; an oscilloscope with a photodiode is the correct one.

■ 8.6.6 WORDS - remember these

1. OLED — a screen where each dot makes its own light — organic light emitting diode, current-driven.
2. Burn-in — a permanent ghost of an old image — differential luminance degradation, worst in blue.

3. PWM — flashing fast to look dimmer — pulse width modulation of drive current.
4. Quantum dot — a nanocrystal that converts colour — emission wavelength set by particle diameter.
5. ABL — the brightness limiter — automatic brightness limiter, cutting output as lit area grows.
6. Tandem OLED — two emitter stacks in one pixel — series units sharing a charge generation layer.

8.7 The older ways

■ 8.7.1 PLAIN – in simple words

1. Before flat screens there was the cathode ray tube, or CRT: a glass vacuum bottle with a gun firing electrons at a phosphor coating that glows.
2. Magnets bend the beam so it sweeps in lines, left to right, top to bottom.
3. Each spot fades fast, so the picture is redrawn many times a second. Colour tubes use three guns and three phosphors.
4. Plasma screens later used tiny gas cells that glow when excited, and e-ink moves coloured particles and leaves them, holding an image with no power.

■ 8.7.2 PLAIN – a picture in your head

1. Imagine writing on a wall with a torch whose trail fades in a fraction of a second.
2. To keep a picture visible you must run the torch over every line again and again, faster than the glow fades.
3. One sweep across is a scanline; switching off to return to the left edge is horizontal blanking.
4. Switching off to return to the top is vertical blanking, and complete sweeps per second is the refresh rate.

Where this comparison breaks: flat panels have no beam and nothing fades. They keep the same words and blanking gaps only because every cable, timing standard and driver was written around the sweeping beam.

■ 8.7.3 PLAIN – a worked example

1. Take VGA: 640 by 480 at 60 Hz. The real signal is 800 by 525, the extra 160 columns and 45 rows being blanking.
2. $800 \times 525 \times 60 = 25,200,000$, and the published pixel clock is 25.175 MHz.

Field	Active	Total	Blanking
Horizontal (px)	640	800	160
Vertical (lines)	480	525	45

3. Modern links still carry blanking. Even DisplayPort to an OLED sends those empty periods, though reduced blanking timings shrink them.

■ 8.7.4 PLAIN – what is really happening inside

1. In a colour CRT a metal sheet full of holes, the shadow mask, sits behind the phosphor so the red gun can only reach red phosphor. It absorbs most of the beam energy, which is why CRTs ran hot.
2. In a plasma panel each cell holds a noble gas that voltage ionizes into plasma, emitting ultraviolet light.
3. That ultraviolet strikes phosphor inside the cell, which emits visible red, green or blue. Plasma is an array of tiny fluorescent tubes.

4. Plasma cells are on or off, so grey comes from switching them many times per frame, which some viewers saw as flicker.
5. In e-ink each capsule holds white particles with one charge and black ones with the opposite, in clear fluid. Voltage pulls one colour to the top, and removing it leaves them held by friction.
6. That is bistability, and it is why an e-reader holds a page on a flat battery.

■ 8.7.5 TECHNICAL - the engineer's version

1. The cathode ray tube was built by Karl Ferdinand Braun at the University of Strasbourg in 1897, and was long called the Braun tube.
2. The plasma display panel was invented in 1964 by Donald Bitzer, H. Gene Slottow and Robert Willson at the University of Illinois, for the PLATO education system.
3. E Ink Corporation was spun out of the MIT Media Lab in 1997 from work by Joseph Jacobson with Barrett Comiskey and J. D. Albert; the founding paper, "An electrophoretic ink for all-printed reflective electronic displays", appeared in Nature in 1998.
4. IBM introduced VGA in 1987 with the PS/2 line, fixing 640 by 480 at 60 Hz on a 25.175 MHz pixel clock as a floor that still appears in boot screens.
5. Vocabulary inherited from CRTs and still in every specification: scanline, horizontal and vertical blanking interval, front porch, back porch, sync pulse, interlacing, overscan.
6. Overscan is the clearest fossil: CRT televisions drew beyond the visible glass, so many televisions still crop about 2.5% of an HDMI input in 2026 unless you find the 1:1 pixel mapping setting.
7. Plasma manufacturing ended in the 2010s: Panasonic stopped in 2014, LG and Samsung in 2014 and 2015.
8. Tools: `cvt 1920 1080 60` generates timings including blanking, and `xrandr --verbose` prints the active modeline.

■ 8.7.6 WORDS - remember these

1. CRT — the old glass tube screen — deflected electron beam exciting phosphor.
2. Scanline — one horizontal sweep — one raster row, still the unit of display timing.
3. Blanking — the gap when nothing is drawn — retrace intervals, now carrying control data.
4. Phosphor — the powder that glows — a luminescent coating with a decay time.
5. Bistable — holds its image without power — a display whose two states are both stable.
6. Overscan — cropping the picture edges — legacy CRT television margin, worth switching off.

8.8 The framebuffer

■ 8.8.1 PLAIN - in simple words

1. Before a picture can be shown it must exist as numbers in memory. That block of memory is the framebuffer.
2. It holds one complete picture, row by row from the top-left corner, each pixel taking a fixed number of bytes, usually four.
3. So finding any pixel is arithmetic, not searching.
4. One thing writes into this block and another reads out of it, at different speeds and without agreeing, which causes most of the trouble that follows.

■ 8.8.2 PLAIN - a picture in your head

1. Think of a long shelf of numbered boxes. The first 1920 are the top row, the next 1920 the second row, and so on.
2. To find column 100 of row 50, count 50 full rows then 100 more boxes.

- Now suppose the shelf is built in sections of 64 boxes and each row must start at a section boundary, leaving a few boxes empty at the end.
- Those wasted boxes are padding, and the distance from one row's start to the next is the stride.

Where this comparison breaks: real memory is not one flat shelf. It has caches, pages and interleaved banks, and the padding exists to line those up, not for tidiness.

■ 8.8.3 PLAIN – a worked example

- A tiny image: 5 pixels wide, 3 tall, 4 bytes per pixel. A row of real data is 20 bytes, but rows must start on 32-byte boundaries, so the stride is 32 with 12 bytes of padding.

```
addr    contents
0x1000  px(0,0) px(1,0) px(2,0) px(3,0) px(4,0)    20 bytes
0x1014  pad pad pad                                12 bytes
0x1020  px(0,1) px(1,1) px(2,1) px(3,1) px(4,1)
0x1034  pad pad pad
0x1040  px(0,2) px(1,2) px(2,2) px(3,2) px(4,2)
0x1054  pad pad pad
total: 3 rows x 32 bytes = 96 bytes
```

- The address formula, and one 32-bit pixel in memory:

```
addr = base + (y * stride) + (x * bytes_per_pixel);
/* pixel (3,1): 0x1000 + (1*32) + (3*4) = 0x102C */

/* orange, R=255 G=128 B=0 A=255, little-endian BGRA */
/* bytes in memory:  00   80   FF   FF */
/*                   B   G   R   A */
/* read as a word:  0xFFFF8000 */
```

- The byte order in memory is the reverse of the number you would print, a permanent source of bugs.

■ 8.8.4 PLAIN – what is really happening inside

- The display controller reads the framebuffer continuously at the rate the panel needs. At 60 Hz it starts a full read every 16.67 milliseconds whether new content exists or not.
- If your program writes into the buffer while the controller reads it, the controller sends a mixture of old and new.
- On screen that is a horizontal line with one frame above and another below. That is tearing.
- The fix is two buffers: the controller reads A while the program writes B, then they swap. That is double buffering.
- The swap must happen during vertical blanking, the gap between frames, or you tear anyway.
- Triple buffering adds a third so the program can start another frame instead of stalling, at the cost of memory and about one frame of delay.

■ 8.8.5 TECHNICAL – the engineer's version

Resolution	One buffer	Triple buffered
1920 x 1080	7.91 MiB	23.7 MiB
2560 x 1440	14.06 MiB	42.2 MiB
3840 x 2160	31.64 MiB	94.9 MiB
7680 x 4320	126.6 MiB	379.7 MiB

1. Common pixel formats: XRGB8888 and ARGB8888 at 32 bits, RGB565 at 16 bits, XRGB2101010 for 10-bit HDR, and NV12 for 4:2:0 video planes.
2. Stride, also called pitch, is in bytes and is always at least width times bytes per pixel. Alignment of 64 or 256 bytes is common; some GPUs demand far more.
3. Modern GPUs rarely store the framebuffer in simple row order. They use tiled or swizzled layouts so a 2D block of pixels lands in one cache line; linear layout is what gets exported for scanout.
4. The Linux DRM/KMS model exposes this directly: a `drm_framebuffer` wraps GEM buffer objects, and `drmModePageFlip` requests a swap at the next vertical blank, returning a `DRM_EVENT_FLIP_COMPLETE` event.
5. Legacy interfaces survive: `/dev/fb0` with `FBIOGET_VSCREENINFO`, and on many boards a linear framebuffer set up by UEFI GOP before any driver loads.
6. Tearing without vsync is not a bug in the strict sense. It is the defined behaviour of an unsynchronized page flip; calling it a bug is a convention of user expectation.
7. Tools: `modetest` lists planes, CRTC and connectors; `wayland-info` reports buffer formats; `sudo fbset -i` prints stride and pixel format.

■ 8.8.6 WORDS - remember these

1. Framebuffer — memory holding one full picture — a pixel buffer scanned out by the display engine.
2. Stride — bytes from one row start to the next — the row pitch including alignment padding.
3. Double buffering — draw in one, show the other — front and back buffers swapped by a page flip.
4. Page flip — the swap itself — changing the scanout base address at vertical blank.
5. Tearing — two frames visible at once — scanout crossing a mid-frame buffer change.
6. Vertical blank — the gap between frames — the safe window for atomic updates.

8.9 Getting the picture out

■ 8.9.1 PLAIN - in simple words

1. The display controller reads the framebuffer and turns it into a signal, in the same order the old CRT beam moved. That is scanout.
2. The signal goes down a cable carrying not a picture but a stream of numbers plus timing marks saying “new line here” and “new frame here”.
3. How much data must flow depends on pixel count, bits per pixel and refreshes per second. Multiply them for the bandwidth.
4. If that is bigger than the cable can carry, you lower one of the three or squeeze the data.

■ 8.9.2 PLAIN - a picture in your head

1. Think of a conveyor belt feeding a machine that must never stop.
2. The belt must deliver at a fixed rate; too slow and the machine stalls, because there is no store to draw on.
3. Every part is one pixel, and belt speed is bandwidth.
4. A wider belt is more lanes in the cable, a faster belt is a higher rate per lane, and packing parts tighter is compression.

Where this comparison breaks: the link has a small buffer at the receiving end, and it does stop and restart during blanking and link training. The “never stops” rule is about display timing, not literally the wire.

■ 8.9.3 PLAIN – a worked example

1. Work out 4K at 60 Hz with 8 bits per colour, then at 120 Hz with 10 bits.

4K60, 8-bit:
 $3840 \times 2160 = 8,294,400$ pixels
 $\times 60$ refreshes $= 497,664,000$ pixels/s
 $\times 24$ bits per pixel $= 11.94$ Gbit/s

4K120, 10-bit:
 $8,294,400 \times 120 = 995,328,000$ pixels/s
 $\times 30$ bits per pixel $= 29.86$ Gbit/s

2. Those are active pixels only. Blanking adds about 5% with reduced blanking and about 20% with old broadcast timings, so 4K at 60 Hz needs a 594 MHz pixel clock, giving 14.26 Gbit/s on the wire.
3. HDMI 2.0 has 18 Gbit/s raw, but 8b/10b encoding leaves 14.4 Gbit/s of payload. 14.26 fits, barely. Add 10-bit colour and it does not, which is why early HDR televisions dropped to 4:2:0 chroma.

■ 8.9.4 PLAIN – what is really happening inside

1. The controller keeps a small FIFO buffer topped up from memory; if memory is too busy the FIFO empties and you see a glitch line, an underrun.
2. Pixels are serialized: parallel bytes become a fast stream of ones and zeros on a few differential wire pairs.
3. The encoding scheme adds extra bits so the receiver can recover the clock and detect errors, which is why raw and useful speed differ.
4. Before any picture flows, the source reads the display's EDID capability block, then trains the link by trying speeds until one works reliably.
5. The cable carries pixel lanes, a clock, an auxiliary control channel, hot-plug detect and power, all at once.
6. If the picture will not fit, DSC is switched on: Display Stream Compression squeezes each line with a fixed small delay and a fixed output size, so timing stays predictable.

■ 8.9.5 TECHNICAL – the engineer's version

Interface	Raw Gbit/s	Payload	Year
HDMI 2.0	18	14.4	2013
HDMI 2.1	48	42.67	2017
HDMI 2.2	96	85.3	2025
DP 1.4 HBR3	32.4	25.92	2016
DP 2.1 UHBR20	80	77.37	2022

1. Encoding efficiency explains those pairs. HDMI 2.0 TMDS uses 8b/10b at 80%; HDMI 2.1 and 2.2 use FRL with 16b/18b at 88.9%; DisplayPort to 1.4 uses 8b/10b at 80%, and DisplayPort 2.x uses 128b/132b at about 96.9%.
2. HDMI 1.0 was released on 9 December 2002 by a founder group including Hitachi, Matsushita, Philips, Silicon Image, Sony, Thomson and Toshiba. DVI 1.0 came from the Digital Display Working Group in April 1999, and DisplayPort 1.0 was approved by VESA in May 2006.
3. The HDMI Forum released version 2.2 in June 2025 with a new Ultra96 cable category. Important caveat: the higher rates are optional, so an HDMI 2.2 label does not guarantee 96 Gbit/s.

4. DisplayPort 2.1 was announced on 17 October 2022, and DP 2.1b followed with the DP80LL low-loss cable category, announced alongside HDMI 2.2 in January 2025.
5. eDP, embedded DisplayPort, is the internal laptop link; version 1.5 was published by VESA in October 2021. Its distinguishing feature is Panel Self Refresh, which lets the source stop sending frames for a static image.
6. DSC is VESA Display Stream Compression, current version 1.2a: line-based, fixed-rate and low-latency, typically 8 to 12 bits per pixel against a 24 or 30 bit source, so roughly 2:1 to 3:1. VESA calls it visually lossless, verified under ISO/IEC 29170 testing; it is not mathematically lossless, which matters for medical and reference work.
7. The journey from memory to eye:

```

framebuffer in RAM
| scanout read, row by row
display controller (CRTC) -> colour pipeline, gamma LUT
|
optional DSC encoder
|
serializer -> 4 differential lane pairs + AUX channel
|
cable (HDMI / DisplayPort / eDP)
|
timing controller (TCON) in the panel
|
source and gate drivers -> sub-pixel voltages
|
light -> your retina

```

8. Tools: `xrandr --listmonitors`, `drm_info` and `get-edid` | `parse-edid` on Linux; `dmesg` | `grep -i drm` reveals link training failures.

■ 8.9.6 WORDS - remember these

1. Scanout — reading the picture out in order — the display engine's fixed rate raster read.
2. Bandwidth — data per second on the link — bits per second, raw or payload after encoding.
3. TMDS and FRL — the two HDMI signalling schemes — Transition Minimized Differential Signalling and Fixed Rate Link.
4. DSC — squeezing the picture to fit the cable — VESA Display Stream Compression, fixed-rate.
5. Link training — the handshake before the picture — negotiating lane count, rate and equalization.
6. TCON — the chip inside the panel — timing controller driving the row and column drivers.

8.10 Refresh rate, frame rate and smoothness

■ 8.10.1 PLAIN - in simple words

1. Refresh rate is how many times a second the screen redraws itself, in hertz. Frame rate is how many new pictures a second your software makes.
2. These are two different numbers and are usually not equal.
3. A 60 Hz screen redraws 60 times a second even if a game is stuck at 17 FPS. It just shows the same picture again.
4. When the two do not line up you get a torn image or uneven motion, and adaptive sync fixes it by letting the screen wait for the software.

■ 8.10.2 PLAIN – a picture in your head

1. A photographer takes one picture every second, on the second, no matter what. A dancer moves through poses at her own irregular pace.
2. If she changes pose halfway through a shot, the photo catches half of each. That is tearing.
3. If she holds some poses for two shots and others for one, the series looks jerky even though every photo is sharp. That is stutter.
4. Adaptive sync is the photographer agreeing to shoot whenever she settles.

Where this comparison breaks: real adaptive sync works only inside a stated range, such as 48 to 144 Hz, and outside it the display repeats frames instead.

■ 8.10.3 PLAIN – a worked example

1. A 60 Hz screen has a frame period of $1000 / 60 = 16.67$ milliseconds; at 120 Hz it is 8.33 ms, at 240 Hz 4.17 ms.
2. Suppose a game takes 20 ms per frame, 50 FPS, with vsync on at 60 Hz. It misses the deadline, waits for the next one at 33.3 ms, and the effective rate collapses to 30 FPS.
3. A 20% overrun caused a 40% drop. That cliff is why vsync feels bad near a game's limit.

Refresh	Frame period	Missed deadline
60 Hz	16.67 ms	falls to 30 FPS
120 Hz	8.33 ms	falls to 60 FPS
144 Hz	6.94 ms	falls to 72 FPS
240 Hz	4.17 ms	falls to 120 FPS

4. With adaptive sync the screen simply refreshes at 50 Hz to match, and there is no cliff.

■ 8.10.4 PLAIN – what is really happening inside

1. Adaptive sync works by stretching the vertical blanking interval, the gap between frames. The display holds the last line and waits.
2. A panel can only wait so long before its pixels drift, so there is a minimum rate; below it, drivers send the same frame two or three times.
3. Response time is a panel property: how long a pixel takes to change colour.
4. Input lag is a system property: the whole time from your action to a change on screen, across device, operating system, engine, queue, cable and panel.
5. Motion blur on a sample-and-hold display is not slow pixels. It is your eye tracking a moving object while that object sits still on screen for the whole frame period.
6. That is why black frame insertion and backlight strobing reduce blur: they shorten how long each frame is visible, at the cost of brightness.

■ 8.10.5 TECHNICAL – the engineer's version

1. Nvidia announced G-Sync on 18 October 2013, using a proprietary FPGA module inside the monitor in place of the normal scaler; early modules used an Altera Arria V GX FPGA with 768 MB of DDR3L.
2. VESA added Adaptive-Sync as an optional part of DisplayPort 1.2a in May 2014, and AMD's FreeSync branding built on it with monitors shipping from 2015. Nvidia began certifying such monitors as "G-Sync Compatible" at CES in January 2019. HDMI added its own variable refresh rate in HDMI 2.1 in 2017, which is the path consoles use.

3. VESA introduced AdaptiveSync Display certification in 2022, with tested tiers such as AdaptiveSync Display 144, plus MediaSync Display for judder-free video.
4. Response time reporting is a mess. “1 ms” is usually grey-to-grey with maximum overdrive, measured between the 10% and 90% points, often with visible overshoot. GtG, MPRT and black-to-white numbers are not comparable.

Stage	Typical time
USB mouse poll, 1 kHz	1 ms
Render + present queue	5–30 ms
Scanout of full frame	frame period
LCD pixel transition	1–8 ms
OLED pixel transition	under 0.2 ms

5. Persistence blur is roughly the frame period times motion speed, a rule of thumb popularized by Blur Busters: an object moving 1000 pixels per second on a 60 Hz sample-and-hold display smears about 16.7 pixels wide.
6. So 60 Hz to 120 Hz is not mainly “more frames”. It halves persistence blur and halves the worst-case wait for a new frame, both visible in ordinary scrolling.
7. Tools: `vblank_mode=0` and `__GL_SYNC_TO_VBLANK=0` disable vsync on Linux; `presentmon` on Windows measures real present intervals and latency.

■ 8.10.6 WORDS – remember these

1. Refresh rate — how often the screen redraws — the vertical scan frequency in hertz.
2. Frame rate — how many pictures the software makes — presents per second.
3. Vsync — waiting for the screen before swapping — syncing page flips to the vertical blank.
4. Adaptive sync — the screen waits for the software — variable refresh by extending blanking.
5. Input lag — total delay from action to picture — end-to-end latency across the whole chain.
6. Persistence — how long a frame stays lit — hold time of a sample-and-hold display.

8.11 Touchscreens

■ 8.11.1 PLAIN – in simple words

1. A touchscreen is a separate sensor on or inside the display. Resistive screens feel a press; capacitive screens feel a conductive object.
2. A resistive screen is two flexible conductive sheets with a tiny gap: press hard enough and they touch, completing a circuit at that spot.
3. Resistive works with gloves and pens but needs pressure and reliably senses only one point.
4. A capacitive screen has an invisible grid of transparent wires that your finger disturbs, needing no pressure and sensing many fingers at once.
5. A dedicated chip scans the grid many times a second and hands the operating system a list of contact points as coordinates.

■ 8.11.2 PLAIN – a picture in your head

1. Picture a chessboard with a wire under every row and over every column.
2. At each crossing the wires are close but not touching, and a small amount of charge passes between them.
3. Put a finger near one crossing and some of that charge takes a shortcut through your body, so less arrives.

4. The controller pulses each row in turn, measures every column, and builds a grid of numbers. Wherever the numbers dip, there is a finger.

Where this comparison breaks: your finger is not a hole that swallows charge. It is a conductor tied to a large body acting as a weak ground, and the dip spreads over several crossings, which is what allows sub-cell accuracy.

■ 8.11.3 PLAIN – a worked example

1. A small 6 by 4 sensor grid reports these values, where higher means more disturbance.

	c0	c1	c2	c3	c4	c5
r0	0	0	0	0	0	0
r1	0	2	14	22	6	0
r2	0	3	28	45	11	0
r3	0	1	10	16	4	0

2. The peak is row 2, column 3, value 45, but whole-cell accuracy is far too coarse, so the controller interpolates across row 2: $(2 \times 28 + 3 \times 45 + 4 \times 11) / (28 + 45 + 11) = 235 / 84 = 2.80$.
3. If the panel is 1080 pixels wide over 6 columns, each column is 180 pixels, so $x = 2.80 \times 180 + 90 = 594$ pixels. That number, a matching y and a contact identifier are what the operating system receives.

■ 8.11.4 PLAIN – what is really happening inside

1. The touch controller runs its own scan loop, independent of the display and usually faster, because tracking a fast swipe needs more samples than showing it does.
2. Each scan gives a raw grid, and the controller subtracts a stored baseline to remove the effect of the case, temperature and water film.
3. It finds peaks, interpolates centres, and matches each to a contact from the previous frame, so a finger keeps its identifier while it moves.
4. It rejects blobs that are too large, which is how a phone ignores your palm and your cheek during a call.
5. Coordinates go to the main processor over I2C or SPI, with a separate interrupt line saying “new data ready”.
6. The operating system’s input stack turns that stream into gestures: tap, long press, drag, pinch, fling.

■ 8.11.5 TECHNICAL – the engineer’s version

1. The capacitive touchscreen was described by E. A. Johnson at the Royal Radar Establishment in Malvern, England, in 1965, for air traffic control. Resistive touch came from Samuel Hurst, whose Elographics was founded in 1971.
2. Two capacitive modes exist. Self capacitance measures each electrode against ground, is very sensitive, but ghosts with more than two fingers. Mutual capacitance measures each row-column intersection and is the basis of true multi-touch.
3. The transparent conductor is traditionally indium tin oxide, ITO; metal mesh and silver nanowire are used on larger and flexible panels because ITO’s sheet resistance scales badly.

Parameter	Typical value
Sensor pitch	4 to 5 mm
Scan/report rate	120 to 480 Hz
Capacitance change	around 1 picofarad
Host bus	I2C or SPI plus IRQ
Reported contacts	5 to 10 simultaneous

4. High report rates are a measurable latency win: 240 Hz or 480 Hz sampling shortens the sensing step to 4.2 ms or 2.1 ms, though the rest of the pipeline usually dominates.
5. On Linux, multi-touch reaches userspace through evdev as type B events: `ABS_MT_SLOT`, `ABS_MT_TRACKING_ID`, `ABS_MT_POSITION_X` and `ABS_MT_POSITION_Y`. Windows uses HID digitizer usage pages; Android uses the same evdev events.
6. In-cell and on-cell integration put the touch electrodes inside or directly on the display stack instead of a separate glass layer, saving thickness and an air gap. In-cell is standard on flagship phones.
7. Tools: `evtest` on Linux prints raw touch events live; `getevent -lt` does the same on Android over adb.

■ 8.11.6 WORDS - remember these

1. Resistive touch — two sheets pressed together — a voltage-divider scheme needing deflection.
2. Capacitive touch — sensing a conductive finger — measuring capacitance change on an electrode array.
3. Mutual capacitance — measuring each crossing — the row-column method enabling multi-touch.
4. Baseline — the no-touch reference reading — a slowly adapting per-node value.
5. Report rate — how often touch positions are sent — contact reporting frequency, not panel refresh.
6. In-cell — touch sensing built into the panel — electrodes integrated in the display stack.

8.12 Scaling and sharpness

■ 8.12.1 PLAIN - in simple words

1. Every flat panel has exactly one real pixel grid, its native resolution. Unlike a CRT it cannot change that grid, because it is physical.
2. Send it a picture of a different size and something must stretch or shrink it. That is scaling, and it always guesses.
3. One output pixel usually falls between input pixels, so its colour is invented from the neighbours, which is why non-native looks soft.
4. Doubling exactly is special: each input pixel becomes a clean 2 by 2 block with no guessing, which is why 1080p looks clean on a 4K panel and poor on a 1440p one.

■ 8.12.2 PLAIN - a picture in your head

1. Think of copying a drawing from squared paper onto paper with differently sized squares.
2. If the new squares are exactly half the size, each old square becomes four new ones. Perfect copy, no decisions.
3. If they are 1.33 times smaller, most old squares straddle two new ones and you must decide how to split the colour.
4. Pick the nearest and you get hard, uneven edges; blend the neighbours and you get smooth but blurry edges. Neither is wrong, because the information was never there.

Where this comparison breaks: real scalars work on gamma-encoded values, so naive averaging darkens the result. Correct scaling converts to linear light first, which many cheap scalars skip.

■ 8.12.3 PLAIN – a worked example

1. Scale one row of 4 pixels up to 8.

```

input  index:  0    1    2    3
input  value: 10   20   30   40

nearest : 10  10   20   20   30   30   40  40
bilinear : 10 12.5 17.5 22.5 27.5 32.5 37.5 40

```

2. Nearest keeps original values exactly but shows visible steps; bilinear gives a smooth ramp where no output pixel except the ends matches any original value.
3. Bicubic uses four neighbours and a smoother curve, sharper than bilinear at the cost of slight overshoot, a faint bright halo beside hard edges.

Method	Neighbours used	Look
Nearest	1	blocky, exact
Bilinear	2 per axis	soft, no halo
Bicubic	4 per axis	sharp, mild halo
Lanczos	6 or 8 per axis	sharpest, ringing

■ 8.12.4 PLAIN – what is really happening inside

1. Scaling can happen in three places, and it matters which.
2. The application can render lower and scale up itself, choosing a good method; the graphics chip can scale during output, fast and usually bilinear; the monitor's own scaler is often worst and adds lag.
3. Integer scaling is the case where the ratio is a whole number, so each input pixel becomes an exact block and nothing is invented.
4. High-DPI operating system scaling is a different problem: not how to stretch an image, but how big to draw the interface in the first place.
5. The right approach is to tell applications “one logical point is now two physical pixels” and let them draw text and vectors at full detail.
6. The fallback, for applications that do not understand this, is to let them draw small and stretch the window. That is what a blurry old application on a modern laptop looks like.

■ 8.12.5 TECHNICAL – the engineer's version

Platform	Mechanism	Typical factors
macOS	Backing scale	1x, 2x
Windows	Per-monitor DPI	100% to 350%
GNOME	Wayland fractional	100% to 300%
Android	Density buckets	mdpi to xxxhdpi

1. Apple's approach at non-integer settings is to render the desktop at 2x into an offscreen buffer and downscale to the panel. It looks correct and costs memory bandwidth; that is an implementation detail of macOS, not a general rule.

2. Wayland fractional scaling was standardized with the `wp_fractional_scale_v1` protocol, merged in 2022, letting a client render at the exact fractional scale instead of the next integer and down.
3. Android density buckets are nominal: `mdpi` is 160 dpi and defines 1 dp equal to 1 px, `hdpi` 240, `xhdpi` 320, `xxhdpi` 480, `xxxhdpi` 640. Devices are assigned to the nearest bucket, so dp is not a physical unit.
4. CSS fixes the reference pixel at 1/96 inch, and `devicePixelRatio` reports the ratio to physical pixels. That specification is why a “1px” border can be 3 physical pixels on a phone.
5. Scaling should be done in linear light. Scaling gamma-encoded sRGB values directly darkens fine detail, worst on high-contrast text, and many hardware scalars do it wrong for speed.
6. Reconstruction is moving past fixed filters. Temporal upscalers such as Nvidia DLSS, AMD FSR and Intel XeSS combine previous frames plus motion vectors, and learned variants use neural networks. Established fact: they beat bilinear at equal cost. Active research: removing ghosting and temporal instability. Marketing claim: output indistinguishable from native. Chapter 22 covers this properly.
7. Tools: `xrandr --output DP-1 --scale 1.5x1.5` applies GPU scaling on X11; on Windows, `SetProcessDpiAwarenessContext` declares whether an application handles scaling itself.

■ 8.12.6 WORDS – remember these

1. Native resolution — the panel’s real pixel grid — the fixed physical addressable matrix.
2. Scaling — resizing a picture to fit — resampling to a different sample grid.
3. Nearest neighbour — copy the closest pixel — zero-order hold resampling, exact but aliased.
4. Bilinear and bicubic — blend nearby pixels — interpolation over 2 or 4 taps per axis.
5. Integer scaling — exact whole-number blocks — resampling at 2:1 or 3:1 with no interpolation.
6. `devicePixelRatio` — real pixels per logical one — CSS reference pixels against device pixels.

8.98 Common wrong ideas

1. Wrong: more pixels always looks better. Right: beyond about 60 pixels per degree at your actual viewing distance, extra pixels cost power, bandwidth and frame time while adding very little you can see. The 2025 Nature Communications figure of 94 PPD raises that ceiling but does not remove it.
2. Wrong: 4K on a phone matters. Right: at 12 inches you need about 287 PPI to pass the 1 arcminute test, and a 6.8-inch flagship is already over 500 PPI, so 4K at that size mostly costs battery.
3. Wrong: OLED and LED are different backlights. Right: an “LED TV” is an LCD with an LED backlight, so it has one, while an OLED has no backlight at all because every sub-pixel is its own light source. The names are marketing and they are actively misleading.
4. Wrong: a bigger screen has better quality. Right: size and quality are independent, and making a panel bigger at the same resolution lowers PPI. What matters is resolution, density, contrast, colour accuracy and viewing distance together.
5. Wrong: a pixel is a single coloured dot. Right: it is a control unit of two, three or four separately driven sub-pixels, and on PenTile layouts some are shared with the neighbouring pixel.
6. Wrong: 1 ms response time means no lag. Right: response time is only the pixel transition, while input lag also covers polling, rendering, queueing, scanout and panel processing, usually ten to fifty times larger.
7. Wrong: a higher refresh rate only matters for games. Right: persistence blur scales with frame period, so 120 Hz makes ordinary text scrolling measurably clearer whatever the application is.
8. Wrong: HDR means brighter. Right: HDR means code values carry absolute luminance under a defined transfer function. A panel that accepts HDR but peaks at 350 candelas per square metre

often looks worse than in standard mode.

9. Wrong: DSC is lossy so it degrades my picture. Right: it is lossy mathematically and visually lossless under ISO/IEC 29170 testing, a real trade-off for reference work and a non-issue for desktop and gaming use.
10. Wrong: running below native resolution is like having a smaller monitor. Right: it forces a resampling step that invents pixel values, which is why text goes soft. Only exact integer ratios avoid it.

8.99 Chapter summary in 20 lines

1. Light is a wave, and your eye reacts to roughly 380 to 700 nanometres of it, with no hard boundary at either end.
2. Three cone types peaking near 420, 530 and 560 nanometres reduce every spectrum to three numbers.
3. Because only three numbers survive, different spectra can look identical. That is metamerism, and it makes RGB screens possible.
4. A pixel is not a dot. It is a group of separately driven sub-pixels, most often a red, green and blue vertical stripe.
5. Resolution is a count of pixels; PPI is a density, the diagonal pixel count divided by the diagonal in inches.
6. Sharpness is decided by angular density: about 60 pixels per degree meets the 1 arcminute limit, and the PPI needed is about 3438 divided by viewing distance in inches.
7. A 24-inch 1080p panel is 91.8 PPI, a 27-inch 1440p is 108.8 PPI and a 6.8-inch phone is 505 PPI, and all can look correct at their own distances.
8. Colour is three numbers per pixel: 8 bits each gives 16.7 million combinations, 10 bits each about 1.07 billion.
9. A colour space such as sRGB, DCI-P3 or Rec.2020 states which exact red, green, blue and white those numbers mean.
10. Gamma exists because perception is not linear: mid grey code 128 emits about 21% of white, and that is deliberate.
11. One uncompressed 4K frame at 24 bits per pixel is 23.7 MiB, so 60 per second is about 1.49 GB.
12. An LCD blocks light from a permanent backlight, so its black is never truly black, and TN, VA and IPS trade speed, contrast and viewing angle.
13. An OLED emits its own light per sub-pixel, giving true black and very fast transitions, at the cost of burn-in and brightness limiting.
14. CRTs gave us the vocabulary every modern display standard still uses: scanline, blanking, refresh rate and overscan.
15. E-ink is bistable: once the pigment particles move, they hold position with no power at all.
16. A framebuffer stores pixels row by row, and the stride is the byte distance between row starts, including alignment padding.
17. Double buffering exists because scanout never stops; swapping during the vertical blank is what prevents tearing.
18. Bandwidth is pixels times bits times refresh rate: 4K at 60 Hz 8-bit is 11.94 Gbit/s and 4K at 120 Hz 10-bit is 29.86 Gbit/s, which is why HDMI 2.1, DisplayPort 2.1 and DSC exist.
19. Refresh rate and frame rate are separate numbers; vsync couples them harshly, while adaptive sync lets the panel wait for the frame.
20. Scaling always invents pixel values except at exact integer ratios, which is why non-native resolutions look soft and 1080p is clean on a 4K panel.

Sound, Magnets, Speakers and Phone Calls

9.0 What this chapter gives you

1. You will be able to say what sound physically is, with numbers.
2. You will be able to draw a speaker and name every part in it.
3. You will be able to explain why a magnet and a coil of wire move air.
4. You will be able to explain a microphone as a speaker run backwards.
5. You will be able to turn a wave into numbers and back, and say why 44,100 samples per second was chosen.
6. You will be able to calculate the size of a song from first principles.
7. You will be able to say what a codec throws away and why you do not hear it.
8. You will be able to trace a phone call from 1876 copper to a modern internet call, and say what changed at each step.
9. You will be able to explain why voice tolerates lost data but hates delay, and why file transfer is the exact opposite.
10. You will be able to explain noise cancelling without any magic.

9.1 What sound actually is

■ 9.1.1 PLAIN – in simple words

1. Sound is air being pushed and pulled.
2. When something vibrates, it squashes the air next to it, and that squashed patch pushes the next patch, and so on outward.
3. A squashed patch is a **compression**. A stretched patch, with less air in it, is a **rarefaction**.
4. So sound is a travelling pattern of “more pressure, less pressure”.
5. Nothing travels from the drum to your ear except the pattern. The air particles jiggle back and forth in place.
6. Cycles per second is the **frequency**, in hertz (Hz). It sets the pitch.
7. How hard the squeeze is, is the **amplitude**. It sets the loudness.
8. A young ear hears roughly 20 Hz to 20,000 Hz. The top falls with age.
9. Sound moves at about 343 metres per second, which is slow. That is why you see lightning long before you hear thunder.

■ 9.1.2 PLAIN – a picture in your head

1. Picture a line of people packed shoulder to shoulder in a corridor.
2. Push the first person. They bump the next, who bumps the next.
3. A bump travels the whole corridor, but nobody walked anywhere. Each person leaned forward and came back.
4. Push rhythmically twice a second and bumps leave twice a second: 2 Hz.
5. Push harder for amplitude, faster for frequency. The gap between bumps is the **wavelength**.

Where this comparison breaks: real air particles are not in a line and are not touching. They already fly about randomly at hundreds of metres per second. Sound is a small organized pressure change riding on that permanent chaos. And the corridor carries the wave one way, while real sound spreads in a sphere.

■ 9.1.3 PLAIN – a worked example

1. Take the note A above middle C, 440 Hz by convention.
2. Wavelength = speed divided by frequency. $343 / 440 = 0.78$ m.
3. At the bottom of hearing, $343 / 20 = 17.15$ m, longer than a bus.
4. At the top, $343 / 20,000 = 0.017$ m, about 17 mm.
5. Across your hearing range, wave sizes differ by a factor of 1,000.
6. That single fact explains most of loudspeaker design. Hold on to it.

Frequency	Wavelength in air	About the size of
20 Hz	17.15 m	A five-storey building
100 Hz	3.43 m	A small room
1,000 Hz	34.3 cm	A forearm
10,000 Hz	3.43 cm	A thumb
20,000 Hz	1.72 cm	A fingernail

■ 9.1.4 PLAIN – what is really happening inside

1. Loudness is measured in decibels (dB), on a scale that is odd on purpose.
2. The ear responds to ratios, not to absolute energy. Whisper to shout is about a million times in pressure, and taking the logarithm turns that into 120.
3. Every extra 6 dB doubles the pressure. Every extra 10 dB multiplies power by ten, and sounds roughly twice as loud.
4. Zero decibels is not silence. It is a chosen reference near the quietest sound a good young ear can detect.
5. Now the ear. The outer ear funnels pressure down the canal to the eardrum, and three tiny bones pass that movement on and strengthen it.
6. The last bone pushes fluid inside a spiral tube, the cochlea.
7. Inside is a strip that is stiff at one end and floppy at the other. High notes shake the stiff end, low notes the floppy end, so the cochlea sorts frequency by physical shape alone.
8. Hair cells on that strip bend as it moves and fire a nerve signal. Pressure has become nerve pulses.

■ 9.1.5 TECHNICAL – the engineer's version

1. Sound in air is a longitudinal wave: particle motion is parallel to travel, unlike light, which is transverse.

2. Speed in dry air: $c = 331.3 + 0.606 T$ m/s, T in degrees Celsius, so 343.4 m/s at 20 C. It depends on temperature, not pressure.
3. Sound Pressure Level: $SPL = 20 \log_{10} (p / p_0)$, with $p_0 = 20$ micropascals RMS. That reference is a standard, set near threshold at 1 kHz. Intensity level uses $10 \log_{10} (I / I_0)$ with $I_0 = 1e-12$ W/m²: 10 for power, 20 for pressure.
4. Characteristic acoustic impedance of air at 20 C is about 413 Pa*s/m.
5. Equal-loudness contours, standardized in ISO 226 (2003 revision), peak near 3 to 4 kHz, matching the quarter-wave resonance of the ear canal. Age-related high-frequency loss is called presbycusis.
6. The cochlea is about 35 mm long uncoiled, with roughly 3,500 inner hair cells as sensors and about 12,000 outer hair cells acting as an active amplifier. The frequency-to-place mapping is called tonotopy.

Sound	SPL	Pressure (RMS)
Threshold of hearing	0 dB	20 uPa
Normal speech at 1 m	60 dB	20 mPa
Busy road	80 dB	200 mPa
Rock concert, front	110 dB	6.3 Pa
Threshold of pain	130 dB	63 Pa

The honest version: “20 Hz to 20 kHz” is a convention, not a wall. Sensitivity fades gradually at both ends. Below about 20 Hz you feel pressure rather than hear a pitch, and the upper figure describes young ears in a laboratory.

■ 9.1.6 WORDS – remember these

1. Compression — a squashed patch of air — pressure above ambient.
2. Rarefaction — a stretched patch of air — pressure below ambient.
3. Frequency — how fast it wobbles — cycles per second, unit hertz.
4. Amplitude — how big the wobble is — peak deviation from ambient pressure.
5. Wavelength — the length of one wobble — $\lambda = c / f$, in metres.
6. Decibel — a loudness step — $20 \log_{10}$ of a pressure ratio against 20 uPa.
7. Cochlea — the spiral in your ear — the organ performing a mechanical frequency analysis on the basilar membrane.

9.2 The speaker, in full

■ 9.2.1 PLAIN – in simple words

1. A speaker turns an electrical wiggle into an air wiggle, using four things.
2. A **permanent magnet**, a lump of magnetized metal that always pulls.
3. A **voice coil**, thin wire wound into a tube, sitting in the magnet’s gap.
4. A **cone** of stiff paper or plastic, glued to that coil.
5. A **suspension** of rubber and cloth, holding it centred so it can only move in and out.
6. Send electricity through the coil and the coil becomes a magnet itself.
7. Now there are two magnets. They push apart or pull together depending on which way the current flows, so reversing the current reverses the force.
8. The coil is glued to the cone, so the cone moves out, then in, squashing the air in front of it and then stretching it. That is a pressure wave.
9. If the signal wiggles 440 times a second, so does the cone, and you hear a 440 Hz note.

10. The speaker knows nothing about music. It copies the shape of an electrical signal into the shape of a pressure wave.

■ 9.2.2 PLAIN – a picture in your head

1. Picture standing in a swimming pool holding a large flat tray.
2. Push it forward and you shove a wall of water away. Pull it back and water rushes in to fill the space.
3. Do that rhythmically and you make waves. Faster gives closer waves, further gives taller ones.
4. The tray is the cone, the water is the air, your arms are coil and magnet.
5. A big slow wave needs a big tray and a long movement, while tiny fast ripples want a small light paddle. That is why a speaker box has a big driver for bass and a small one for treble.

Where this comparison breaks: water is nearly incompressible, while the air wave exists precisely because air can be squashed. And your arms overpower the water, whereas a real speaker turns only about 0.5 to 2 percent of its electrical power into sound. The rest becomes heat in the coil.

■ 9.2.3 PLAIN – a worked example

1. A bookshelf speaker is marked 8 ohms and 87 dB.
2. The 87 dB means: put in 1 watt, stand 1 metre away, measure 87 dB.
3. One watt into 8 ohms needs 2.83 volts, because power = volts squared divided by resistance, and 2.83 squared is 8.
4. Since +10 dB needs ten times the power, 10 W gives 97 dB and 100 W gives 107 dB.
5. So going from 87 to 107 dB needs one hundred times the amplifier power.
6. Swap in a 90 dB speaker and it reaches 97 dB on 5 W instead of 10 W. A sensitive speaker buys you more than a big amplifier does.

```
power for a target level, 8 ohm speaker
sensitivity 87 dB at 1 W / 1 m, target 100 dB at 1 m

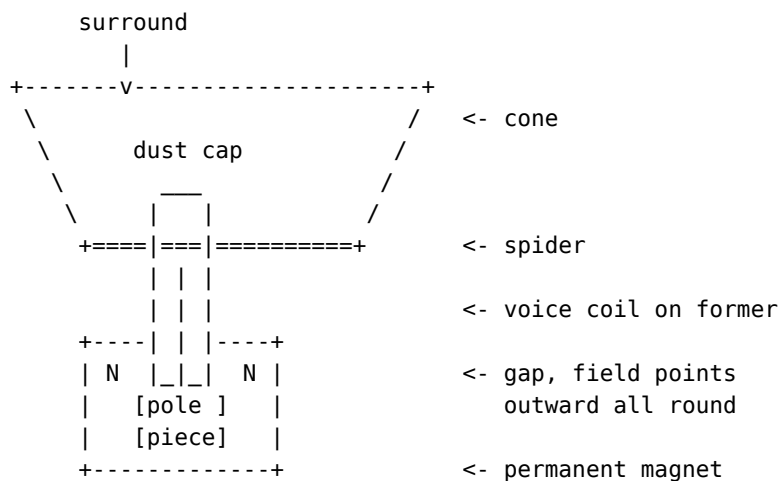
difference   = 100 - 87 = 13 dB
power ratio  = 10 ^ (13 / 10) = 19.95
power needed = 19.95 W
voltage      = sqrt(19.95 * 8) = 12.6 V RMS
current      = 12.6 / 8 = 1.58 A RMS
```

■ 9.2.4 PLAIN – what is really happening inside

1. Name the parts from the back forwards. The **magnet** is a ring or slug of ferrite or neodymium.
2. Steel shaped into a **top plate** and a **pole piece** squeezes its field into a narrow ring-shaped **gap**, where the field points straight outward.
3. The **voice coil** is wound on a stiff tube, the **former**, and sits there.
4. The **spider**, a corrugated cloth disc, keeps the coil centred, and the **surround**, a rubber ring, holds the cone edge. Together they are the suspension and act as a return spring.
5. The **cone** is glued to the former, a **dust cap** covers the middle, and the assembly bolts to a **basket**.
6. Physics law one, the Lorentz force: a wire carrying current in a magnetic field feels a sideways push, equal to field times current times wire length.
7. Because the field points outward all round the gap, and the wire runs round the gap, that push is straight forwards or backwards. Nothing is wasted.
8. Double the current and you double the force, so the cone faithfully copies the signal.

9. Physics law two, Faraday's law: a wire moving in a magnetic field generates a voltage. A moving cone therefore opposes its own drive, called back-EMF, and that is why a speaker also works, badly, as a microphone.
10. Loudness depends on the volume of air moved: cone area times travel. For the same loudness, dropping an octave needs four times the travel.
11. A 20 cm cone moving 5 mm shifts plenty of air but is too heavy to reverse 15,000 times a second. A 25 mm dome is light enough but shifts almost no air, so it makes no bass.
12. Hence two or three drivers, with a **crossover** splitting the signal.
13. Finally the box. When the cone pushes forward it pulls backward on the air behind it, producing an exactly opposite wave.
14. At high frequencies the wave is short and the cone body blocks the path. At low frequencies the wave is metres long, wraps round, and cancels.
15. A naked bass driver held in your hand makes almost no bass. The box exists to stop the back wave reaching the front.

cross-section of one driver (side view)



■ 9.2.5 TECHNICAL – the engineer's version

1. Force on the coil: $F = B * l * I$, with B the gap flux density in tesla, l the conductor length in the gap in metres, I the current in amperes. The product Bl , in tesla metres, is the driver's motor strength. Back-EMF is $V = Bl * u$, with u the cone velocity in metres per second.
2. Nominal impedance of 4, 6, 8 or 16 ohms is a **convention**, not a measured constant. An "8 ohm" driver measures about 5.5 to 6.5 ohms DC (R_e) and swings from roughly 4 ohms to over 40 ohms across the band. The peak sits at free-air resonance F_s , where the coil moves fastest and generates the most back-EMF.
3. Sensitivity is stated as dB SPL at 1 m for 2.83 V RMS, which is 1 W only into 8 ohms. Into 4 ohms that is 2 W, so 4 ohm figures flatter the driver by 3 dB.
4. Typical values: 84 to 88 dB for a sealed bookshelf, 90 to 96 dB for a floorstander, 100 to 110 dB for a horn-loaded professional cabinet. Electroacoustic efficiency is 0.5 to 2 percent for a direct radiator.
5. Low-frequency behaviour is described by the Thiele/Small parameters, developed by Neville Thiele in 1961 and extended by Richard Small in the early 1970s: F_s , Q_{ts} , V_{as} , S_d , X_{max} , R_e , L_e and M_{ms} . Volume displacement $V_d = S_d * X_{max}$, and for constant SPL required displacement rises as $1/f^2$.
6. Crossovers are LC filters, passive after the amplifier or active before it, with slopes of 6, 12, 18 or 24 dB per octave. Fourth-order Linkwitz-Riley is the common modern choice, and a two-way

typically crosses at 1.8 to 3 kHz.

7. Enclosures: sealed (acoustic suspension, 12 dB per octave rolloff, best transient behaviour), bass reflex (a Helmholtz resonator formed by port and box, 24 dB per octave, about 3 dB more output near tuning), and transmission line.
8. Why a phone cannot do bass, with numbers: S_d about 1 cm^2 , X_{max} under 0.3 mm, back volume under 1 cm^3 , so V_d is around 0.03 cm^3 . Producing 80 dB SPL at 50 Hz at half a metre needs tens of cm^3 of displacement. The phone is short by three orders of magnitude. It is a volume-of-air problem, so no amount of processing fixes it.
9. Other driver types:
 1. **Piezoelectric**: a ceramic disc that bends under voltage. No magnet or coil, cheap, high impedance, poor bass. Buzzers and alarms.
 2. **Balanced armature**: a magnetized reed balanced between two poles, driven by a coil, coupled by a rod to a tiny diaphragm. Efficient and small, standard in hearing aids since the 1940s and in in-ear monitors. Narrow bandwidth each, so in-ears stack two to twelve with crossovers.
 3. **Planar magnetic**: a thin film with a printed conductor between magnet arrays, so force spreads over the whole diaphragm and it flexes less.
 4. **Electrostatic**: a charged film between perforated stators driven at high voltage, needing a bias of one to six kilovolts. Very low moving mass and distortion, poor bass without a large panel.

Driver type	Moving part	Typical use
Dynamic (coil)	Cone plus coil	Almost everything
Balanced armature	Reed plus rod	In-ear monitors
Piezo	Ceramic disc	Buzzers, alarms
Planar magnetic	Printed film	Studio headphones
Electrostatic	Charged film	High-end panels

The honest version: “the amplifier drives the speaker” is half the story. The speaker drives back, through back-EMF and its reactive impedance. A real amplifier must absorb returning energy, which is why output impedance must be low and why cable resistance measurably shifts response with some loads.

■ 9.2.6 WORDS - remember these

1. Voice coil — the wire that gets pushed — the conductor in the magnetic gap, described by Re , Le and Bl .
2. Bl — motor strength — flux density times conductor length, in tesla metres.
3. Excursion — how far the cone travels — X_{max} , peak linear displacement in mm.
4. Sensitivity — how loud for a given input — dB SPL at 1 m for 2.83 V RMS.
5. Crossover — the circuit splitting bass from treble — a filter network with a stated order and slope in dB per octave.
6. Bass reflex — a box with a hole — a Helmholtz resonator extending output.
7. Back-EMF — the speaker generating as it moves — the motional voltage $Bl \cdot u$.
8. Thiele/Small parameters — the driver’s spec sheet — the lumped-parameter model of a driver near and below resonance.

9.3 The microphone as a speaker in reverse

■ 9.3.1 PLAIN – in simple words

1. A speaker turns electricity into movement, and movement into pressure. A microphone does the same three things in the opposite order.
2. Pressure arrives, pushes a thin sheet, and the sheet moves.
3. The moving sheet changes something electrical, and that change is the signal.
4. Way one: move a coil near a magnet. Moving wire in a field makes voltage. That is a **dynamic** microphone.
5. Way two: make the sheet one plate of a capacitor, which is two plates with a gap. Change the gap and the electrical behaviour changes. That is a **condenser** microphone.
6. Way three: build the whole thing on a chip the size of a grain of rice. That is a **MEMS** microphone, and it is the one in your phone.

■ 9.3.2 PLAIN – a picture in your head

1. Think of a trampoline with a heavy rope ring tied underneath.
2. Somebody drops a ball. The trampoline dips and springs back, and the ring goes down and up with it.
3. Now imagine the ring passes through an invisible field that makes a small voltage whenever the ring moves.
4. Big bounce, big voltage. Fast wobble, fast wobbling voltage. The trampoline is the diaphragm and the ring is the coil: a dynamic microphone.

Where this comparison breaks: a real diaphragm moves nanometres for ordinary speech, the scale of a hundred atoms. And a trampoline keeps bouncing, whereas a good diaphragm is damped so it stops the instant the sound stops, or it would add its own ringing to everything.

■ 9.3.3 PLAIN – a worked example

1. Take a condenser microphone: one fixed plate, one moving diaphragm.
2. Say the gap is 20 micrometres and the plate area is 3 cm^2 . That is roughly 13 picofarads, a very small capacitor.
3. Put a fixed charge on it and never let the charge change.
4. Voltage across a capacitor = charge divided by capacitance.
5. Sound pushes the diaphragm in by 20 nanometres, so the gap shrinks by one part in a thousand and capacitance rises by about the same fraction.
6. Charge is fixed, so voltage falls by about one part in a thousand. Charged to 48 volts, that is a change of about 48 millivolts. That is the signal.
7. Notice the trick. We did not generate power from the sound. We used the sound to modulate a power supply that was already there.

■ 9.3.4 PLAIN – what is really happening inside

1. The **dynamic** microphone is a small speaker used backwards: a light diaphragm with a coil glued to it, hanging in a magnet gap.
2. Sound moves the coil through the field and Faraday's law gives a voltage. It needs no power supply, because the sound itself does the work.
3. Because the sound must move the coil's mass, dynamics are less sensitive and less detailed high up, but rugged and happy with very loud sources.

4. The **ribbon** microphone replaces the coil with a corrugated strip of very thin aluminium which is itself the diaphragm. Very low mass, fragile, and naturally equally sensitive front and back.
5. The **condenser** needs an external voltage to charge its plates, plus an amplifier right beside the capsule, because the signal is tiny and the source impedance is enormous. That voltage is usually 48 volts sent down the same cable as the audio, called phantom power.
6. The **electret** is a condenser whose charge is permanently baked into a plastic film during manufacture, so no polarizing supply is needed. Electrets made microphones cheap enough for every telephone and laptop for forty years.
7. The **MEMS** microphone is that idea built by chip-making processes: a diaphragm a fraction of a millimetre across etched from silicon, with a perforated silicon backplate a few micrometres behind it.
8. Beside it sits a small chip that buffers the signal and often digitizes it immediately. The package is about $3 \times 4 \times 1$ mm with a sound hole.
9. Because they are made photographically, thousands come out identical. That is what makes microphone arrays and beamforming possible in a phone.

■ 9.3.5 TECHNICAL – the engineer’s version

1. Dynamic output follows $V = Bl * u$, the same expression as speaker back-EMF. Sensitivity is typically 1 to 3 mV/Pa. The Shure SM58, introduced in 1966, is specified at 1.85 mV/Pa at 1 kHz into 150 ohms.
2. Condenser capsules are 10 to 60 pF. Capacitive reactance at 20 Hz for 20 pF is about 400 megohms, so the impedance converter, a FET or valve stage, must sit within millimetres of the capsule.
3. Phantom power is standardized as P48 in IEC 61938: 48 V nominal fed through two 6.81 kilohm resistors, one per leg of a balanced pair.
4. The electret was invented at Bell Labs by Gerhard Sessler and James West in 1962, and the foil-electret condenser became the most manufactured microphone type in history.
5. Polar pattern is a property of capsule venting, not of transducer type: omnidirectional if only the front is open, figure-of-eight if front and back are equally open, cardioid if the rear path is acoustically delayed.
6. Digital MEMS parts output PDM, a one-bit stream clocked at 1 to 3.072 MHz and decimated in the host codec, or I2S/TDM directly. PDM lets a microphone sit far from the processor on two wires without picking up noise.
7. Observation: `arecord -l` lists Linux capture devices, and `system_profiler SPAudioDataType` lists macOS inputs.

Parameter	Typical MEMS	Note
Sensitivity	-38 dBV/Pa	About 12.6 mV/Pa
Signal-to-noise ratio	64 to 70 dB(A)	Higher is better
Acoustic overload point	120 to 133 dB SPL	Where 10 % THD hits
Supply current	100 to 250 μ A	Always-on capable

The honest version: “a microphone is a speaker in reverse” is exactly true for dynamic microphones and false for condenser and MEMS types. In a condenser the sound does not supply the signal energy; it modulates a bias already present. That is why a condenser needs power and a dynamic does not.

■ 9.3.6 WORDS – remember these

1. Diaphragm — the thin sheet the sound pushes — the moving membrane.
2. Transducer — anything converting one energy to another — here acoustic to electrical, or the reverse.
3. Phantom power — voltage sent up the microphone cable — P48 per IEC 61938.
4. Electret — a permanently charged film — a dielectric with frozen-in polarization, removing the need for a bias supply.
5. MEMS — a microphone built like a chip — micro-electro-mechanical system with a silicon diaphragm and an integrated ASIC.
6. Acoustic overload point — where it distorts — the SPL at which total harmonic distortion reaches 10 percent.

9.4 From wave to numbers

■ 9.4.1 PLAIN – in simple words

1. A computer cannot store a wiggle. It can only store numbers.
2. So we measure the wiggle over and over, very fast, and write the numbers down.
3. Each measurement is a **sample**: a snapshot of the voltage at one instant.
4. How many snapshots per second is the **sample rate**. A music CD takes 44,100 per second, of each of two channels.
5. How precisely each snapshot is written is the **bit depth**. A CD writes each one as a 16-bit number, one of 65,536 levels.
6. Measure often, measure precisely, write it down. That is the whole idea.
7. Playback means reading the numbers and rebuilding the voltage.

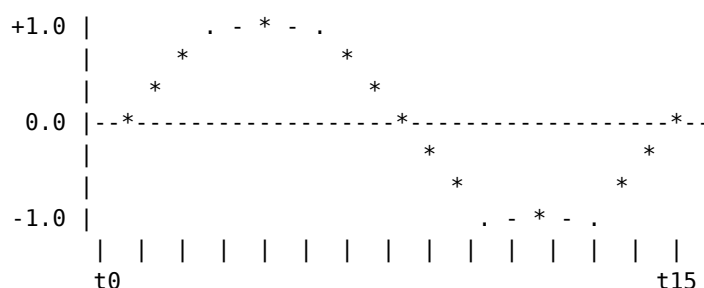
■ 9.4.2 PLAIN – a picture in your head

1. Imagine a heart-rate monitor drawing a line on moving graph paper.
2. You cannot keep the paper. You may only write the height of the line at fixed moments: 42, 58, 71, 65, 44, 30, 41.
3. Later somebody plots your numbers as dots.
4. If the dots are close enough together, exactly one smooth curve fits them, and it is the original line.
5. If they are too far apart, many curves fit equally well, and the information is gone forever.

Where this comparison breaks: on paper you would join dots with straight lines and get a jagged shape. Real reconstruction does not. The mathematics says that if you sampled fast enough there is exactly one band-limited curve through those points, and a filter finds it. The output is a smooth wave, not a staircase.

■ 9.4.3 PLAIN – a worked example

sampling a wave: samples over one cycle



the "*" marks are the values we keep. everything between them is rebuilt by the reconstruction filter.

1. Now watch what goes wrong when we sample too slowly.
2. Suppose the real signal is 30,000 Hz but we take 44,100 samples a second.
3. The readings are indistinguishable from a 14,100 Hz signal, because 44,100 minus 30,000 is 14,100.
4. A tone you could barely hear reappears as a loud whistle in the middle of your hearing range. That is **aliasing**, and it cannot be removed afterwards, because both signals produced the same numbers.
5. So we put a filter before the converter that removes everything above half the sample rate. That is the **anti-aliasing filter**, and it is not optional.
6. You have seen aliasing in films: a wheel spinning forward appears to turn slowly backwards, because 24 frames a second cannot track the spokes.

■ 9.4.4 PLAIN - what is really happening inside

1. There is a theorem behind all this, worth stating properly.
2. If a signal contains no frequency above F , sampling at any rate above $2F$ captures it perfectly, with nothing lost.
3. That is the Nyquist-Shannon sampling theorem. Harry Nyquist stated the underlying result at Bell Labs in 1928, and Claude Shannon published the modern form in 1949. "Perfectly" means perfectly, not approximately.
4. So to record up to 20,000 Hz we need more than 40,000 samples a second.
5. Why 44,100 and not 40,000? A practical accident. Before disks were big enough, early digital audio was stored on video tape using a PCM adaptor which packed samples into fake video lines.
6. 44,100 fits both video standards: 60 fields x 245 lines x 3 samples, and 50 fields x 294 lines x 3 samples, both give 44,100.
7. The extra 4,100 Hz gives the anti-aliasing filter room to roll off between 20 kHz and 22.05 kHz.
8. Now bit depth. Each sample is rounded to the nearest available level, and that **quantization error** appears as faint hiss.
9. Each extra bit halves the error, which is 6 dB quieter, so 16 bits gives about 96 dB of range and 24 bits about 144 dB.
10. One subtlety: for very quiet signals the error stops being random hiss and correlates with the music, sounding like gritty distortion.
11. The fix is **dither**: add a tiny amount of deliberate noise before rounding. It sounds wrong and it works, turning ugly distortion into benign hiss, and letting you hear signals quieter than one single level.

■ 9.4.5 TECHNICAL - the engineer's version

1. Nyquist-Shannon: a signal band-limited to B hertz is completely determined by samples at $f_s > 2B$, and $f_s/2$ is the Nyquist frequency. Energy above $f_s/2$ folds back to $|f - f_s|$ and cannot be separated afterwards, so anti-alias filtering before the sample-and-hold is mandatory.
2. Signal-to-quantization-noise ratio for an ideal uniform quantizer with a full-scale sine: $SQNR = 6.02 N + 1.76$ dB. For $N = 16$ that is 98.09 dB. The commonly quoted 96 dB is 6.02×16 without the 1.76 term.
3. Dither is usually triangular PDF noise at 2 LSB peak-to-peak, fully decorrelating the error at the cost of 4.77 dB more noise. Noise-shaped dither moves that noise above 15 kHz where the ear is least sensitive.

4. Two converter architectures dominate.
 1. **Successive approximation (SAR)**: a binary search. The held sample is compared against half full scale, then a quarter, and so on, one bit per clock, so N bits take N comparisons and latency is one conversion.
 2. **Sigma-delta**: samples at 64 to 256 times the target rate with a coarse, often one-bit, quantizer inside a feedback loop that shapes quantization noise up out of the audio band, then decimates digitally. This trades speed for resolution and is why 24-bit converters are cheap. Nearly every audio ADC and DAC since the mid-1990s is sigma-delta.
5. Oversampling also lets the analogue anti-alias filter be gentle, because the first alias lands 64 times higher. The steep filtering happens in the digital decimator, where it can be made exact and phase-linear.

Sample rate	Nyquist limit	Where it is used
8 kHz	4 kHz	Telephone, G.711
16 kHz	8 kHz	Wideband voice, AMR-WB
44.1 kHz	22.05 kHz	CD, most music files
48 kHz	24 kHz	Video, broadcast, most OS
96 kHz	48 kHz	Studio production
192 kHz	96 kHz	Studio, largely marketing

Experts disagree here. A 2007 AES paper by E. Brad Meyer and David R. Moran found listeners could not distinguish high-resolution playback from the same signal passed through a 16-bit 44.1 kHz loop. A 2016 AES meta-analysis by Joshua Reiss pooled many studies and found a small but statistically significant ability to discriminate, especially with training. Any effect is small, and higher rates are most clearly useful during production, where repeated processing accumulates error.

■ 9.4.6 WORDS – remember these

1. Sample — one measurement of the wave — one quantized amplitude at one instant.
2. Sample rate — how often we measure — samples per second, in hertz.
3. Bit depth — how precisely we measure — bits per sample, giving 2^N levels.
4. Nyquist frequency — the highest note recordable — half the sample rate.
5. Aliasing — a high note pretending to be a low one — spectral folding of content above $f_s/2$ back into the baseband.
6. Dither — helpful noise added on purpose — low-level TPDF noise decorrelating quantization error from the signal.
7. Sigma-delta — the converter in everything — oversampling noise-shaping converter plus digital decimation filter.

9.5 From numbers back to wave

■ 9.5.1 PLAIN – in simple words

1. Playback runs the recording chain backwards.
2. A chip reads the numbers in order, at exactly the right speed, and sets an output voltage to match each one. That is the **DAC**, the digital-to-analogue converter.
3. Its raw output is a series of steps, because it holds each value until the next arrives.
4. A filter smooths those steps back into the original wave. That is the **reconstruction filter**.
5. The smoothed wave is correct but far too weak to move a speaker.

6. So it goes into an **amplifier**, which uses the small wave as a template and builds a big copy, taking the energy from the power supply.

■ 9.5.2 PLAIN – a picture in your head

1. Imagine a precise but tiny artist painting a perfect miniature.
2. It is correct in every detail but one centimetre across, so nobody at the back of the hall can see it.
3. A second worker copies it onto a wall, stroke for stroke, a thousand times bigger, using buckets from a store room.
4. The tiny artist is the DAC, the wall painter is the amplifier, the store room is the power supply.
5. The painter adds nothing new, and a mistake in the miniature becomes a huge mistake on the wall. The paint comes from the store room, not the miniature.

Where this comparison breaks: a real amplifier is not a passive copier. It is a feedback system constantly comparing its output against its input and correcting the difference far faster than the music changes. It also adds a little of its own noise and distortion.

■ 9.5.3 PLAIN – a worked example

1. A laptop headphone socket puts out at most about 1 volt RMS, with a source impedance of perhaps 10 ohms.
2. Headphone A is a 32 ohm in-ear rated 105 dB per milliwatt. Power = volts squared divided by ohms, so at 0.8 V that is $0.64 / 32 = 20$ milliwatts. It will be painfully loud.
3. Headphone B is a 250 ohm studio model rated 96 dB per milliwatt. At 1 V, power = $1 / 250 = 4$ milliwatts, which is +6 dB over one milliwatt, giving about 102 dB. Loud enough, but with little headroom.
4. Now the source impedance problem. A 10 ohm output feeding a 32 ohm headphone is a divider, losing about a quarter of the voltage.
5. Worse, headphone impedance varies with frequency, so a high source impedance changes the tone. The design rule is to keep source impedance under one eighth of the load.
6. That is why 250 and 600 ohm headphones want a real amplifier, and why 16 ohm in-ears sound tonally different from a high-impedance output.

■ 9.5.4 PLAIN – what is really happening inside

1. The DAC receives a stream of numbers plus a clock saying when each is due.
2. The clock matters enormously. If its timing wobbles, the wave is rebuilt at slightly wrong moments. That is **jitter**.
3. Most audio DACs oversample rather than building the exact voltage in one step.
4. A digital filter invents extra samples between the real ones, raising the internal rate by 8, 16, 64 or 128 times.
5. Then a very fast, very coarse converter produces an output at that high rate, with its errors pushed up into ultrasonic frequencies.
6. Then a gentle analogue filter removes the ultrasonic mess. It is the sigma-delta idea run in reverse.
7. The output is typically 1 to 2 volts RMS at a few milliamps: line level, meant for another circuit, not for a speaker.
8. The amplifier keeps that voltage shape but supplies amperes.
9. A class AB amplifier uses transistors that are always partly on, wasting power as heat, at roughly 50 to 70 percent efficiency.

10. A class D amplifier switches fully on and fully off hundreds of thousands of times a second, varying how long it stays on, and a filter averages that into the wanted wave. Efficiency reaches 85 to 95 percent.
11. Class D is why a phone, a soundbar and a car stereo can be loud without a heatsink the size of a brick.

■ 9.5.5 TECHNICAL – the engineer’s version

1. The raw DAC output is a zero-order hold, imposing a sinc amplitude droop of $\sin(x)/x$ with $x = \pi \cdot f/f_s$, about -3.2 dB at 20 kHz with $f_s = 44.1$ kHz. Oversampling DACs correct it in the digital interpolation filter.
2. Reconstruction is usually a linear-phase FIR filter at 8x to 128x oversampling followed by a second or third order analogue filter. The choice between linear phase (pre-ringing) and minimum phase (phase shift) is selectable on many current DAC chips and is a matter of taste.
3. Clock jitter budget: below roughly 100 picoseconds RMS for 16-bit performance at 20 kHz, and about 1 picosecond for 24-bit. That is why asynchronous USB audio, where the DAC owns the clock, replaced adaptive USB audio.
4. Headphone impedances in use: 16 to 32 ohms for in-ears and headsets, 32 to 80 ohms for consumer over-ears, 250 and 600 ohms in the Beyerdynamic DT770 and DT880 families, 300 ohms for the Sennheiser HD 600 and HD 650. The high-impedance variants were designed for professional equipment with high output voltage. Damping factor is load impedance divided by source impedance, and the “rule of eighth” convention says keep it above 8.
5. What a product sold as “a DAC” actually contains: a USB, S/PDIF or Bluetooth receiver, a clock, the converter silicon, an analogue output stage, a volume control, usually a headphone amplifier, and a power supply. The converter chip is often a few dollars of the bill of materials, and most audible difference between such products comes from the analogue output stage, the headphone amplifier and the volume control, not the chip named on the box.
6. Observation: `cat /proc/asound/card0/pcm0p/sub0/hw_params` on Linux shows the rate, format and buffer actually in use while audio plays. On macOS, Audio MIDI Setup shows the current device rate and bit depth.

Class	How it works	Efficiency
A	Always fully conducting	20 to 30 percent
AB	Mostly on, brief overlap	50 to 70 percent
D	Switching at 300 kHz plus	85 to 95 percent
G / H	AB with stepped rails	60 to 80 percent

■ 9.5.6 WORDS – remember these

1. DAC — the chip turning numbers into voltage — digital-to-analogue converter, usually oversampling sigma-delta.
2. Zero-order hold — holding each value until the next — the staircase output causing sinc amplitude droop.
3. Reconstruction filter — the smoother — the low-pass filter recovering the unique band-limited waveform.
4. Jitter — timing wobble — deviation in sample clock edge timing, in picoseconds RMS.
5. Line level — a weak but correct signal — nominally 2 V RMS consumer or 1.228 V RMS (+4 dBu) professional.

6. Class D — the efficient switching amplifier — pulse-width modulated output stage with an LC output filter.
7. Damping factor — how firmly the amplifier grips the driver — load impedance divided by amplifier output impedance.

9.6 Storing audio

■ 9.6.1 PLAIN – in simple words

1. The simplest way to store sound is to write every sample in order, with nothing clever done to it. That is **PCM**, pulse-code modulation.
2. A WAV file is mostly raw PCM with a small label saying how fast, how many bits and how many channels.
3. Raw PCM is honest and huge, so we compress it, in two different ways.
4. **Lossless** compression packs the data tighter and gives back every original number exactly. FLAC and ALAC do this, roughly halving the size.
5. **Lossy** compression throws information away permanently, choosing parts you are unlikely to notice. MP3, AAC and Opus do this, cutting by ten or more.
6. A **codec** is the method of coding and decoding. A **container** is the file format holding the coded data plus the title, artist and length.
7. The same codec can live in different containers, and one container can hold different codecs. That is the most confusing thing about audio files.

■ 9.6.2 PLAIN – a picture in your head

1. Imagine posting a long letter. Raw PCM is writing every word out in full.
2. Lossless compression is shorthand. The reader expands it back to the same letter word for word. Nothing lost, thinner envelope.
3. Lossy compression is writing a summary. The reader gets the meaning and never knows which words you dropped.
4. And you cannot un-summarize. Converting a summary back into a full letter just produces a fatter summary.
5. That is why converting an MP3 to FLAC restores nothing. It stores the damaged version losslessly.

Where this comparison breaks: a written summary drops whole ideas, which you would notice. A good audio codec drops only sounds physically masked by louder sounds happening at the same moment, which your ear could not have detected. It is closer to not printing the parts of a photograph that are behind a wall.

■ 9.6.3 PLAIN – a worked example

1. Step one: 44,100 samples per second.
2. Step two: 16 bits each, so $44,100 \times 16 = 705,600$ bits per second per channel.
3. Step three: stereo doubles it, giving 1,411,200 bits per second. That is the famous 1.411 Mbps.
4. Step four: divide by 8 for bytes, giving 176,400 bytes per second.
5. Step five: a three-minute song is 180 seconds, so $176,400 \times 180 = 31,752,000$ bytes, about 31.8 MB, shown as 30.3 MiB by a file manager.
6. The same song as a 128 kbps MP3 is $128,000 / 8 \times 180 = 2,880,000$ bytes, about 2.9 MB, roughly eleven times smaller.
7. As FLAC at a typical 60 percent, about 19 MB. As 64 kbps Opus, about 1.4 MB.

CD audio bitrate, from first principles

```

44100 samples/s
x   16 bits/sample      = 705600 bits/s per channel
x   2 channels          = 1411200 bits/s = 1.4112 Mbit/s
/   8                   = 176400 bytes/s
x 180 s (a 3 minute song) = 31752000 bytes
                        = 31.75 MB (30.28 MiB)

a full 74 minute CD:
176400 x 4440 s        = 783216000 bytes = 783 MB

```

■ 9.6.4 PLAIN – what is really happening inside

1. Lossless codecs work in two stages. First, prediction: guess the next sample from the previous few using a small polynomial or adaptive filter. Music is smooth, so the guess is usually close.
2. Second, store only the error between guess and truth. Those errors are small numbers clustered near zero, and they compress very well with a code that gives short bit patterns to common values.
3. Nothing is discarded, so decoding reproduces the original bit for bit.
4. Lossy codecs add a third idea: a model of your ear.
5. Fact one, the threshold of hearing: very quiet sounds are inaudible, and the level at which that happens depends strongly on frequency.
6. Fact two, simultaneous masking: a loud tone makes nearby quieter tones inaudible. A loud 1 kHz tone can hide a tone 40 dB quieter at 1.1 kHz.
7. Fact three, temporal masking: for a few milliseconds before, and up to about 200 milliseconds after a loud sound, quieter sounds near it are inaudible.
8. So the encoder splits the signal into frequency bands, works out how much noise each band can hide, and spends just enough bits per band to keep its own quantization noise under that hidden threshold.
9. The bits go where the ear is looking, and everywhere else gets almost nothing.
10. That is why 128 kbps MP3 is not “lower quality audio”. It is full-range audio with its errors deliberately parked underneath louder sounds.
11. When the bitrate is too low the encoder runs out of bits, errors climb above the masking threshold, and you hear it: swishy cymbals, a metallic ring on voices, and a hard cut-off of high frequencies.

■ 9.6.5 TECHNICAL – the engineer’s version

1. WAV is a container built on the Microsoft and IBM RIFF chunk format from 1991: a fmt chunk declares rate, channels and bits, a data chunk holds interleaved PCM. AIFF is Apple’s 1988 equivalent on the Electronic Arts IFF format, big-endian where WAV is little-endian.
2. MP3 is formally MPEG-1 Audio Layer III, from the Fraunhofer Institute for Integrated Circuits in Erlangen and the University of Erlangen-Nuremberg, with Karlheinz Brandenburg completing the underlying doctoral work in 1989. ISO/IEC 11172-3 reached committee draft in 1991, was finalized in 1992 and published in 1993. MPEG-2 Layer III, ISO/IEC 13818-3, added lower sample rates and was published in 1995.
3. The .mp3 extension was chosen by the Fraunhofer team on 14 July 1995; before that the files used .bit. The last US patent covering MP3 expired on 16 April 2017.
4. MP3 supports 32, 44.1 and 48 kHz under MPEG-1 and 16, 22.05 and 24 kHz under MPEG-2, with 14 fixed bitrates from 32 to 320 kbit/s. The 128 kbit/s convention is an 11:1 ratio against 1,411.2 kbit/s CD audio. Internally it uses a 32-band polyphase filter followed by an MDCT of 18 or 6 lines, non-uniform quantization, Huffman coding, and a bit reservoir.

5. AAC was standardized as MPEG-2 Part 7, ISO/IEC 13818-7, in 1997, then as MPEG-4 Part 3, ISO/IEC 14496-3, in 1999, developed jointly by AT&T Labs, Dolby, Fraunhofer IIS and Sony, with Nokia joining as co-licensor in 2002. It uses a pure MDCT filter bank with 2048 or 256 sample windows.
6. Opus was standardized by the IETF as RFC 6716 on 10 September 2012, merging SILK, a linear-prediction speech coder from Skype, with CELT, a low-latency MDCT coder from Xiph.Org. It spans 6 to 510 kbit/s with frames of 2.5 to 60 ms and a default algorithmic delay of 26.5 ms, reducible to 5 ms. It is the default codec for WebRTC.
7. FLAC reached version 1.0 on 20 July 2001, written by Josh Coalson, and typically compresses to 50 to 70 percent of the original. In December 2024 it was formally specified as IETF RFC 9639. ALAC is Apple's equivalent, introduced in 2004 and open sourced in 2011.
8. Codec versus container, precisely: a codec defines a bitstream syntax and a decoding process, while a container defines how coded streams, metadata, timestamps and index tables are packed into a file. MP4 and M4A are the same container, the ISO base media file format, and may hold AAC or ALAC. Ogg and WebM commonly hold Opus. The .mp3 case is unusual: a bare stream with no real container, which is why its metadata is a bolt-on called ID3.

Format	Type	Typical bitrate	Best use
PCM in WAV	None	1,411 kbps	Editing, mastering
FLAC	Lossless	700 to 1,000 kbps	Archiving music
ALAC	Lossless	700 to 1,000 kbps	Apple archiving
MP3	Lossy	128 to 320 kbps	Legacy compatibility
AAC	Lossy	96 to 256 kbps	Streaming, video, iOS
Opus	Lossy	6 to 128 kbps	Calls, WebRTC, chat

```
# what does this file actually contain?
ffprobe -hide_banner song.m4a
# container = mov,mp4,m4a   codec = aac   48000 Hz   stereo

# convert losslessly, keeping every sample
ffmpeg -i track.wav -c:a flac -compression_level 8 track.flac

# encode speech efficiently
opusenc --bitrate 32 --speech voice.wav voice.opus
```

The honest version: “MP3 removes frequencies above 16 kHz” is a symptom, not the method. Low-bitrate encoders often apply a lowpass because high frequencies are expensive and least missed, but that is a bit-allocation decision by one encoder, not part of the standard. Implementation detail: LAME, the dominant open-source MP3 encoder, and Fraunhofer's own encoder make different choices at the same bitrate, and LAME's variable-bitrate presets beat fixed 128 kbps.

■ 9.6.6 WORDS - remember these

1. PCM — the raw numbers — pulse-code modulation, uncompressed linear samples.
2. Codec — the coding method — the coder/decoder pair defining a bitstream.
3. Container — the box the data sits in — a file format multiplexing streams, metadata and timing.
4. Lossless — exactly what you put in — bit-exact reconstruction of the PCM.
5. Lossy — some detail gone for good — perceptually irrelevant information discarded permanently.
6. Masking — a loud sound hides a quiet one — simultaneous and temporal elevation of the hearing threshold.

7. MDCT — the maths inside every modern codec — modified discrete cosine transform with 50 percent overlapping windows.

9.7 How recorded sound was stored before computers

■ 9.7.1 PLAIN – in simple words

1. Before computers, sound was stored as a physical shape or a magnetic pattern.
2. The phonograph stored it as a wiggly groove cut into a spinning cylinder.
3. The groove is literally the waveform. You can see the sound under a microscope.
4. Magnetic tape stored it as magnetized specks on a plastic ribbon.
5. The vinyl record used a groove again, on a flat pressed disc.
6. The compact disc stored numbers instead, as microscopic dents read by a laser.
7. Only the last is digital. The first three are direct physical copies of the wave.

■ 9.7.2 PLAIN – a picture in your head

1. Picture drawing a wobbly line along a very long road with chalk.
2. Later, walk the same road dragging your finger along the chalk line.
3. Your finger is forced to wobble exactly as the line wobbles.
4. Connect the finger to a drum skin, and the skin wobbles, and the air wobbles, and you hear the original sound.
5. That is a record player. No code, no numbers, no cleverness. The groove is a drawing of the pressure wave, and the needle is pushed by it.

Where this comparison breaks: a record groove wobbles sideways, and a stereo groove wobbles both walls independently at 45 degrees. Records are also not cut flat: bass is reduced and treble boosted when cutting, then reversed on playback, so grooves stay narrow and surface noise falls. That curve is the RIAA equalization standard from 1954.

■ 9.7.3 PLAIN – a worked example

1. Take a 33 and a third rpm LP, 12 inches across.
2. It turns once every 1.8 seconds, so the outer groove passes the needle at about 50 cm per second and the inner groove at only about 21 cm per second.
3. That is why the last track on a side often sounds slightly worse. Less groove length is available per second of music.
4. A side holds about 25 minutes, and the microgroove standard allows a stylus tip radius of 25 micrometres.
5. A CD instead runs at constant linear velocity, 1.2 to 1.4 m/s, so it spins faster in the middle and slower at the edge, and every part of the surface carries data at the same density.

■ 9.7.4 PLAIN – what is really happening inside

1. Magnetic tape is plastic ribbon coated with billions of needle-shaped iron oxide particles.
2. Each particle is a tiny permanent magnet holding whichever direction you last pushed it into.
3. The record head is an electromagnet with a very narrow gap, and audio current through it makes a field at that gap.
4. Tape passes the gap, and particles leaving the field freeze in whatever magnetization the field last gave them: strongly aligned for a loud moment, weakly aligned for a quiet one.
5. On playback the moving magnetized tape passing another gap induces a voltage in a coil, by Faraday's law, the same law as the microphone.

6. Hold this for Chapter 12: a hard disk is the same idea with the tape replaced by a rigid spinning platter, a head flying nanometres above it, and the magnetization standing perpendicular to the surface rather than along it. The physics of “particles keep the direction you last pushed them” is unchanged.
7. The compact disc is different, because it stores numbers rather than a wave.
8. A spiral track of pits is pressed into plastic and coated with reflective aluminium. A laser shines up at it and a photodiode watches the reflection.
9. A pit is about a quarter wavelength deep, so light from a pit edge cancels light from the flat land beside it and the brightness drops sharply.
10. The honest version: a pit is not a 1 and a land is not a 0. Every transition from pit to land or land to pit is a 1, and no transition is a 0. That coding is called NRZI, and it exists because an edge is far more reliably detected than an absolute level.

■ 9.7.5 TECHNICAL – the engineer’s version

1. Edouard-Leon Scott de Martinville patented the phonautograph on 25 March 1857. It traced sound onto soot-blackened paper but could not play back. The oldest recovered recording from it, of “Au Clair de la Lune”, dates from 9 April 1860 and was recovered digitally in 2008.
2. Thomas Edison announced the phonograph on 21 November 1877 and demonstrated it on 29 November 1877, recording on tinfoil wrapped round a cylinder.
3. Emile Berliner moved to flat discs, beginning commercial gramophone production in 1892 with five-inch records and moving to shellac in 1895.
4. Valdemar Poulsen demonstrated magnetic recording on steel wire at the end of the 1890s. Fritz Pfleumer patented magnetic tape in Germany in 1928, AEG built it into the Magnetophon with BASF tape in the mid-1930s, and American engineers, notably Jack Mullin, brought captured machines home after the Second World War, after which Ampex productionized them.
5. Columbia Records introduced the 33 and a third rpm microgroove LP at a press conference on 21 June 1948. Stereo LPs followed from 1957.
6. Compact Disc Digital Audio, the Red Book, was published by Philips and Sony in 1982 and adopted as IEC 60908 in 1987. The first commercial CD release was on 17 August 1982, and the first fifty titles went on sale in Japan on 1 October 1982.
7. EFM, devised by Kees Schouhamer Immink, maps each 8-bit byte to a 14-bit pattern chosen so runs of zeros are between 2 and 10 long, keeping pit lengths within what the optics resolve while leaving enough transitions for clock recovery. Three merging bits join the patterns.
8. CIRC is Sony’s cross-interleaved Reed-Solomon code. Interleaving spreads consecutive samples across the disc so a scratch destroys one symbol from many codewords rather than many symbols from one, which is exactly what Reed-Solomon repairs. It fully corrects bursts of about 4,000 bits, roughly 2.5 mm of track.

CD specification	Value
Disc diameter	120 mm
Laser wavelength	780 nm, infrared
Track pitch	1.6 micrometres
Maximum playing time	74 min 33 s
Channel coding	EFM, eight-to-fourteen
Error correction	CIRC, Reed-Solomon

■ 9.7.6 WORDS – remember these

1. Groove — the wiggly channel in a record — a mechanically modulated spiral carrying the wave-form laterally.
2. Stylus — the needle — the tip tracing the groove, typically 25 μm radius on microgroove records.
3. Coercivity — how hard a magnet is to flip — the reverse field needed to demagnetize a recorded particle.
4. Pit and land — the dents and the flat between them — features whose transitions encode 1 bits under NRZI.
5. EFM — how bytes become pit patterns — eight-to-fourteen modulation with a run-length constraint of 2 to 10.
6. CIRC — the CD's scratch repair — cross-interleaved Reed-Solomon coding.
7. RIAA curve — the record equalization standard — the 1954 pre-emphasis and playback de-emphasis specification.

9.8 Audio inside a computer

■ 9.8.1 PLAIN – in simple words

1. Inside a computer, sound is a stream of numbers moving to a deadline.
2. A chip called the audio codec holds the converters, in both directions.
3. Your program writes numbers into a shared area of memory called a **buffer**, and the hardware reads from it at a fixed, unforgiving rate.
4. If your program is late even once, the hardware runs out of numbers and plays silence or repeats old data. That gap is heard as a click.
5. So the whole design problem of computer audio is: never be late, ever.
6. A bigger buffer gives more safety margin but more delay before you hear anything. That delay is **latency**.

■ 9.8.2 PLAIN – a picture in your head

1. Think of a conveyor belt feeding a machine that must never stop.
2. You stand at one end putting parts on. The machine takes one every second, forever.
3. A long belt lets you wander off for a minute, but a part you place now is not used for a long time.
4. A short belt means what you place is used almost immediately, which is what you want when playing a guitar through the computer, but blink at the wrong moment and the machine starves and jams.
5. Length buys safety, shortness buys responsiveness, and you cannot have both.

Where this comparison breaks: the computer's belt is circular. It is a ring buffer, written at one point and read at another chasing it round. Nothing is added to an end, and if the reader catches the writer you get the click.

■ 9.8.3 PLAIN – a worked example

1. Buffering latency = frames in the buffer divided by the sample rate.
2. At 48,000 samples a second, a 256-frame buffer is $256 / 48000 = 5.33$ milliseconds.
3. But there are normally at least two such periods, one playing and one being filled, so real output latency is around 10.7 ms.
4. Add the input side for live monitoring and you are near 21 ms.
5. Sound travels 343 metres a second, so 21 ms is the delay of standing 7.2 metres from an amplifier. Musicians notice about 10 ms and hate 25 ms.

- Drop to a 64-frame buffer and each period is 1.33 ms. Beautiful, and it will crackle on a busy machine.

Buffer size	At 48 kHz	Typical use
32 frames	0.67 ms	Live guitar, low latency
128 frames	2.67 ms	Studio recording
512 frames	10.7 ms	Mixing, general use
2048 frames	42.7 ms	Video playback, games

■ 9.8.4 PLAIN – what is really happening inside

- The codec chip contains ADCs, DACs, a mixer, headphone and speaker amplifiers, and often a microphone bias supply.
- It talks to the processor over a small serial bus: I2S or TDM for the audio samples, I2C or SoundWire for control commands.
- A DMA engine, which is hardware that moves memory without troubling the CPU, walks round the ring buffer handing samples to the codec on a timer.
- When it finishes a period it raises an interrupt, the driver wakes the audio thread, and that thread must fill the next period before the engine arrives.
- If it does not, that is an **underrun**, called an **xrun** in ALSA.
- Why does an underrun click rather than just go quiet? Because the waveform jumps instantly to zero, and a vertical step contains energy at every frequency at once. Your ear hears a broadband snap.
- Sample rate conversion is the other everyday complication: your file is 44.1 kHz and your device runs at 48 kHz, so somebody must resample.
- The ratio is 160 to 147, so good resampling needs a long filter and costs CPU, while bad resampling adds audible aliasing.
- Above the driver sits a mixer, because many applications want the one device at once. It resamples everything to a common rate, applies per-stream volume, sums, and hands a single stream to the hardware.

■ 9.8.5 TECHNICAL – the engineer's version

- Historical anchors: the Creative Labs Sound Blaster arrived in 1989 and set the PC convention for a decade. Intel's AC'97 specification came in 1997 and split the digital controller from the analogue codec. Intel High Definition Audio, codenamed Azalia, replaced it in 2004 and remains the PC standard, usually paired with a Realtek ALC-series codec.
- WASAPI has two modes. Shared mode goes through the Windows audio engine, is resampled to a fixed device format and mixes with other applications. Exclusive mode hands the device to one application bit-perfectly. ASIO bypasses the Windows engine entirely, which is why it survives in professional work.
- On Linux, ALSA is the kernel driver interface. PulseAudio, from 2004, added per-application volume and network transparency but had poor low-latency behaviour, while JACK served professional audio. PipeWire replaced both with one graph-based server, becoming the default in Fedora 34 in April 2021 and in Ubuntu 22.10 in October 2022.
- Core Audio expresses everything as float32 at the device rate, and is the only one of the three major platforms where the low-latency path is also the ordinary path.
- Rule of thumb for round-trip latency: buffer periods, plus converter group delay (sigma-delta decimation and interpolation filters add roughly 0.5 to 1.5 ms each way), plus any resampling

delay. Manufacturers quote only the buffer figure, so measured round-trip is always worse than the control panel.

Stack	Platform	Note
Core Audio	macOS and iOS	Since Mac OS X 10.0, 2001
WASAPI	Windows	Since Vista, 2007
ASIO	Windows	Steinberg, 1997, pro use
ALSA	Linux kernel	Kernel driver layer
PipeWire	Linux userspace	Fedora 34 default, 2021

```
# Linux: what hardware exists, and what it is doing now
aplay -l
cat /proc/asound/card0/pcm0p/sub0/hw_params
pw-top           # live quantum, rate and xrun counts

# macOS
system_profiler SPAudioDataType
```

■ 9.8.6 WORDS – remember these

1. Buffer — the waiting area for samples — a ring buffer read by DMA on a fixed timer.
2. Latency — the delay before you hear it — frames divided by sample rate, times the number of periods, plus converter delay.
3. Underrun / xrun — the moment audio runs out — the DMA pointer overtaking the application's write pointer.
4. Codec chip — the audio front end on the board — integrated ADC, DAC, mixer and amplifiers on an I2S/I2C interface.
5. Resampling — changing the sample rate — polyphase interpolation and decimation, 147:160 between 44.1 and 48 kHz.
6. Exclusive mode — one application owns the device — bit-perfect path bypassing the system mixer, WASAPI exclusive or ASIO.

9.9 Phone calls, part 1 – the old way

■ 9.9.1 PLAIN – in simple words

1. The original telephone is astonishingly simple.
2. A microphone at one end changes the electric current in a wire, following the sound.
3. That same current runs down the wire through a small speaker at the far end, which moves and makes the sound again.
4. No code, no numbers, no computer. The wire carries a shrinking and growing electrical copy of the pressure wave.
5. The pair of copper wires from your house to the exchange is the **local loop**.
6. To connect any phone to any other, you go through an **exchange**, a building whose whole job is joining wires together.
7. At first humans did the joining, plugging cords into sockets. Later machines did it, driven by the clicks your dial sent down the line.
8. The crucial point: while you talked, a continuous electrical path existed from your house to theirs, and nobody else could use it. That is **circuit switching**.

■ 9.9.2 PLAIN – a picture in your head

1. Think of the old phone network as a railway that gives you a private track.
2. To make a call, the railway builds a track from your station to theirs, and it stays yours for the whole call, even during silences.
3. When you hang up, the track is dismantled and the pieces return to the pool.
4. This is wonderful for quality. Nothing can get in your way, so the delay never changes and the sound never breaks up.
5. It is terrible for efficiency. About half of a conversation is silence, and all that track sits idle.

Where this comparison breaks: after the 1960s the private track stopped being a continuous piece of copper. Your voice was digitized at the exchange and given a fixed repeating slot on a shared high-speed line. It behaves exactly like a private track, with guaranteed capacity and constant delay, but physically it shares one cable with dozens of other calls.

■ 9.9.3 PLAIN – a worked example

1. Here is how a phone call became exactly 64,000 bits a second.
2. The telephone band is 300 Hz to 3,400 Hz, chosen to keep speech intelligible while using as little of the wire as possible.
3. To sample up to about 4,000 Hz you need at least 8,000 samples a second, so 8 kHz was chosen, and each sample is stored in 8 bits.
4. $8,000 \times 8 = 64,000$ bits per second. That is one voice channel, called a DS0.
5. Why is 8 bits acceptable when a CD needs 16? Because of **companding**.
6. Instead of 256 evenly spaced levels, the steps are fine near zero and coarse near the loudest values, following a logarithmic curve.
7. Quiet passages get fine resolution where you would notice the noise, and loud passages get coarse steps where loudness hides them.
8. An 8-bit companded sample carries roughly the useful range of a 12 to 13-bit linear sample.

```
one voice channel, from air to bits

speech -> 300-3400 Hz filter -> sample at 8 kHz
        -> compand to 8 bits -> 64000 bit/s (one DS0)

T1 = 24 DS0 + 1 framing bit
    = 193 bits every 125 us = 1.544 Mbit/s

E1 = 32 timeslots x 8 bits
    = 256 bits every 125 us = 2.048 Mbit/s
    (TS0 framing, TS16 signalling, 30 voice channels)
```

■ 9.9.4 PLAIN – what is really happening inside

1. The original transmitter was the **carbon microphone**: a chamber of carbon granules behind a diaphragm.
2. Squeeze the granules and they touch better, so resistance falls. Release them and resistance rises.
3. A battery pushes current through the granules, so sound modulates a current far stronger than the sound itself.
4. That makes the carbon microphone an amplifier as well as a microphone, which is why telephones worked over kilometres of wire decades before the electronic amplifier existed. It is the single reason early telephony was commercially possible.

5. The exchange supplies power down your own line, which is why a landline kept working when the house lost electricity.
6. Automatic switching started with an undertaker's grudge. Almon Brown Strowger believed an operator was diverting calls to a rival and patented an automatic exchange in 1891.
7. His **step-by-step** switch used the dial pulses directly. The first digit's pulses ratcheted a contact arm up one of ten rows, and the second digit's pulses rotated it to one of ten contacts in that row. Ten by ten is a hundred lines, and switches were stacked in stages for longer numbers.
8. That is why old dials sent pulses, and why "dialling" is still the word.
9. Signalling, meaning the instructions about a call rather than the call itself, originally travelled in the same audio path as the voice, as tones.
10. That was elegant and disastrous: anyone able to play the right tone into a handset could instruct the network. Out-of-band signalling replaced it, so call control now travels on its own separate data network.

a 1960s long-distance call, end to end

```
handset -> local loop (copper pair, -48 V from exchange)
        -> local exchange (switch)
        -> channel bank: filter, sample, compand -> DS0
        -> multiplexed into a T1/E1 with other calls
        -> trunk exchanges, path reserved end to end
        -> demultiplex -> DAC -> local loop -> handset
```

the path is held for the duration of the call,
whether anyone is speaking or not.

■ 9.9.5 TECHNICAL – the engineer's version

1. Alexander Graham Bell was granted US patent 174,465 on 7 March 1876, for "Improvement in Telegraphy", covering transmission of vocal sounds by electrical undulations. The first intelligible sentence followed on 10 March 1876. Elisha Gray filed a caveat on 14 February 1876, and Antonio Meucci had filed caveat 3335 in 1871, which lapsed in 1874. The priority dispute has never fully died down.
2. Edison filed for a carbon transmitter on 27 April 1877; US patent 222,390 was granted in 1879.
3. Strowger's automatic exchange patent dates from 1891, and the first installation opened in La Porte, Indiana, on 3 November 1892 with about 75 subscribers. Crossbar switches displaced step-by-step from the late 1930s, and stored-program electronic switching from the mid-1960s.
4. Local loop electrical figures, which are conventions of the North American and most European networks rather than one global standard: -48 V DC central office battery, 20 to 50 mA loop current off-hook, ringing at about 90 V RMS at 20 Hz in North America and 25 Hz in much of Europe. Loops are typically 24 or 26 AWG copper, up to roughly 5 km without loading coils.
5. G.711, released by the CCITT in 1972, defines the 8 kHz, 8-bit, 64 kbit/s voice channel. A-law compands a 13-bit signed linear sample to 8 bits and is used in most of the world; mu-law compands 14 bits to 8 and is used in North America and Japan. Algorithmic delay is 0.125 ms with no look-ahead.
6. Multiplexing: the North American DS1 (T1) carries 24 DS0s plus one framing bit in a 193-bit frame every 125 microseconds, giving 1.544 Mbit/s. The CEPT E1 carries 32 timeslots of 8 bits in a 256-bit frame every 125 microseconds, giving 2.048 Mbit/s, with timeslot 0 for framing and timeslot 16 usually for signalling, leaving 30 voice channels.
7. SS7, Signalling System No. 7, grew out of Bell System work in the 1970s and was standardized by the ITU-T as the Q.700-series recommendations in 1980. It moved call control onto a separate

packet network. Links run at 56 or 64 kbit/s, with high-speed links at 1.5 or 2.0 Mbit/s. ISUP sets up and tears down voice circuits, and SIGTRAN carries SS7 over IP using SCTP.

8. SS7 was designed for a closed club of national carriers and has essentially no authentication between operators. Since 2014, published research and reported incidents have demonstrated subscriber location tracking and SMS interception, which is the concrete reason SMS is now considered a weak second factor for authentication.

■ 9.9.6 WORDS – remember these

1. Local loop — the pair of wires to your house — the subscriber line from premises to the serving exchange.
2. Circuit switching — a reserved path for the whole call — end-to-end resource allocation held for the call duration.
3. Companding — squeezing loud and stretching quiet — logarithmic quantization per ITU-T G.711 A-law or mu-law.
4. DS0 — one voice channel — 64 kbit/s, 8 kHz at 8 bits.
5. TDM — sharing one cable by taking turns — time-division multiplexing into fixed 125 microsecond frames.
6. SS7 — the network's own instruction channel — out-of-band common channel signalling, ITU-T Q.700 series.
7. Voice band — the slice of hearing a phone carries — 300 to 3,400 Hz.

9.10 Phone calls, part 2 – mobile and internet

■ 9.10.1 PLAIN – in simple words

1. A mobile phone cannot send a wave down a wire, so it sends numbers by radio.
2. Radio time is scarce and expensive, so those numbers must be squeezed hard. A digital landline used 64,000 bits a second per call, and early mobile used about 13,000.
3. To get that small, mobile phones do not send the sound at all.
4. They send a description of how your throat and mouth were shaped, updated fifty times a second, and the far end builds a fresh voice from that recipe.
5. That is why mobile voice sounds correct but slightly synthetic, and why music down a phone sounds ruined. The recipe assumes a human voice.
6. Internet calls, such as WhatsApp or a work meeting, work differently again.
7. They chop the sound into small parcels of about twenty milliseconds, address each one, and throw it onto the ordinary internet.
8. Nobody reserves anything. The parcels take their chances with everyone else's video, downloads and games.
9. Most arrive. Some arrive late. Some never arrive at all.

■ 9.10.2 PLAIN – a picture in your head

1. The old phone call was a private railway track, held for you alone.
2. An internet call is posting numbered postcards, one every twenty milliseconds, and hoping.
3. Postcards may arrive out of order, bunched up, or not at all.
4. So the receiver keeps a small tray. Postcards land in it, get sorted, and are read out at a perfectly steady rhythm from the tray rather than as they arrive. That tray is the **jitter buffer**.
5. If a postcard is missing when its turn comes, the receiver does not wait. It invents a plausible twenty milliseconds from what came before.

- Asking for a re-send would be pointless. By the time the replacement arrived, that moment of the conversation would be long past.

Where this comparison breaks: postcards are independent, but audio frames are not. Modern codecs predict each frame from the last, so a lost frame degrades the next few as well. Opus reduces this by optionally embedding a low-quality copy of the previous frame inside each new one, a form of forward error correction.

■ 9.10.3 PLAIN - a worked example

- Cost out a plain internet call using G.711, the old 64 kbps telephone codec, over IPv4.
- We send one packet every 20 milliseconds, so 50 packets a second.
- 20 ms of G.711 is 8,000 samples a second times $0.02\text{ s} = 160$ samples of one byte each, so 160 bytes of audio.
- Now the addressing: RTP header 12 bytes, UDP header 8 bytes, IPv4 header 20 bytes. That is 40 bytes of overhead.
- Total 200 bytes per packet, so $200 \times 8 \times 50 = 80,000$ bits per second. A 64 kbps codec costs 80 kbps on the wire, and a quarter of the traffic is envelopes.
- Repeat with Opus at 24 kbps and 20 ms frames: 60 bytes of audio plus the same 40 bytes of headers = 100 bytes, so 40 kbps on the wire.
- The fixed 40 bytes dominates at low bitrates, which is why voice codecs settled on 20 ms frames. Shorter frames would multiply the header cost.

Codec	Payload bitrate	On the wire, IPv4
G.711 (64 kbps)	64.0 kbps	80.0 kbps
G.729 (8 kbps)	8.0 kbps	24.0 kbps
AMR-NB (12.2 kbps)	12.2 kbps	28.2 kbps
Opus (24 kbps)	24.0 kbps	40.0 kbps

■ 9.10.4 PLAIN - what is really happening inside

- A 2G GSM phone runs the GSM Full Rate codec: 13 kbit/s, working on frames of 160 samples at 8 kHz, which is 20 milliseconds.
- It is a vocoder. It fits a mathematical model of the vocal tract to your speech, then transmits the model's settings and a description of the buzz or hiss driving it, rather than the waveform itself.
- That is why cellular voice sounds thin: band-limited to about 3.4 kHz, and then re-synthesized rather than reproduced.
- Setting up an internet call needs two separate things, and confusing them is the classic beginner error.
- Signalling** is the conversation about the call: who you want, are they there, which codecs can we both do, are they ringing, have they answered. SIP does this.
- Media** is the actual sound. RTP does this, over UDP, which never retransmits anything.
- Why UDP and not TCP? Because TCP guarantees delivery by retransmitting, and waiting for a retransmission is worse than the loss it fixes.
- So voice deliberately chooses a protocol that gives up on lost data, then repairs the hole locally.
- Packet loss concealment** extends the last pitch period and fades it over a few tens of milliseconds. One or two lost packets are genuinely hard to hear.
- Echo** is the other big problem. Your speaker plays my voice, your microphone picks it up, and I hear myself two hundred milliseconds later, which is far more disruptive than noise.

11. **Acoustic echo cancellation** works because the device already knows exactly what it sent to its own speaker. It learns a filter describing the room path from speaker to microphone, predicts the echo, and subtracts it.
12. That is why speakerphone in a bathroom is hard. The room path is long, changing and full of reflections, so the filter cannot keep up.

■ 9.10.5 TECHNICAL – the engineer’s version

1. AMR-NB frames are 160 samples at 8 kHz, 20 ms long, and the network can switch mode frame by frame to trade voice bits against error protection as radio conditions change.
2. AMR-WB samples at 16 kHz with a 50 to 7,000 Hz passband and was developed by Nokia and VoiceAge. It is what carriers market as HD Voice. EVS, Enhanced Voice Services, extends this to superwideband and fullband.
3. SIP is defined in RFC 3261 (2002), replacing RFC 2543 (1999). It is a text request/response protocol resembling HTTP, on port 5060 plain and 5061 over TLS. Core methods are INVITE, ACK, BYE, CANCEL, REGISTER and OPTIONS. Media parameters are negotiated by carrying SDP bodies in an offer/answer exchange.
4. RTP is defined in RFC 3550 (2003), replacing RFC 1889 (1996). Its 12-byte header carries a 7-bit payload type, a 16-bit sequence number for loss and reorder detection, a 32-bit media timestamp, and a 32-bit synchronization source identifier. RTCP reports loss, jitter and round-trip time to the sender.
5. Jitter buffers are adaptive: they measure arrival variation and resize themselves, typically holding 20 to 100 ms, trading added delay against the probability that a packet arrives too late to use.
6. ITU-T Recommendation G.114 gives the delay budget: under 150 ms one-way mouth-to-ear is acceptable for essentially all applications, 150 to 400 ms is usable but degraded, and over 400 ms is unacceptable for conversation. Line echo cancellers are specified in ITU-T G.168.
7. VoLTE carries voice as IP over an LTE bearer, controlled by IMS, the IP Multimedia Subsystem, rather than by a circuit-switched core. The first commercial launch was by MetroPCS in Dallas in August 2012. Its media plane uses AMR-NB as a minimum, AMR-WB for HD Voice, and EVS in newer deployments.
8. The key architectural difference from an over-the-top application: VoLTE media rides a dedicated bearer with a guaranteed bit rate and a strict quality-of-service class. A WhatsApp call is best-effort traffic on the same data bearer as everything else, is end-to-end encrypted by the application, uses Opus, and crosses the operator’s network as ordinary internet data. Both are voice over IP. Only one has the network holding capacity open.
9. Voice tolerates loss and hates delay because it is an isochronous stream consumed in real time: a 20 ms hole is concealed, but a 400 ms delay breaks turn-taking, which humans manage on a scale of about 200 ms. File transfer is the exact opposite. It is verified in full at the end, so one wrong byte ruins it, while an extra thirty seconds costs nothing. That is why one runs on UDP with concealment and the other on TCP with retransmission.

Codec	Bitrate	Standardized
GSM Full Rate (RPE-LTP)	13 kbit/s	Early 1990s
GSM Half Rate	5.6 kbit/s	1990s
GSM Enhanced Full Rate	12.2 kbit/s	1990s
AMR-NB, 8 modes	4.75 to 12.2 kbit/s	3GPP, Oct 1999
AMR-WB / G.722.2	6.6 to 23.85 kbit/s	ITU-T, 2002

a SIP call setup, simplified

```

caller                                callee
|----- INVITE (with SDP offer) --->|
|<----- 100 Trying -----|
|<----- 180 Ringing -----|
|<----- 200 OK (with SDP answer) ---|
|----- ACK ----->|
|=== RTP audio over UDP, both ways =|
|----- BYE ----->|
|<----- 200 OK -----|

```

■ 9.10.6 WORDS – remember these

1. Vocoder — a codec that rebuilds your voice rather than copying it — a parametric coder transmitting vocal tract model coefficients.
2. RTP — the protocol carrying the sound — Real-time Transport Protocol, RFC 3550, over UDP.
3. SIP — the protocol that rings the phone — Session Initiation Protocol, RFC 3261, with SDP offer/answer.
4. Jitter buffer — the tray that smooths arrival times — an adaptive de-jitter queue trading latency for late-packet loss.
5. PLC — filling in a lost moment — packet loss concealment by pitch-period extrapolation.
6. AEC — stopping the caller hearing themselves — acoustic echo cancellation by adaptive filtering of the known playback signal.
7. VoLTE — carrier calls over the data network — IMS-controlled voice on a dedicated LTE bearer with guaranteed bit rate.
8. Best effort — no promises about delivery — the default internet service model with no reserved capacity.

9.11 Noise cancelling, spatial audio and the modern tricks

■ 9.11.1 PLAIN – in simple words

1. There are two ways to make the world quieter in headphones.
2. **Passive** means physically blocking sound: a tight seal, heavy cups, foam. It is just a wall.
3. **Active** means fighting sound with sound.
4. A microphone on the outside listens to the noise arriving, and a chip works out the exact opposite wave: where the noise pushes, the opposite pulls.
5. The speaker plays that opposite. Push and pull arrive together and cancel.
6. This works well for low rumbling noise such as an aircraft or a bus engine, and poorly for sharp high noise such as speech or a fork on a plate.
7. Separately, phones use several microphones at once to work out which direction a sound came from and keep only the direction you speak from. That is **beamforming**.
8. And headphones can fool you into hearing a sound as if it came from your left, above, or behind, by copying the way your own head would have changed it.

■ 9.11.2 PLAIN – a picture in your head

1. Imagine two people pushing a swing.
2. If they push together at the right moments, the swing goes higher. That is adding two waves in phase.
3. If the second pushes exactly when the first pulls, with equal strength, the swing does not move at all. That is cancellation.

4. Noise cancelling is the second person, and the chip's whole job is getting the timing exactly right.
5. To push at exactly the wrong moment, you must know when that moment is before it happens. For a slow swing that is easy. For one wobbling twenty thousand times a second, the chip has microseconds to hear, calculate and play.

Where this comparison breaks: cancellation is only exact at one point in space. Move your head a few centimetres and the arrival times change, so what cancelled now partly adds. At low frequencies the wave is metres long, so a few centimetres hardly matter, which is the real reason active cancellation is a low-frequency technique.

■ 9.11.3 PLAIN – a worked example

1. Take a 100 Hz engine rumble. Its wavelength is 3.43 metres.
2. To cancel it you must be right within a small fraction of that, say 10 cm, which is 3 percent of a wavelength, about 10 degrees of phase error.
3. 10 cm of air is 0.29 milliseconds, comfortable for a chip running millions of operations a second.
4. Now take a 5,000 Hz hiss. Its wavelength is 6.9 cm.
5. The same 10 cm of positional error is now more than a whole wavelength. Your cancellation arrives at some arbitrary phase and may make it louder.
6. That single calculation is why every noise cancelling headphone works well below about 1 kHz and relies on the ear cup seal above it.
7. Beamforming numbers: two phone microphones 14 cm apart. Sound from directly ahead reaches both at the same instant, while sound from the side arrives $0.14 / 343 = 408$ microseconds later at the far microphone. Delay one microphone by 408 microseconds and add, and side sounds partly cancel while front sounds double.

■ 9.11.4 PLAIN – what is really happening inside

1. Active noise cancellation comes in three arrangements.
2. **Feedforward**: a microphone outside the cup hears the noise before it reaches your ear, so there is time to compute. It cannot tell what actually arrived at the eardrum.
3. **Feedback**: a microphone inside the cup hears the result, including whatever the headphone did, and corrects the error. It has almost no time, so it works only at low frequencies.
4. **Hybrid**: both. Nearly all current premium headphones do this.
5. Now spatial audio. Your brain locates sounds with three main clues.
6. **Interaural time difference**: sound from your left reaches the left ear first, by up to about 660 microseconds.
7. **Interaural level difference**: your head shadows the far ear, mostly at high frequencies where the head is large compared with the wavelength.
8. **Spectral shaping**: the folds of your outer ear boost and notch particular frequencies depending on whether a sound comes from above, in front or behind. That is what resolves up from down, which timing alone cannot.
9. All three together, measured as a filter for every direction, are the **head-related transfer function**, or HRTF.
10. Headphone spatial audio applies that filter to a sound, so your brain receives the clues it would have received from a real source.
11. Head tracking makes it far more convincing, because when you turn your head the virtual source stays put, exactly as a real one would.

■ 9.11.5 TECHNICAL – the engineer’s version

1. Paul Lueg was granted US patent 2,043,416 in 1936 for cancelling sinusoidal tones in a duct by phase advance. Lawrence Fogel patented cockpit noise cancellation in the 1950s. Bose shipped an active noise cancelling aviation headset in 1989 and the consumer QuietComfort headphone in 2000.
2. Practical performance for current premium over-ear models: roughly 20 to 30 dB of active attenuation below 300 Hz, tapering to near zero by 1.5 to 2 kHz, on top of 10 to 25 dB of passive attenuation above 1 kHz. Manufacturers quote the sum and rarely publish the curve.
3. Feedforward controllers are typically fixed or slowly adaptive FIR filters. Feedback loops are analogue or very low-latency digital, and are limited by the Bode sensitivity integral, which guarantees that attenuation at one frequency must be paid for with amplification at another. That is why poorly designed cancellation adds a faint hiss or a pressure sensation around 2 to 4 kHz.
4. Beamforming: delay-and-sum is the simplest form. Adaptive methods, notably minimum variance distortionless response, steer nulls at interfering sources instead. A two-microphone phone array gives modest directivity, while laptop and smart-speaker arrays of four to eight microphones do far better.
5. Channel layouts, as a standard: 2.0 stereo; 5.1, meaning left, centre, right, surround left, surround right and a low-frequency effects channel limited to about 120 Hz; 7.1; and object-based systems such as Dolby Atmos, introduced in 2012, which transmit audio objects with position metadata and render them to whatever speakers exist.
6. HRTF datasets used in research include the CIPIC database and the MIT KEMAR measurements. Rendering convolves source audio with the left and right HRTF impulse responses for the wanted direction, interpolating between measured points.
7. Active research, not settled fact: individualized HRTFs. Generic HRTFs give weaker elevation cues and more front-back confusion for listeners whose ear geometry differs from the measurement dummy. Approaches under investigation include photogrammetry from phone cameras, anthropometric regression and machine-learned personalization. Vendor claims of personalized spatial audio from a short ear scan are an engineering approximation, not a solved problem.

■ 9.11.6 WORDS – remember these

1. Anti-phase — the exact opposite wave — a signal inverted by 180 degrees at the frequency of interest.
2. Feedforward ANC — the outside microphone — control derived from the disturbance before it reaches the ear.
3. Feedback ANC — the inside microphone — closed-loop correction of the residual error at the ear.
4. Beamforming — listening in one direction — spatial filtering across a microphone array by delay-and-sum or adaptive weights.
5. ITD and ILD — which ear hears it first and loudest — interaural time and level differences, the primary horizontal localization cues.
6. HRTF — how your own ears colour a sound by direction — head-related transfer function, the direction-dependent filter of head, torso and pinna.
7. Object audio — sounds with positions rather than channels — metadata-driven rendering to an arbitrary speaker layout.

9.98 Common wrong ideas

1. Wrong: higher bitrate always sounds better. Right: bitrate matters only until the codec has enough bits to hide its errors under the masking threshold. Beyond that, extra bits change nothing audible, and a good encoder at 96 kbps AAC can beat a poor encoder at 256 kbps MP3.
2. Wrong: more watts means louder. Right: loudness follows speaker sensitivity far more than amplifier power. Doubling power adds only 3 dB, while a speaker 6 dB more sensitive is as loud on a quarter of the power.
3. Wrong: lossless always sounds audibly better than lossy. Right: lossless is guaranteed bit-identical, which matters for archiving, editing and re-encoding. In blind listening at modern high bitrates, most listeners cannot reliably tell them apart. Keep lossless because it is future-proof, not because you can hear it.
4. Wrong: the microphone records the sound. Right: the microphone only converts pressure into a voltage. Nothing is recorded until something samples that voltage and stores numbers. In a condenser or MEMS microphone the sound does not even supply the signal energy; it modulates a bias already present.
5. Wrong: digital audio is a staircase, so it is less smooth than analogue. Right: the staircase is an intermediate artefact inside the converter. The reconstruction filter recovers the unique smooth band-limited wave passing through the samples, so there is no jaggedness in the output.
6. Wrong: MP3 works by cutting off high frequencies. Right: that is one bit-allocation decision some encoders make. The mechanism is shaping quantization noise to hide beneath louder sounds across the whole spectrum.
7. Wrong: noise cancelling headphones cancel all noise. Right: they work well below roughly 1 kHz, because cancellation must be accurate to a fraction of a wavelength. Higher frequencies are blocked by the physical seal instead.
8. Wrong: a WhatsApp call and a mobile phone call are the same thing on the same network. Right: both are voice over IP, but a carrier VoLTE call rides a dedicated bearer with guaranteed bit rate under IMS control, while an application call is best-effort internet traffic with no reserved capacity.
9. Wrong: a bigger speaker is always better. Right: a large cone can move enough air for bass but is too heavy to reproduce treble accurately, which is why real systems split the range between drivers sized for each job.
10. Wrong: 44.1 kHz means the system can handle only 44,100 different sounds. Right: it means 44,100 measurements per second, which by the sampling theorem perfectly captures every frequency below 22,050 Hz.

9.99 Chapter summary in 20 lines

1. Sound is a travelling pattern of air pressure: compressions and rarefactions.
2. Frequency in hertz sets pitch, amplitude sets loudness, and wavelength is the speed of sound, about 343 m/s at 20 C, divided by frequency.
3. Decibels are logarithmic because hearing responds to ratios: +6 dB doubles pressure and +10 dB multiplies power by ten.
4. A speaker is a coil in a magnet's gap glued to a cone. Current makes force by $F = Bl \cdot I$, the cone moves, the air moves, and you hear it.
5. Faraday's law runs the same machine backwards, which is why a moving cone generates voltage and why a dynamic microphone needs no power.
6. Big cones move enough air for bass but are too heavy for treble, so systems split the band with crossovers, and enclosures stop the back wave cancelling the front.

7. A phone speaker cannot make bass because it cannot displace enough air, by about three orders of magnitude. That is physics, not software.
8. Condenser, electret and MEMS microphones modulate a bias voltage rather than generating the signal energy from the sound itself.
9. Sampling captures a wave perfectly if the rate exceeds twice the highest frequency present, which is the Nyquist-Shannon theorem.
10. Anti-aliasing filtering before the converter is mandatory, because folded frequencies can never be separated afterwards.
11. Bit depth sets dynamic range at about 6 dB per bit, so 16 bits gives roughly 96 to 98 dB, and dither turns ugly quantization distortion into benign hiss.
12. Almost every audio converter today is oversampling sigma-delta, in both directions.
13. CD audio is 44,100 samples a second, 16 bits, two channels, which is 1.4112 Mbit/s and about 31.75 MB for a three-minute song.
14. Lossless codecs such as FLAC predict and store the error, reaching 50 to 70 percent, while lossy codecs such as MP3, AAC and Opus discard what masking makes inaudible and reach a tenth or less.
15. A codec is a coding method and a container is the file that holds it, which is why MP4 can hold either AAC or ALAC and why extensions mislead.
16. Records store the waveform as a physical groove, tape stores it as aligned magnetic particles, and the CD stores numbers as pit-to-land transitions read by a 780 nm laser.
17. Computer audio is a race against a DMA deadline: buffer size sets latency, and missing the deadline puts a step in the waveform heard as a click.
18. The old telephone reserved a whole path per call and digitized voice as 8 kHz, 8-bit companded samples giving 64 kbit/s, multiplexed into 1.544 Mbit/s T1 or 2.048 Mbit/s E1 and controlled by SS7.
19. Mobile voice sends vocal tract parameters instead of a waveform at around 13 kbit/s, which is why it sounds thin, while internet calls send RTP over UDP with jitter buffers, loss concealment and echo cancellation.
20. Voice tolerates loss but hates delay, because a 20 ms hole can be concealed and a 400 ms delay destroys turn-taking, while file transfer is the opposite, which is why one uses UDP and the other TCP.



PART C

The Machine

The processor, the memory ladder, storage, the buses and drivers that join it all up, and how we got here.

Inside the CPU

10.0 What this chapter gives you

1. You will be able to name every part inside a processor and say what it does.
2. You will be able to walk one instruction from memory to result, step by step, and say which wire carries what at each step.
3. You will be able to read a line of x86-64 assembly and the same line in ARM64 assembly, and say what the machine does with it.
4. You will be able to explain what an instruction set architecture is, and why it is a contract rather than a design.
5. You will be able to explain pipelining with real timing tables, and say exactly why a very deep pipeline made the Pentium 4 slower, not faster.
6. You will be able to explain out-of-order execution, register renaming and branch prediction without any magic, with real accuracy figures.
7. You will be able to explain why processors stopped getting faster in megahertz around 2005 and started getting wider and more numerous instead.
8. You will be able to walk the MESI cache coherence protocol between two cores by hand, and calculate Amdahl's law for a real program.
9. You will be able to read a desktop processor spec sheet and a phone system-on-chip spec sheet and decode every single number on the page.
10. You will be able to say why gigahertz alone tells you almost nothing, and what number to look at instead.

10.1 What a CPU is: a box that repeats one loop forever

■ 10.1.1 PLAIN - in simple words

1. A CPU is a small square of silicon that does one thing over and over.
2. The one thing is: get the next order, work out what it means, do it.
3. Then it does it again. And again. Billions of times each second.
4. Nothing else. There is no cleverness hidden anywhere. It is a loop.
5. The orders are called **instructions**. Each is just a number in memory.
6. An instruction is tiny. "Add these two numbers." "Copy this to there." "If that number is zero, jump somewhere else."
7. A program is a long list of these tiny orders, one after another.
8. Everything you have ever seen a computer do is millions of these orders running fast enough that the steps blur into one smooth thing.

9. Earlier chapters built the pieces: switches, gates, an adder, an arithmetic unit, and cells that remember a bit. This chapter wires them into a CPU.

■ 10.1.2 PLAIN – a picture in your head

1. Picture a very fast, very obedient cook alone in a kitchen.
2. On the wall is a long shelf of recipe cards, numbered 1, 2, 3, and so on.
3. The cook has a small clip that holds one card at a time. That clip is the **instruction register**.
4. The cook has a counter on a string round the neck showing which card is next. That is the **program counter**.
5. The cook has a few small bowls on the bench, always within reach. Those are the **registers**.
6. There is a big cold store at the far end of the building. That is main memory. Walking there takes a long time.
7. Near the bench there is a small fridge holding whatever was fetched recently, so the cook does not have to walk. That is the **cache**.
8. The cook does exactly what the card says, moves the counter to the next number, and picks up the next card. Forever.

Where this comparison breaks: the cook understands the recipe. The CPU does not understand anything. The bit pattern of the instruction physically opens and closes switches, and the result falls out. There is no reading and no deciding, only voltage arriving at gates.

■ 10.1.3 PLAIN – a worked example

1. Say memory holds the number 7 at address 64 and the number 5 at address 65.
2. We want their sum written to address 66.
3. The CPU does this in four orders, and each order is one trip round the loop.

Step	The order	What moves
1	Load address 64 into bowl 0	7 arrives in bowl 0
2	Load address 65 into bowl 1	5 arrives in bowl 1
3	Add bowl 1 into bowl 0	bowl 0 becomes 12
4	Store bowl 0 to address 66	12 arrives in memory

4. Notice that steps 1, 2 and 4 move data around and do no maths at all.
5. Only step 3 does arithmetic. That is normal. Most instructions in a real program are moving things, not calculating things.
6. Notice also that the CPU never added “7 plus 5”. It added whatever happened to be in bowl 0 and bowl 1. It has no idea what those numbers mean.

■ 10.1.4 PLAIN – what is really happening inside

1. Split the CPU into two halves: the part that carries numbers, and the part that decides what happens to them.
2. The carrying part is the **datapath**: registers, the arithmetic unit, and the wires between them.
3. The deciding part is the **control unit**. It reads the bit pattern of the instruction and switches on exactly the right set of wires.
4. Think of the control unit as a room full of light switches. An instruction pattern is a hand that flips a particular set of switches on.
5. Those control wires say things like: “let register 1 out onto bus A”, “tell the arithmetic unit to add, not subtract”, “write the answer back into register 0”, “increase the counter by one”.

6. Everything else is plumbing. Buses are shared bundles of wires that carry an address, or data, or control signals.
7. A **memory controller** sits at the edge of the chip and talks the exact electrical language that memory chips expect.
8. All of it moves in step with a clock, a square wave that ticks billions of times a second and says “now everyone move”.

■ 10.1.5 TECHNICAL – the engineer’s version

1. The functional blocks of any general-purpose CPU core are as follows.
2. **Control unit**: decodes the instruction and drives the control signals. In a modern x86 core it is itself a small program in read-only memory, plus hardwired fast paths. See section 10.6 on microcode.
3. **ALU (arithmetic logic unit)**: add, subtract, AND, OR, XOR, shift, compare. Built from the adders and multiplexers of Chapter 5.
4. **Register file**: a small multi-ported SRAM array. Multi-ported means several reads and writes can happen in the same clock cycle.
5. **Program counter (PC)**, called RIP on x86-64 and PC on ARM64: holds the address of the next instruction.
6. **Instruction register (IR)**: holds the instruction word currently being decoded. On out-of-order cores this is not one register but a queue.
7. **Caches**: L1 instruction, L1 data, L2 per core, L3 shared. Section 10.11.
8. **Buses and interconnect**: on-die, these are now networks, not buses. AMD uses Infinity Fabric, Arm uses CMN mesh interconnects.
9. **Memory controller**: generates DDR5 command sequences, refresh, training. On AMD desktop parts it lives on a separate I/O die, not on the core die.

Chip	Year	Transistors	Clock
Intel 4004	1971	2,300	740 kHz
Intel 8086	1978	29,000	5 MHz
Intel 80386	1985	275,000	12 MHz
Intel Pentium	1993	3.1 million	60 MHz
Core 2 Duo	2006	291 million	2.93 GHz

10. The Intel 4004 shipped on 15 November 1971. Federico Faggin led the design with Ted Hoff, Stan Mazor and Masatoshi Shima of Busicom.
11. An AMD Zen 4 core complex die holds 6.57 billion transistors in 70 mm² on TSMC N5. The Zen 5 equivalent is roughly 8.3 billion in about 70.6 mm² on TSMC N4P. Treat the Zen 5 figure as approximate.
12. To observe the blocks on Linux: `lscpu`, `lscpu -C` for caches, `cat /proc/cpuinfo`. On macOS: `sysctl -a | grep machdep.cpu`.

■ 10.1.6 WORDS – remember these

1. Instruction — one tiny order — an encoded operation in the ISA.
2. Datapath — the pipes numbers flow through — registers, ALU, buses.
3. Control unit — the switchboard — logic that drives control signals from the decoded opcode.
4. Program counter — a bookmark — register holding the next instruction address, RIP on x86-64, PC on ARM64.

5. Instruction register — the clip holding the current card — latch holding the instruction word being decoded.
6. Register file — the small bowls on the bench — multi-ported SRAM array of architectural or physical registers.
7. Memory controller — the translator at the door — on-die block generating DDR command and timing sequences.

10.2 The stored-program idea

■ 10.2.1 PLAIN - in simple words

1. Here is the single biggest idea in computing, and it is very simple.
2. Instructions are numbers. Data is numbers. Memory holds numbers.
3. So instructions and data can live in the same memory, side by side.
4. Nothing in memory has a label saying “this one is an order”. It is only an order because the program counter pointed at it.
5. That is why one machine can run any program. You do not rewire it. You just put different numbers in memory.
6. Before this idea, changing a computer’s job meant physically replugging it.
7. It also means a program can write numbers that are then run as orders. That is enormously useful and enormously dangerous, and both are covered below.

■ 10.2.2 PLAIN - a picture in your head

1. Picture a huge wall of numbered pigeonholes, all identical.
2. Some hold shopping lists. Some hold instructions for the shop assistant.
3. Nothing on the outside of a pigeonhole tells you which is which.
4. The assistant simply starts at pigeonhole 100 and treats whatever is there as an instruction, because that is where they were told to start.
5. If someone slips a shopping list into pigeonhole 100, the assistant will try to obey the shopping list as if it were an order.
6. That mistake is the root of a whole class of security holes.

Where this comparison breaks: real hardware now does add labels. Modern memory pages carry permission bits saying “may be executed” or “may be written”, and in most cases not both. The pigeonholes are still identical, but the shelf now has rules about which ones the assistant may read orders from.

■ 10.2.3 PLAIN - a worked example

1. Take the four bytes 48 83 C0 01 sitting in memory.
2. Read as data, that is just four numbers: 72, 131, 192, 1.
3. Read as an x86-64 instruction, it is `add rax, 1`, meaning “add one to register RAX”.
4. Both readings are correct. The bytes do not choose. The CPU chooses, by where the program counter points.
5. Now take the two bytes EB FE. As an x86-64 instruction that is `jmp -2`, a jump back to itself: an infinite loop in two bytes.
6. If a program accidentally jumps into a block of text, the CPU will happily try to execute the letters. H is 0x48, which starts many valid x86 instructions. That is why crashes from bad jumps look so random.

■ 10.2.4 PLAIN – what is really happening inside

1. There are two ways to arrange memory, and they have names.
2. **Von Neumann**: one memory, holding both instructions and data, reached through one set of wires.
3. **Harvard**: two separate memories with separate wires, one for instructions and one for data.
4. Von Neumann is flexible. You can load a program as if it were data, then run it. That is exactly what an operating system does when it starts an app.
5. Von Neumann has a bottleneck. One set of wires must carry both the next instruction and the data it needs, so they take turns.
6. Harvard has no such traffic jam, because the two roads are separate.
7. Real CPUs use a mixture called **modified Harvard**. Deep down there is one memory, von Neumann style. But right next to the core there are two separate small caches: one for instructions, one for data.
8. So the core sees Harvard, with two roads, and the system sees von Neumann, with one memory. You get the speed of one and the flexibility of the other.

■ 10.2.5 TECHNICAL – the engineer’s version

1. John von Neumann’s “First Draft of a Report on the EDVAC” circulated on 30 June 1945 and described a single addressable store for code and data. The design work was joint with J. Presper Eckert and John Mauchly, whose names were left off the draft, a dispute that is still argued about.
2. The Harvard Mark I, completed in 1944 under Howard Aiken at Harvard, read instructions from punched paper tape and data from separate counters. That physical separation is where the name comes from.
3. Modified Harvard in practice: separate L1 instruction and L1 data caches fed from a unified L2. On AMD Zen 5 that is 32 KB L1i and 48 KB L1d per core, backed by a unified 1 MB L2.
4. This split means self-modifying code needs explicit help. Writing bytes goes into L1d. The fetcher reads L1i. The two can disagree.
 1. On x86-64 the hardware keeps them coherent for you, but a serializing instruction is still required before the new bytes are guaranteed run.
 2. On ARM64 you must do it by hand: DC CVAU to clean the data cache to the point of unification, IC IVAU to invalidate instruction cache, then DSB and ISB barriers. Skipping this is a classic JIT bug.
5. The danger side. A stack buffer overflow writes past the end of an array and overwrites the saved return address, so the CPU returns into attacker bytes. The Morris worm of 2 November 1988 used exactly this against the Unix fingerd service.
6. The defence is the no-execute page permission bit. AMD shipped NX with Athlon 64 in 2003, Intel called it XD, and Microsoft enabled it as Data Execution Prevention in Windows XP Service Pack 2 in August 2004.
7. Attackers answered with return-oriented programming: chain together fragments of code that already exists and is already marked executable. No new code is written, so NX does not stop it.
8. The useful side. A just-in-time compiler writes machine code into a data buffer at run time, flips the page permission from writable to executable, and jumps into it. The JavaScript engines in every browser do this thousands of times while a page loads.

Term	Code and data memory	Seen in
Von Neumann	One shared	Program model of a PC

Term	Code and data memory	Seen in
Harvard	Two separate	Small microcontrollers
Modified Harvard	Split caches, one store	Every mainstream CPU

9. Tools: `readelf -l` shows segment permissions, R E for read-execute and RW for read-write. On Linux `cat /proc/self/maps` shows the same live.

■ 10.2.6 WORDS – remember these

1. Stored program — orders live in memory like any other number — code and data share one address space.
2. Von Neumann architecture — one memory for everything — unified instruction and data store, with a shared bus bottleneck.
3. Harvard architecture — two separate memories — separate instruction and data address spaces and buses.
4. Modified Harvard — one memory but two front doors — split L1i and L1d over a unified lower hierarchy.
5. Buffer overflow — writing past the end of a box — overwriting adjacent stack or heap state, classically the saved return address.
6. NX bit — a page marked “not runnable” — no-execute permission bit in the page table entry, XD on Intel.
7. JIT — writing new code while running — just-in-time compilation into a writable page later marked executable.

10.3 The fetch-decode-execute cycle

■ 10.3.1 PLAIN – in simple words

1. The loop the CPU repeats forever has three parts and standard names.
2. **Fetch**: read the instruction at the address in the program counter.
3. **Decode**: work out what it is and where its inputs are.
4. **Execute**: do it, and write the answer somewhere.
5. Then increase the program counter and start again.
6. Some instructions add a fourth part, a memory access, and a fifth, writing the result back into a register. We will use the five-part version later.
7. If the instruction was a jump, the program counter is not increased by one. It is loaded with a new address. That is how loops and decisions work.
8. That is the entire behaviour of a processor. Everything after this in the chapter is a trick to run this loop faster.

■ 10.3.2 PLAIN – a picture in your head

1. Picture a postal sorting clerk at a desk.
2. The clerk reads the number on a slip of paper: “go to shelf 100”.
3. They walk to shelf 100 and bring back the envelope. That is fetch.
4. They open it and read what kind of job it is. That is decode.
5. They do the job at their desk. That is execute.
6. They cross out 100 and write 101 on the slip, then start again.
7. If the envelope said “next, go to shelf 240”, they write 240 instead.

Where this comparison breaks: a real CPU does not finish one envelope before starting the next. It has several envelopes open at once, in different stages, and it often guesses which shelf comes next before it knows. Those are pipelining and branch prediction, sections 10.7 and 10.8.

■ 10.3.3 PLAIN - a worked example

1. Let us invent a small machine so nothing is hidden. Call it KB-1.
2. It has four registers R0 to R3, 16-bit words, and 16-bit instructions.
3. An instruction splits into fields like this.

```

bit 15..12  11..10  9..8    7..0
+-----+-----+-----+-----+
| opcode |  rd  |   rs   | imm/addr |
+-----+-----+-----+-----+

```

4. Five opcodes are enough for our program.

Opcode	Name	Meaning
0001	LOAD rd,[a]	rd ← MEM[a]
0010	STORE rd,[a]	MEM[a] ← rd
0011	ADD rd,rs	rd ← rd + rs
0100	SUB rd,rs	rd ← rd - rs
0111	JZ addr	jump if last result 0

5. Memory starts with 7 at address 0x40, 5 at 0x41, and 0 at 0x42.
6. The program, with its exact machine words, is this.

```

addr  word  assembly
0x00  0x1040  LOAD  R0, [0x40]
0x01  0x1441  LOAD  R1, [0x41]
0x02  0x3100  ADD   R0, R1
0x03  0x2042  STORE R0, [0x42]

```

7. Check the encoding of 0x1441 by hand. In binary it is 0001 01 00 01000001. Opcode 0001 is LOAD, rd is 01 meaning R1, the low eight bits are 0x41. So: load from address 0x41 into R1.
8. Now run it and watch every value change.

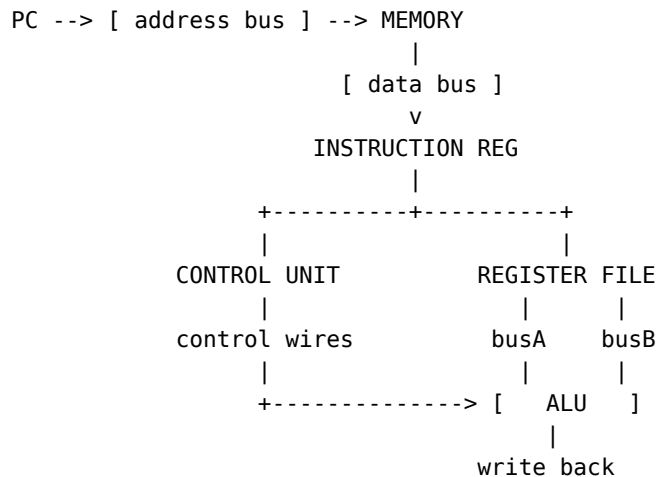
After	PC	R0, R1	Mem[0x42]
start	0x00	0, 0	0
instr 1	0x01	7, 0	0
instr 2	0x02	7, 5	0
instr 3	0x03	12, 5	0
instr 4	0x04	12, 5	12

9. Three kinds of instruction did all the work: a load, an arithmetic operation, and a store. Almost all real code is these three kinds.

■ 10.3.4 PLAIN - what is really happening inside

1. Now walk one cycle at the level of the wires. Take instruction 3, ADD.
2. Tick 1, fetch. The program counter holds 0x02. Its value is driven onto the address bus. The read line is raised.

3. Memory answers with the 16 bits 0x3100 on the data bus.
4. The write-enable line of the instruction register is raised, so on the next clock edge the instruction register latches 0x3100.
5. Tick 2, decode. The top four bits, 0011, go into the control unit. The control unit is a lookup that turns 0011 into a set of raised wires.
6. Bits 11 to 10 pick R0 as the destination and as source A. Bits 9 to 8 pick R1 as source B. Those go to the register file's address inputs.
7. The register file drives 7 onto bus A and 5 onto bus B. No clock needed: reading is combinational, the values just appear after a small delay.
8. Tick 3, execute. The control unit puts the code for "add" on the ALU's function input. The adder settles and 12 appears at its output.
9. The write-back line for R0 is raised. On the next clock edge, R0 latches 12.
10. In parallel, the program counter's own adder computes 0x02 plus 1, and the program counter latches 0x03 on the same edge.
11. Every one of these steps is switches opening and closing, as in Chapter 4. Nothing is interpreted. The bit pattern is the command.



■ 10.3.5 TECHNICAL – the engineer's version

1. The same program in real x86-64, with RDI holding the base address.

```

8B 07      mov  eax, dword ptr [rdi]
03 47 04    add  eax, dword ptr [rdi+4]
89 47 08    mov  dword ptr [rdi+8], eax

```

2. Note the byte counts: two, three, three. x86-64 instructions are variable length, from 1 to 15 bytes, with 15 a hard architectural limit.
3. Note also that x86-64 can add straight from memory. One instruction does a load and an add. That is the classic CISC habit.
4. The same program in ARM64, with X0 holding the base address.

```

B9400001    ldr  w1, [x0]
B9400402    ldr  w2, [x0, #4]
0B020021    add  w1, w1, w2
B9000801    str  w1, [x0, #8]

```

5. Every ARM64 instruction is exactly four bytes. Always. That is the classic RISC habit, and it makes the decoder far simpler.

6. ARM64 cannot add from memory. Loads and stores are the only instructions that touch memory. This is called a load-store architecture.
7. Decode 0xB9400402 by hand. Bits 31 to 22 are 1011100101, the 32-bit LDR unsigned-offset form. Bits 21 to 10 are the scaled immediate, value 1, and the scale for a 32-bit load is 4, so the offset is 4 bytes. Bits 9 to 5 are Rn equals X0. Bits 4 to 0 are Rt equals W2.
8. On a modern core, none of this happens in three tidy ticks. The x86-64 `add eax, [rdi+4]` is split into two internal micro-operations, an address generation plus load, and an add. Section 10.6.
9. To watch real instructions: `objdump -d ./a.out` disassembles a binary, `gdb` with `layout asm` and `stepi` single-steps them, and `perf stat ./a.out` counts how many actually retired.

■ 10.3.6 WORDS – remember these

1. Fetch — go and get the next order — read memory at the PC into the instruction register.
2. Decode — work out what it says — map opcode and operand fields onto control signals and register file ports.
3. Execute — do it — drive the ALU or address unit and produce a result.
4. Write-back — put the answer away — latch the result into the destination register on the clock edge.
5. Load-store architecture — only two instructions touch memory — arithmetic operates on registers only, as in ARM64, RISC-V and MIPS.
6. Combinational — settles by itself, no clock — logic whose output depends only on current inputs after a propagation delay.

10.4 Instruction set architecture

■ 10.4.1 PLAIN – in simple words

1. An **instruction set architecture**, or ISA, is the list of orders a CPU understands, and exactly how each one is written as a number.
2. It also says how many registers there are, what they are called, how big a number is, and what happens when something goes wrong.
3. It is a promise, not a design. It says what the chip must appear to do. It says nothing about how the chip does it inside.
4. That promise is why a program compiled in 2003 still runs today. The inside of the chip changed completely. The promise did not.
5. Two chips can keep the same promise with wildly different insides. An Intel and an AMD desktop chip share almost nothing internally, yet run the same programs, because they honour the same ISA.
6. The three ISAs that matter today are x86-64, ARM64 and RISC-V.

■ 10.4.2 PLAIN – a picture in your head

1. Think of an ISA as the menu in a restaurant, not the kitchen.
2. The menu says: item 14 is soup, item 22 is rice, item 30 is bread.
3. Any kitchen that produces the right dish for each number is a valid kitchen. One may use a wood fire, another an induction hob.
4. A customer who learned the menu twenty years ago can still order item 14 and get soup, even though the kitchen was rebuilt three times.
5. If a new kitchen decided item 14 now means salad, every old customer breaks. That is why ISAs almost never remove things.
6. That is also why x86-64 still carries instructions from 1978.

Where this comparison breaks: menus are short, and ISAs are not. A full x86-64 manual set from Intel runs to several thousand pages across its volumes, and nobody knows all of it. Also, some menu items are far slower to cook than others, and the menu does not tell you which.

■ 10.4.3 PLAIN – a worked example

1. Every instruction is a number split into fields. Take RISC-V, which is the tidiest example, with a fixed 32-bit length.
2. The R-type format, used for register-to-register arithmetic, is this.

```

bits 31..25 24..20 19..15 14..12 11..7 6..0
+-----+-----+-----+-----+-----+
| funct7 | rs2  | rs1  | funct3|  rd  | opcode |
+-----+-----+-----+-----+-----+

```

3. For `add x5, x6, x7`: opcode 0110011, funct3 000, funct7 0000000, rs1 is x6, rs2 is x7, rd is x5. The whole word comes out as 0x007302B3.
4. Change funct7 to 0100000 and the same opcode becomes `sub`. One bit flip in a field turns add into subtract. That is what a field is for.
5. An **immediate** is a constant baked into the instruction itself, rather than read from a register. `addi x5, x6, 100` carries the 100 in 12 bits.
6. Now addressing modes: the different ways an instruction names where its data is. Here they are with worked values, using x86-64 notation.

Mode	Example	Where the data is
Immediate	<code>mov rax, 42</code>	In the instruction
Register	<code>mov rax, rbx</code>	In RBX
Absolute	<code>mov rax, [0x600100]</code>	At that fixed address
Reg indirect	<code>mov rax, [rbx]</code>	At the address in RBX
Base+disp	<code>mov rax, [rbx+8]</code>	At RBX plus 8
Indexed	<code>mov rax, [rbx+rcx*4]</code>	At RBX plus 4 times RCX
PC-relative	<code>lea rax, [rip+0x20]</code>	0x20 past the next instr

7. Work one through. If RBX holds 0x1000 and RCX holds 3, then `mov rax, [rbx+rcx*4]` reads the 8 bytes at 0x1000 plus 12, which is 0x100C. That single instruction indexes an array of 4-byte elements.
8. That scale factor of 1, 2, 4 or 8 exists precisely because arrays of bytes, shorts, ints and longs are so common.

■ 10.4.4 PLAIN – what is really happening inside

1. The decoder is a big lookup. It takes the instruction bits and produces control signals, plus register numbers, plus a constant.
2. With fixed-length instructions, the decoder knows where every field is before it has read anything. It can slice all fields in parallel.
3. With variable-length instructions it cannot. It must work out how long the first instruction is before it knows where the second begins.
4. On x86-64 the length depends on prefixes, on the opcode, on the ModRM byte, on the SIB byte and on the displacement size. Length decoding is a serial chain, and it is genuinely hard.
5. Intel and AMD solve it by brute force: try to decode at many byte offsets at once and throw away the wrong answers, plus cache the decoded results so the work is not repeated. Section 10.6.

6. That is a real, permanent cost of x86-64. It costs transistors and power that an ARM64 decoder simply does not spend.
7. It is also, in 2026, a survivable cost. It is a fixed overhead at the front of a core that is otherwise dominated by caches and execution units.

■ 10.4.5 TECHNICAL – the engineer’s version

1. The RISC idea began at IBM with John Cocke’s 801 project, which ran from 1975 onward and was not published widely at the time.
2. David Patterson’s group at Berkeley built RISC-I in 1981 and coined the term. John Hennessy’s group at Stanford built MIPS from 1981.
3. Hennessy and Patterson shared the 2017 ACM Turing Award for this work.
4. The original argument: compilers do not use complex instructions, so spend the transistors on registers, pipelines and caches instead.
5. What is the real difference today. Honest answer: at the level of the execution core, almost none. Both x86-64 and ARM64 cores decode into internal micro-operations, rename registers, execute out of order and retire in order. The differences that survive are these.
 1. Decode complexity. Fixed 32-bit versus 1 to 15 bytes.
 2. Memory operands in arithmetic instructions, allowed on x86-64, not on ARM64.
 3. Memory ordering model, strong on x86-64, weak on ARM64. Section 10.10.
 4. Legacy baggage: x86-64 must still support 16-bit real mode at power-on.
6. Where experts disagree: some argue variable-length decode caps how wide an x86 front end can practically get, others point to Intel and AMD shipping 8-wide decode in 2024 and say the ceiling keeps moving. Both sides have a point and the argument is not settled.

Feature	x86-64	ARM64	RISC-V
Year	2000 spec	2011	2010 start
Designer	AMD	Arm Ltd	UC Berkeley
Licence	Cross-licence	Paid licence	Open, BSD terms
Registers	16 integer	31 integer	32 integer
Instr length	1 to 15 bytes	4 bytes fixed	4 or 2 bytes
Typical use	PC, server	Phone, Mac	Embedded, chips

7. Two footnotes to that table. RISC-V’s embedded variant has 16 registers rather than 32, and its 2-byte instruction length comes from the optional C compressed extension, not the base set.
8. Dates to keep straight. AMD published the x86-64 specification in 2000 and shipped Opteron in April 2003. Intel shipped its compatible EM64T in the Nocona Xeon in June 2004. Arm announced ARMv8-A with AArch64 in October 2011. RISC-V began at Berkeley in 2010 under Krste Asanovic and David Patterson, its unprivileged ISA was ratified as version 20191213, and RISC-V International moved to Switzerland, with the rename completed in March 2020.
9. Tools: `gcc -march=native -Q --help=target` lists the ISA extensions the compiler will use, and on Linux the `flags` line in `/proc/cpuinfo` lists what the chip actually supports.

■ 10.4.6 WORDS – remember these

1. ISA — the list of orders and how they are written — architectural contract defining instructions, registers, memory model and exceptions.
2. Opcode — the part that says what to do — the operation field of an instruction word.
3. Operand — the part that says what to do it to — register number, memory address or constant.

4. Immediate — a number written into the order itself — constant encoded in the instruction, not fetched.
5. Addressing mode — the recipe for finding the data — the rule turning instruction fields into an effective address.
6. RISC — few simple orders, all the same size — reduced instruction set computer, load-store, fixed length.
7. CISC — many orders, many sizes — complex instruction set computer with memory operands and variable length.
8. Microarchitecture — how one chip actually implements the promise — the internal design, invisible to software.

10.5 Registers in real CPUs

■ 10.5.1 PLAIN - in simple words

1. A **register** is a tiny box inside the CPU that holds one number.
2. There are very few of them. A modern CPU has a few dozen that programs can name, not thousands.
3. They are the only storage the arithmetic unit can reach in the same tick of the clock. Everything else is further away.
4. Some registers are general: you can put anything in them.
5. Some have a fixed job. One always points at the next instruction. One always points at the top of the stack. One holds yes or no answers about the last calculation.
6. Almost all real work happens in registers. Data is pulled in from memory, chewed on in registers, and pushed back out.
7. A good compiler spends most of its effort deciding which values live in registers and which have to spill out to memory.

■ 10.5.2 PLAIN - a picture in your head

1. Think of a carpenter at a bench.
2. The registers are the two or three tools actually in their hands.
3. The L1 cache is the tool tray at the edge of the bench, an arm's reach away.
4. Main memory is the tool cupboard on the other side of the workshop.
5. Storage is the hardware shop down the road.
6. Every step out costs time. Hands are instant. The tray takes a second. The cupboard takes a minute. The shop takes a day.
7. So the carpenter arranges work to keep what is needed in their hands. That is exactly what a compiler's register allocator does.

Where this comparison breaks: hands hold whole tools, and registers hold only fixed-size numbers, typically 64 bits. Also a carpenter can put a tool down anywhere, while a register value must be explicitly written somewhere or it is simply overwritten and lost.

■ 10.5.3 PLAIN - a worked example

1. On x86-64 the general purpose registers have odd names, because they grew from 1978 by accident rather than by plan.
2. The 8086 of 1978 had eight 16-bit registers, and each name meant something.

Name	Stood for	Old special job
AX	Accumulator	Results of maths
BX	Base	A base address
CX	Counter	Loop counts
DX	Data	Second half of results
SI	Source index	Source of a copy
DI	Destination index	Target of a copy
BP	Base pointer	Bottom of a stack frame
SP	Stack pointer	Top of the stack

3. In 1985 the 80386 widened them to 32 bits and put an E in front: EAX, EBX, and so on. E stands for extended.
4. In 2003 x86-64 widened them to 64 bits and put an R in front: RAX, RBX, and so on. It also added eight plain-numbered ones, R8 to R15.
5. So RAX, EAX, AX, AH and AL are all the same physical register, seen through windows of 64, 32, 16, 8 and 8 bits.
6. ARM64 refused all of that. It has 31 registers with no story attached: X0 to X30 for the 64-bit view, W0 to W30 for the low 32 bits.
7. There is no X31. That slot means either the constant zero or the stack pointer, depending on the instruction. Reading the zero register always gives 0, and writing to it throws the value away.

■ 10.5.4 PLAIN – what is really happening inside

1. Why are registers so fast. Four reasons, all physical.
2. They are physically next to the arithmetic unit, often microns away. Signal travel time is close to nothing.
3. They are addressed by a tiny number, three to five bits. That decode is a few gates, not a lookup in a table of tags like a cache.
4. They are built from fast, wide SRAM cells with many ports, so several registers can be read and written in the same cycle.
5. There are very few of them, so the array is small, so its wires are short, so it settles quickly.
6. The cost of all that is area and power per bit. A register file bit costs far more silicon than a cache bit, which is why you cannot have thousands.
7. There is one more thing hiding here. The names in the program are not the real boxes. A modern CPU has hundreds of real registers and shuffles which physical box currently holds RAX. That is register renaming, section 10.8.

■ 10.5.5 TECHNICAL – the engineer's version

1. x86-64 architectural integer state: 16 general purpose registers of 64 bits, RIP as the instruction pointer, RFLAGS as the status register.
2. Writing a 32-bit sub-register zero-extends into the full 64 bits. Writing an 8-bit or 16-bit sub-register does not, and leaves the top bits alone. That asymmetry is a standard, written in the architecture manual, and it causes real partial-register stalls if you get it wrong.
3. RFLAGS bits worth knowing: CF is bit 0 carry, PF bit 2 parity, ZF bit 6 zero, SF bit 7 sign, IF bit 9 interrupt enable, DF bit 10 direction, OF bit 11 overflow.
4. ARM64 architectural integer state: X0 to X30, plus SP, plus PC which is not a general register and cannot be written directly. Condition flags live in PSTATE as N, Z, C, V.
5. By convention, not by hardware rule, ARM64 uses X30 as the link register holding the return address, and X29 as the frame pointer.

6. Calling conventions are conventions, set by an ABI document, not by the chip. The System V ABI used on Linux and macOS passes integer arguments in RDI, RSI, RDX, RCX, R8, R9 and returns in RAX. Microsoft's x64 ABI uses RCX, RDX, R8, R9. ARM64's AAPCS64 passes in X0 to X7 and returns in X0.
7. Physically, the architectural names map onto a much larger physical register file. AMD Zen 5 has a floating point physical register file of 384 entries, doubled from 192 in Zen 4, and an integer file of a couple of hundred entries.

Storage	Typical latency	Typical size
Register	Under 1 cycle	16 x 64 bits
L1 data cache	4 to 5 cycles	48 KB
L2 cache	14 cycles	1 MB
L3 cache	40 to 50 cycles	32 MB per die
DDR5 main memory	70 to 90 ns	32 GB

8. Those latency figures are for AMD Zen 5 on the Ryzen 9000 series, 2024. At 5.7 GHz one cycle is 0.175 nanoseconds, so an L1 hit is about 0.7 ns and a memory access is over 400 times slower.
9. Tools: `gdb command info registers` prints them live. On Linux, `perf stat -e ld_blocks.no_sr` and similar counters expose partial register and store-forwarding stalls on Intel parts.

■ 10.5.6 WORDS – remember these

1. Register — a tiny box for one number inside the CPU — architecturally named fast storage, typically 64 bits wide.
2. General purpose register — one you can use for anything — an integer register with no fixed hardware role.
3. Stack pointer — marks the top of the working pile — RSP on x86-64, SP on ARM64.
4. Flags register — a strip of yes or no answers — RFLAGS on x86-64, the NZCV bits of PSTATE on ARM64.
5. Zero register — a tap that always gives zero — XZR and WZR on ARM64, no equivalent on x86-64.
6. Spill — running out of hands and putting something down — the compiler storing a value to the stack because registers ran out.
7. ABI — the agreement about who passes what where — application binary interface, a convention, not part of the ISA.

10.6 Microcode

■ 10.6.1 PLAIN – in simple words

1. Some instructions are simple, and the hardware can do them directly.
2. Some are not. “Copy this whole block of memory to there” is one order for the programmer but hundreds of steps for the hardware.
3. So inside the CPU there is a second, smaller, more private instruction set.
4. A complicated outside instruction is turned into a short program of these small private steps. Those steps are called **micro-operations**.
5. The little program that produces them is the **microcode**.
6. This is why one x86 instruction can take one clock cycle or one hundred. Some are a single step, and some are a whole routine.
7. The important consequence is this: some of the behaviour of a CPU is stored, not wired. Stored things can be changed later.

- That is how a firmware update can change how a chip behaves without anyone touching the silicon.

■ 10.6.2 PLAIN – a picture in your head

- Picture a restaurant kitchen again. The menu is the instruction set.
- Some menu items are one action: “pour a glass of water”.
- Some are not: “make a three-course meal for six”. The chef has a card file of standard sub-recipes and works through them in order.
- The card file is the microcode. It sits in the kitchen, not on the menu.
- Customers never see the cards. They only see the menu, which never changes.
- If a sub-recipe turns out to be wrong, the owner can replace those cards overnight. The menu still says the same words the next morning.

Where this comparison breaks: the chef can improvise, and the microcode sequencer cannot. Also, most modern instructions skip the card file entirely and are handled by hardwired fast decoders, because going through microcode is slow. Microcode is the exception path, not the normal path.

■ 10.6.3 PLAIN – a worked example

- Take the x86-64 instruction `add [rax], rbx`, which means “add RBX to the number in memory at the address in RAX”.
- To a programmer that is one instruction, four bytes.
- Inside, it becomes roughly three micro-operations.

1. load	<code>t0 <- MEM[rax]</code>	(read 8 bytes)
2. add	<code>t0 <- t0 + rbx</code>	(do the arithmetic)
3. store	<code>MEM[rax] <- t0</code>	(write 8 bytes back)

- `t0` is a temporary that has no name in the instruction set. Programs cannot see it. It exists only for the duration of the instruction.
- Now take `rep movsb`, which copies a whole block of bytes. That is a microcoded loop. It may issue thousands of micro-operations from one instruction, and the exact number depends on the block length.
- Compare with ARM64. The same job needs three separate instructions written out by the compiler, one load, one add, one store. The work is identical. The difference is who wrote the three steps down: the microcode, or the compiler.

■ 10.6.4 PLAIN – what is really happening inside

- The front end of an x86 core has two paths.
- The fast path: several simple hardware decoders, each turning one instruction into one to four micro-operations directly.
- The slow path: the microcode sequencer, a small read-only memory holding canned sequences, used for anything complicated.
- Decoding x86 is expensive, so cores keep a **micro-op cache**. Once an instruction has been decoded, the resulting micro-operations are stored.
- The next time round a loop, the front end reads finished micro-operations straight out of that cache and skips decoding entirely.
- That saves a lot of power, and it also delivers more operations per cycle than the decoders can.
- A microcode update is a signed blob of data. Firmware loads it into a small patch memory very early in boot. It is not stored on the chip permanently.

8. It disappears at every power-off and must be loaded again next boot, by the motherboard firmware or by the operating system.

■ 10.6.5 TECHNICAL – the engineer’s version

1. Maurice Wilkes proposed microprogramming in 1951, at the Manchester University Computer Inaugural Conference. EDSAC 2, running from 1958, was the first microprogrammed machine.
2. IBM made it famous with System/360 in 1964: one ISA implemented by many physically different machines, each with its own microcode. That is the clearest early proof that an ISA is a contract, not a design.
3. On Intel cores, simple decoders handle one-to-one and one-to-four cases. Anything longer goes to the MSROM, the microcode sequencer ROM.
4. Micro-op cache sizes, all implementation details rather than architecture.

Core	Year	Micro-op cache
Sandy Bridge	2011	1,536 micro-ops
Skylake	2015	1,536 micro-ops
Golden Cove	2021	About 4,000 micro-ops
AMD Zen 4	2022	About 6,750 ops
AMD Zen 5	2024	12 ops per cycle out

5. Micro-op fusion and macro-op fusion muddy instruction counting. A compare followed by a branch is commonly fused into one internal operation, so counting retired instructions is not the same as counting work.
6. Microcode update mechanics. On Intel, software writes the physical address of the update to model-specific register 0x79 and the load happens. On AMD the equivalent is MSR 0xC0010020. Updates are encrypted and signed. A CPU will reject an unsigned patch.
7. Limits, and this matters. Microcode cannot add execution units, cannot change cache sizes, and cannot rewrite the hardwired fast decoders. It patches sequences and it can flip configuration bits. The number of patch slots is finite.
8. Spectre and Meltdown were disclosed on 3 January 2018. CVE-2017-5753 is Spectre variant 1, bounds check bypass. CVE-2017-5715 is Spectre variant 2, branch target injection. CVE-2017-5754 is Meltdown.
9. Intel’s microcode updates added entirely new architectural controls: IA32_SPEC_CTRL exposing IBRS and STIBP, and IA32_PRED_CMD exposing IBPB. A firmware update added new registers to a shipped ISA. That is unusual.
 1. IBRS restricts indirect branch prediction across privilege levels.
 2. IBPB flushes predictor state at a chosen barrier point.
 3. STIBP stops one hardware thread steering the other thread’s predictor.
10. Google published retpoline on 4 January 2018, a purely software fix for variant 2 that replaces indirect jumps with a return trampoline the predictor cannot usefully steer.
11. Meltdown was not fixed by microcode at all. It was fixed in software by kernel page-table isolation, which unmaps kernel memory while user code runs, based on the 2017 KAISER work from TU Graz.
12. The rollout went badly. Intel’s January 2018 microcode caused unexpected reboots, and Microsoft shipped an update on 29 January 2018 that disabled it. Measured performance drops of 2 to 14 percent were reported on eighth-generation Core platforms, depending heavily on workload.

13. Observe it: `grep microcode /proc/cpuinfo` on Linux gives the loaded revision, `dmesg | grep -i microcode` shows the load at boot, and `iucode_tool -l` lists available Intel updates.

The honest version: saying “the CPU runs microcode” suggests every instruction goes through an interpreter. It does not. On a modern x86 core the large majority of instructions never touch the microcode sequencer, and on ARM64 cores the sequencer is far smaller still. Microcode is the escape hatch for complicated and rare cases, plus a patching mechanism.

■ 10.6.6 WORDS – remember these

1. Micro-operation — one small private step — the internal RISC-like operation an instruction is broken into, often written uop.
2. Microcode — the card file of stored sub-recipes — ROM sequences that expand complex instructions into micro-operations.
3. Micro-op cache — remembering the decode so you need not redo it — a cache of decoded micro-operations, also called the op cache or DSB.
4. Microcode update — an overnight replacement of some cards — a signed, volatile patch loaded by firmware at every boot.
5. IBRS, IBPB, STIBP — three new switches added by a patch — speculation controls introduced by 2018 microcode for Spectre variant 2.
6. KPTI — hiding the kernel’s map from user code — kernel page-table isolation, the software fix for Meltdown.

10.7 Pipelining

■ 10.7.1 PLAIN – in simple words

1. A simple CPU wastes almost all of itself, almost all of the time.
2. While it is fetching an instruction, the arithmetic unit sits idle. While it is doing arithmetic, the fetch hardware sits idle.
3. **Pipelining** fixes that. Break the work into stages, and keep every stage busy on a different instruction at the same time.
4. It does not make any one instruction faster. Each instruction still takes the same number of stages to get through.
5. It makes instructions come out more often. That is throughput, not latency.
6. If the pipeline has five stages and nothing goes wrong, you finish one instruction every clock cycle instead of one every five.
7. Every CPU made since the 1980s is pipelined. It is not optional.

■ 10.7.2 PLAIN – a picture in your head

1. You have four loads of washing. Each load needs washing, drying, folding.
2. Each of the three jobs takes 30 minutes.
3. The naive way: wash, dry and fold load 1 completely, then start load 2. That is 90 minutes each, 360 minutes in total.
4. The pipelined way: as soon as load 1 leaves the washer, load 2 goes in.
5. Now the timeline looks like this, in 30-minute slots.

slot	1	2	3	4	5	6
wash	L1	L2	L3	L4		
dry		L1	L2	L3	L4	
fold			L1	L2	L3	L4

6. Total: 6 slots, 180 minutes, not 360. Twice as fast for four loads.
7. With 100 loads it approaches three times as fast, the number of stages.
8. Notice load 1 still took 90 minutes. Nothing got faster. Things just overlapped.

Where this comparison breaks: laundry loads are independent, and instructions are not. Load 2 might need the result of load 1. Worse, sometimes you do not know which load comes next until load 1 finishes. Those two problems are data hazards and control hazards, and they are the whole difficulty.

■ 10.7.3 PLAIN - a worked example

1. The classic teaching pipeline has five stages, from the Berkeley and Stanford RISC designs of the early 1980s.

Stage	Short name	What it does
1	IF	Fetch instruction
2	ID	Decode, read registers
3	EX	Do arithmetic or address
4	MEM	Access data memory
5	WB	Write result to register

2. Now run five instructions and watch the pipeline fill and drain.

cycle	1	2	3	4	5	6	7	8	9
instr 1	IF	ID	EX	MEM	WB				
instr 2		IF	ID	EX	MEM	WB			
instr 3			IF	ID	EX	MEM	WB		
instr 4				IF	ID	EX	MEM	WB	
instr 5					IF	ID	EX	MEM	WB

3. Five instructions finish in 9 cycles, not 25. The first takes 5 cycles to appear, then one appears every cycle.
4. The formula for n instructions in a k -stage pipeline is k plus n minus 1 cycles, against n times k without pipelining.
5. For n equals 5 and k equals 5: 9 cycles against 25. Speedup 2.78.
6. For n equals 1,000 and k equals 5: 1,004 cycles against 5,000. Speedup 4.98, which is almost the full 5.
7. There is a second, bigger win. Each stage now does one fifth of the work, so each stage settles in one fifth of the time, so the clock can run roughly five times faster too. Pipelining is why clocks got fast at all.

■ 10.7.4 PLAIN - what is really happening inside

1. Between every pair of stages there is a row of flip-flops, from Chapter 6, holding that stage's result until the next clock edge. These are pipeline registers, and they are the physical dividers.
2. Three things can go wrong. They are called hazards.
3. **Structural hazard:** two instructions want the same piece of hardware in the same cycle. For example, one instruction fetching while another reads data, when there is only one memory port. Fixed by splitting instruction and data caches, which is the modified Harvard idea from section 10.2.
4. **Data hazard:** an instruction needs a result that is not written yet.


```

add r1, r2, r3    ; result of r1 written in stage WB
sub r4, r1, r5    ; needs r1 in stage EX, two cycles earlier

```

5. The fix is **forwarding**, also called bypassing. Extra wires take the answer straight from the output of the arithmetic unit back into its own input, before it has been written anywhere.
6. Forwarding cures most data hazards but not all. If the producer is a load from memory, the value does not exist until the MEM stage, so the consumer must wait one cycle. That wasted cycle is a **stall**, and the empty slot pushed through the pipeline is a **bubble**.
7. **Control hazard**: a branch. Until the branch is resolved, the fetch unit does not know which instruction comes next.
8. In a five-stage pipeline you lose two or three cycles per branch if you just wait. Since roughly one instruction in five or six is a branch, that is a large loss, and it is why branch prediction exists.

■ 10.7.5 TECHNICAL – the engineer’s version

1. The IBM 7030, known as Stretch and delivered in 1961, was the first heavily pipelined machine, and it already fetched past unresolved branches.
2. Pipeline depth by generation, all measured in stages of the integer pipeline, which is the usual quoted figure.

Core	Year	Stages
Classic MIPS R2000	1985	5
Intel Pentium	1993	5
Pentium Pro	1995	12 to 14
Pentium 4 Willamette	2000	20
Pentium 4 Prescott	2004	31
Intel Core 2	2006	14

3. The cautionary tale is Intel NetBurst. Willamette shipped in November 2000 with a 20-stage pipeline. Prescott followed in February 2004 with 31 stages. The design goal was very high clock speed, and Intel spoke publicly about reaching 10 GHz.
4. It did not work, for three reasons that all reinforce each other.
 1. Deeper pipelines make every branch misprediction more expensive, because more work must be thrown away. Prescott’s penalty was well over 30 cycles.
 2. Every extra stage adds pipeline registers, and those flip-flops burn power and add setup and hold overhead that eats the timing gain.
 3. Leakage current at 90 nm rose faster than expected, so the promised clock speeds never arrived. Intel stopped at 3.8 GHz in November 2004.
5. Intel cancelled the successors, Tejas and Jayhawk, in 2004. They were reported to target between 40 and 50 stages. Top NetBurst parts had a thermal design power of 115 W, which was extreme for the time.
6. Intel abandoned NetBurst and shipped the Core microarchitecture in July 2006, which went back to a 14-stage pipeline at lower clocks and was substantially faster in real work.
7. The lesson, stated properly: pipeline depth trades clock frequency against misprediction cost and power. There is an optimum, and it is well short of 30 stages for general-purpose code. Modern cores sit at roughly 14 to 20 stages from fetch to retire.
8. Measurement: `perf stat -e cycles,instructions,stalled-cycles-frontend, stalled-cycles-backend` on Linux separates front-end stalls, which are usually fetch and branch problems, from back-end stalls, which are usually memory and dependency problems.

■ 10.7.6 WORDS – remember these

1. Pipelining — overlapping the stages of different instructions — splitting the datapath into stages separated by pipeline registers.
2. Throughput — how often results come out — instructions completed per unit time.
3. Latency — how long one thing takes end to end — cycles from issue to completion for a single instruction.
4. Hazard — a reason the overlap cannot happen — structural, data or control conflict preventing the next stage from proceeding.
5. Forwarding — passing the answer straight back — bypass paths routing an ALU result to a dependent instruction before write-back.
6. Stall — waiting — holding earlier stages while a dependency resolves.
7. Bubble — an empty slot moving down the pipe — an injected no-operation filling a stalled stage.
8. Pipeline depth — how many stages there are — the count of stages from fetch to retire, a trade-off with clock speed.

10.8 Going wider and smarter

■ 10.8.1 PLAIN – in simple words

1. Pipelining gets you to about one instruction per cycle. Then what.
2. You build more of everything. Two adders, three adders, six adders. Fetch several instructions at once. Finish several at once.
3. That is **superscalar**: more than one instruction completed per cycle.
4. But instructions often wait for each other. If instruction 2 needs the answer from instruction 1, doubling the hardware helps nothing.
5. So the CPU looks further ahead. If instruction 5 is ready and instruction 2 is not, it runs instruction 5 first. That is **out-of-order execution**.
6. It still has to pretend everything happened in order, in case something goes wrong. So results are held back and released in the original order.
7. And when it reaches a branch and does not yet know the answer, it guesses, and carries on as if the guess were right. That is **speculation**.
8. If the guess was right, the work is kept. If wrong, it is all thrown away and the CPU restarts down the other road.
9. Modern CPUs guess correctly the great majority of the time, which is the only reason this works at all.

■ 10.8.2 PLAIN – a picture in your head

1. Picture a busy kitchen with six cooks instead of one.
2. Orders arrive on a rail in strict order. A dispatcher reads them.
3. Order 2 needs a sauce that order 1 is still making, so the dispatcher skips it and gives order 5, which needs nothing, to a free cook.
4. Finished dishes go to a shelf, not to the customer. The shelf is served strictly in the original order, so the customers never notice the reshuffle.
5. That shelf is the **reorder buffer**.
6. Now the head waiter sees a customer approaching the till and guesses they will order the usual. The kitchen starts making it before they speak.
7. Nine times out of ten the waiter is right and the food is already there. One time in ten the food is binned.

Where this comparison breaks: binned food in a real kitchen is wasted money, and binned speculative work in a CPU is only wasted time and energy. But it is not perfectly invisible. The traces speculation leaves in the caches are exactly what Spectre reads, which is why guessing turned out to be a security problem in 2018.

■ 10.8.3 PLAIN – a worked example

1. Look at why renaming is needed. Take this sequence.

```

1: mul  rax, rbx      ; slow, takes several cycles
2: mov  rcx, rax      ; needs the result of 1
3: mov  rax, 5        ; writes rax again, needs nothing
4: add  rdx, rax      ; needs the new rax

```

2. Instruction 3 does not depend on anything. It could run immediately.
3. But it writes RAX, and instruction 2 still needs the old RAX. If 3 ran early, it would destroy the value 2 is waiting for.
4. This is a false dependency. The two instructions do not share data, only a name. It is a collision of labels, not of information.
5. The fix: give them different physical boxes. Say RAX currently lives in physical register P17. Instruction 3 is told to write P42 instead, and from then on RAX means P42.
6. Now instruction 2 reads P17 and instruction 3 writes P42. They are independent, and both can run at once.
7. That relabelling is **register renaming**, and modern cores do it for every single instruction, every cycle.
8. Now count the win. Suppose the multiply takes 4 cycles. Without renaming the sequence takes about 7 cycles. With renaming, instructions 3 and 4 run in parallel with the multiply and the sequence takes about 5.

■ 10.8.4 PLAIN – what is really happening inside

1. Here is the shape of a modern core's back end, in order.
2. Fetch and predict. The branch predictor supplies a stream of addresses; the fetcher pulls bytes from the L1 instruction cache.
3. Decode into micro-operations, or read them from the micro-op cache.
4. Rename: map each architectural register name onto a free physical register, using a register alias table.
5. Allocate: give each operation a slot in the reorder buffer, which keeps program order, and a slot in a scheduler.
6. Schedule and issue: **reservation stations**, or a unified scheduler, hold operations until every input is ready, then fire them at a free unit.
7. Execute: the arithmetic units, address generation units, load and store units, and vector units all work in parallel.
8. Write back the result into the physical register, and mark the reorder buffer slot as complete.
9. Retire, in strict program order. Only at retirement do results become architecturally real. If an exception happened, everything after it is discarded and never becomes visible.
10. That last rule is what makes exceptions **precise**: the software sees a clean, ordered machine even though the hardware ran a mess.

■ 10.8.5 TECHNICAL – the engineer’s version

1. History. The CDC 6600 of 1964, designed by Seymour Cray, had ten functional units and a scoreboard to track dependencies. Robert Tomasulo’s 1967 algorithm for the IBM System/360 Model 91 added reservation stations and register renaming, and it is still the basis of every design today.
2. The Intel Pentium of March 1993 was the first superscalar x86, 2-wide with its U and V pipes. The Pentium Pro of November 1995 was the first out-of-order x86.
3. Modern structure sizes, all implementation details, not architecture.

Structure	Zen 5, 2024	Golden Cove, 2021
Decode width	8, as two of 4	6
Micro-op cache out	12 per cycle	8 per cycle
Reorder buffer	448 entries	512 entries
Integer ALUs	6	5

4. Branch prediction, in order of invention.
 1. Static: always predict not taken, or the backward-taken and forward-not-taken heuristic, which suits loops. No storage needed.
 2. One-bit: remember the last outcome per branch. A loop of N iterations mispredicts twice, at entry and exit.
 3. Two-bit saturating counter: James E. Smith, 1981. Four states, from strongly not taken to strongly taken. It takes two wrong outcomes to change the prediction, so a loop mispredicts once, not twice.
 4. Two-level adaptive: Tse-Yu Yeh and Yale Patt, 1991. Index a table of counters using a history register of recent branch outcomes.
 5. gshare: Scott McFarling, 1993. Combine the branch address with the global history by XOR, which uses the table far better.
 6. Perceptron: Daniel Jimenez and Calvin Lin, 2001. A simple neural predictor, able to use very long histories.
 7. TAGE: Andre Seznec and Pierre Michaud, 2006. Several tagged tables indexed with geometrically increasing history lengths, so short and long correlations are both captured. TAGE-SC-L, with a statistical corrector and a loop predictor, won the Championship Branch Prediction contests and is the basis of what ships today.
5. Real accuracy, in three honest lines.
 1. On many workloads modern predictors get above 99 percent of conditional branches right.
 2. On hard integer code the figure is lower. The 2019 paper “Branch Prediction Is Not A Solved Problem” measured about 95 percent average accuracy for an 8 KB TAGE-SC-L across SPEC CPU 2017 integer benchmarks.
 3. That same paper argues the remaining misses are worth an 18.5 percent instructions-per-cycle opportunity at current core sizes, rising to 55.3 percent for a core four times wider.
6. Real cost of one miss. Measured on Intel Skylake: 16.5 cycles average when the correct path hits in the micro-op cache, 19 to 20 cycles when it does not. At 5 GHz that is roughly 3.3 to 4 nanoseconds. On a core that can retire 6 instructions per cycle, a single misprediction throws away on the order of 100 instruction slots.
7. Two-bit counter state machine, worth memorizing.

```

    taken ->      taken ->      taken ->
[00]----->[01]----->[10]----->[11]

```

strong NT	weak NT	weak T	strong T
<-----	<-----	<-----	
not taken	not taken	not taken	
predict NT	predict NT	predict T	predict T

- Measurement: `perf stat -e branches,branch-misses ./program` gives the real misprediction rate for your code. A rate above about 2 percent on hot code usually means a data-dependent branch that should be turned into branchless arithmetic or a conditional move.

■ 10.8.6 WORDS – remember these

- Superscalar — more than one instruction finished per cycle — multiple issue, with duplicated execution units.
- Out-of-order execution — run whatever is ready — dynamic scheduling with in-order retirement.
- Register renaming — giving the same name different boxes — mapping architectural registers to a larger physical register file.
- Reorder buffer — the shelf that restores order — a circular buffer holding results until in-order retirement.
- Reservation station — a waiting bay for an operation — a scheduler entry holding a micro-op until its operands are ready.
- Speculation — acting on a guess — executing past an unresolved branch and discarding on a miss.
- Branch predictor — the thing that guesses — hardware predicting direction and target, today TAGE-style.
- Precise exception — software sees a tidy machine — the guarantee that on a fault, all earlier instructions completed and none later did.

10.9 SIMD and accelerators

■ 10.9.1 PLAIN – in simple words

- Very often a program does the same sum to a long list of numbers.
- Brighten every pixel. Add every element of two arrays. Scale every audio sample.
- Doing them one at a time wastes the machine, because the operation is identical each time. Only the data changes.
- So CPUs got wide registers that hold several numbers side by side, and instructions that do the same sum to all of them at once.
- This is **SIMD**: single instruction, multiple data.
- A 256-bit register holds eight 32-bit numbers. One add instruction adds all eight pairs in one go. Eight times the work, one instruction.
- That is not the same as having eight cores. It is one core doing one instruction that happens to be eight-wide.
- Later, chips added units for specific jobs: encryption, and now matrix multiply for neural networks.

■ 10.9.2 PLAIN – a picture in your head

- A normal instruction is a single hole punch: one hole per press.
- A SIMD instruction is a row of eight punches on one bar. One press, eight holes, all in line.
- The catch is obvious. The paper must be lined up. If your eight numbers are scattered around memory, you spend more time gathering them into a row than you save by punching them together.

- Also, the bar punches all eight or none. If you only want six of them, you need a mask that blocks two of the punches.

Where this comparison breaks: the punch bar is fixed at eight, but Arm's SVE deliberately does not fix the width. The same program runs on hardware with 128-bit or 512-bit vectors without recompiling. That is a genuinely different design idea, not just a bigger bar.

■ 10.9.3 PLAIN – a worked example

- Add two arrays of eight floating point numbers, using AVX on x86-64. RDI points to array a, RSI to array b, RDX to the result.

```
vmovups ymm0, [rdi]      ; load 8 floats from a
vmovups ymm1, [rsi]      ; load 8 floats from b
vaddps ymm0, ymm0, ymm1  ; 8 adds in one instruction
vmovups [rdx], ymm0      ; store 8 results
```

- Four instructions do 8 additions plus 24 element moves. The scalar version would need a loop of 8 iterations with about 4 instructions each.
- Same idea in ARM64 NEON, which is 128 bits wide, so four floats at a time.

```
ld1 {v0.4s}, [x0]        ; load 4 floats
ld1 {v1.4s}, [x1]        ; load 4 floats
fadd v0.4s, v0.4s, v1.4s  ; 4 adds at once
st1 {v0.4s}, [x2]        ; store 4 results
```

- Read v0.4s as “register v0 treated as 4 single-precision values”. Change it to v0.2d and the same register is 2 doubles. Change it to v0.16b and it is 16 bytes. The bits do not move. Only the interpretation changes, which is the numbers-and-meaning idea from Chapter 7.
- With AVX-512, ZMM registers are 512 bits, so the same loop handles 16 floats per instruction.
- Real speedups are usually below the theoretical multiple. Memory bandwidth, alignment and loop overhead all take a cut. Two to six times is typical for an eight-wide operation on real code.

■ 10.9.4 PLAIN – what is really happening inside

- Physically, a SIMD unit is several copies of the same arithmetic circuit, fed from one wide register and controlled by one decoded instruction.
- There is only one instruction fetch, one decode, one schedule. That saving is a large part of why SIMD is efficient in energy, not just in time.
- Wide units are hot. Running 512-bit operations flat out draws so much current that some Intel chips reduced their clock frequency while doing it, a well-documented behaviour on Skylake-based Xeon parts from 2017.
- Then there are fixed-function accelerators. Instead of a general unit, put an entire algorithm in silicon.
- AES encryption is the classic case. One instruction performs one complete round of the cipher, something that takes dozens of ordinary instructions.
- Now the same idea has arrived for neural networks. The core operation there is multiplying matrices, so chips now include matrix units and separate neural processing units on the same die.
- Those units are not general. They do one shape of arithmetic extremely efficiently and are useless for anything else.

■ 10.9.5 TECHNICAL – the engineer’s version

1. The term SIMD comes from Michael Flynn’s 1966 taxonomy of parallel machines.
2. The x86 vector history, with widths and dates.

Extension	Year	Width
MMX	1997	64 bits, integer only
SSE	1999	128 bits, single float
SSE2	2000	128 bits, double, integer
AVX	2011	256 bits
AVX2	2013	256 bits, integer, FMA
AVX-512	2016 to 2017	512 bits, 32 registers

3. MMX arrived with the Pentium MMX in January 1997 and reused the x87 floating point registers, so you could not mix the two.
4. SSE arrived with the Pentium III in February 1999 and finally gave separate XMM registers. AVX arrived with Sandy Bridge in January 2011 and introduced the three-operand VEX encoding, so the destination need not also be a source.
5. AVX-512 first shipped in Knights Landing in 2016 and in Xeon Skylake-SP during 2017. Consumer support has been inconsistent: present on Rocket Lake, disabled on Alder Lake, and supported on AMD Zen 4 from 2022 and Zen 5 from 2024, with Zen 5 desktop parts having a full 512-bit wide data path. Intel announced AVX10 in July 2023 to tidy this up.
6. Arm’s line runs in parallel.
 1. Advanced SIMD, marketed as NEON, is part of ARMv7-A and mandatory in ARMv8-A, with 32 registers of 128 bits.
 2. SVE was announced at Hot Chips in August 2016. It is vector-length agnostic, permitting 128 to 2048 bits in 128-bit steps.
 3. Fujitsu’s A64FX, used in the Fugaku supercomputer from 2020, implements 512-bit SVE. SVE2 arrived with Armv9-A, announced in March 2021.
7. AES-NI arrived with Intel Westmere in January 2010. Six instructions: AESENC, AESENCLAST, AESDEC, AESDECLAST, AESIMC and AESKEYGENASSIST. One AESENC performs a complete AES round. Beyond speed, the crucial property is that it is constant time and uses no lookup tables, which removes the cache-timing side channels that plagued table-based software AES. Armv8 has the equivalent AESE and AESD instructions.
8. Matrix and neural units on the same die.
 1. Intel Advanced Matrix Extensions, AMX, shipped in Sapphire Rapids Xeon in January 2023: eight tile registers of 1 KB each with a TMUL unit.
 2. Arm’s Scalable Matrix Extension, SME, is part of Armv9 and appears in recent Apple silicon.
 3. Separate NPUs: Intel Meteor Lake in December 2023 was the first Core part with one; Lunar Lake in September 2024 raised it to 48 TOPS. Microsoft’s Copilot+ PC badge requires at least 40 TOPS.
 4. In phones, the Snapdragon 8 Elite Gen 5, announced 24 September 2025, has a Hexagon NPU that Qualcomm states is 37 percent faster than the previous generation. Apple’s A19 Pro, from September 2025, has a 16-core Neural Engine.
9. Separate what is what, as the honesty rules require.
 1. Established fact: the silicon exists, it multiplies matrices at low precision far more efficiently than the general CPU cores, and it is shipping in ordinary phones and laptops today.

2. Active research: how to split a model across CPU, GPU and NPU, which number formats to use, and how to keep memory bandwidth from becoming the limit.
3. Marketing claim: TOPS figures. A TOPS number depends entirely on the number format assumed, is usually a peak that no real workload reaches, and is not comparable between vendors. Treat it as a rough class, not a measurement.
10. Chapters 46 onwards return to this: what those matrix units actually compute, and why a neural network reduces to matrix multiply.
11. Tools: on Linux `cat /proc/cpuinfo` lists `avx2`, `avx512f`, `aes` and so on in the flags line. On macOS, `sysctl hw.optional` lists the Arm features present.

■ 10.9.6 WORDS – remember these

1. SIMD — one order, many numbers — single instruction, multiple data, a lane parallel execution model.
2. Vector register — a wide box holding several numbers — XMM at 128, YMM at 256, ZMM at 512 bits on x86-64.
3. Lane — one slot in the row — one element position within a vector register.
4. Mask register — a stencil saying which lanes count — k0 to k7 on AVX-512, predicate registers on SVE.
5. Vector-length agnostic — the same code on any width — the SVE model where the hardware vector length is discovered at run time.
6. AES-NI — encryption built into the metal — x86 instructions performing AES rounds directly, constant time.
7. NPU — a chunk of silicon for neural maths — neural processing unit, a fixed-function matrix and activation engine.
8. TOPS — a marketing speed number — tera-operations per second, precision dependent and not comparable across vendors.

10.10 Many cores

■ 10.10.1 PLAIN – in simple words

1. Until about 2005, each new chip simply ran at a higher clock speed.
2. Then that stopped. Clocks have barely moved since. A fast desktop chip in 2004 ran at 3.8 GHz, and a fast desktop chip in 2025 boosts to 5.7 GHz. That is not much in twenty years.
3. The reason is heat. Pushing the clock higher needs more voltage, and the power a chip burns rises with the square of the voltage. The heat became impossible to remove.
4. So instead of one faster CPU, chipmakers put several CPUs on one piece of silicon. Each one is a **core**.
5. Two cores can do two things at once, genuinely, in parallel.
6. That only helps if the software can be split into pieces that run at the same time. Many programs cannot be split much, and for those, more cores do very little.
7. Cores also have to agree about memory. If two cores both hold a copy of the same number and one changes it, the other must find out.
8. Keeping them in agreement is called **cache coherence**, and it is one of the hardest parts of building a modern chip.

■ 10.10.2 PLAIN – a picture in your head

1. One cook working faster has a limit: hands only move so quickly, and past a point the kitchen catches fire.
2. So you hire eight cooks. Now eight dishes can be made at once.
3. But if the recipe says “wait for the stock to reduce for an hour”, eight cooks do not make the stock reduce in less than an hour. That step is stubbornly serial.
4. Worse, the cooks share one fridge. If two cooks both take out the butter, and one of them salts it, the other is holding stale butter.
5. So the kitchen needs a rule: before you change something, shout, and everyone else throws their copy away.
6. Shouting takes time, and the more cooks there are, the more shouting.

Where this comparison breaks: cooks can talk to each other in any order, but cores must agree on a strict order of memory events, or programs break in ways that only show up once in a million runs. That is memory ordering, and it is much stricter and much more subtle than shouting.

■ 10.10.3 PLAIN – a worked example

1. Amdahl’s law, from Gene Amdahl in 1967, tells you the ceiling.
2. Say 95 percent of your program can run in parallel and 5 percent cannot.
3. The formula for speedup with n cores is $1 \text{ divided by } ((1 \text{ minus } p) \text{ plus } p \text{ divided by } n)$, where p is the parallel fraction.
4. With p equal to 0.95, work it out.

Cores	Calculation	Speedup
2	$1 / (0.05 + 0.475)$	1.90
4	$1 / (0.05 + 0.2375)$	3.48
8	$1 / (0.05 + 0.11875)$	5.93
16	$1 / (0.05 + 0.059375)$	9.14
64	$1 / (0.05 + 0.0148)$	15.4
Infinite	$1 / 0.05$	20.0

5. Read the last line again. Even with infinite cores, a program that is 5 percent serial can never go more than 20 times faster.
6. At 16 cores you already get 9.14 of a possible 20. Doubling to 32 gets you to 12.6. The returns collapse quickly.
7. If only 50 percent is parallel, the ceiling is 2 times, no matter what.
8. This single calculation explains why buying more cores often does nothing.

■ 10.10.4 PLAIN – what is really happening inside

1. Take two cores, each with its own L1 cache, sharing memory. Both want the variable X at address A . Follow the standard four-state rule set, MESI.
2. Each cached line is in one of four states: **Modified** (I changed it and nobody else has it), **Exclusive** (I have the only copy and it is clean), **Shared** (others may have it too, clean), **Invalid** (I have nothing).

Step	Action	Core 0 / Core 1
0	Nothing cached	I / I
1	Core 0 reads A	E / I
2	Core 0 writes A	M / I
3	Core 1 reads A	S / S
4	Core 1 writes A	I / M
5	Core 0 reads A	S / S

3. Step by step. At step 1 core 0 misses, fetches the line, and since no other core has it, it takes it Exclusive.
4. At step 2 core 0 writes. Because it already had it Exclusive, no message is needed at all. It flips silently to Modified. Memory is now out of date.
5. At step 3 core 1 asks for the line. Core 0 sees the request, notices it holds a Modified copy, and supplies the fresh data. Both settle at Shared.
6. At step 4 core 1 wants to write. It must first broadcast a request for ownership. Core 0 drops its copy to Invalid. Core 1 becomes Modified.
7. At step 5 core 0 wants to read again, and the whole dance repeats the other way round.
8. Each of those transfers costs tens of cycles. Two cores fighting over one variable in a tight loop is one of the slowest things a modern CPU can do.

■ 10.10.5 TECHNICAL – the engineer’s version

1. The physics. Dynamic power is approximately $P = \alpha C V^2 f$. Alpha is switching activity, C capacitance, V supply voltage, f frequency.
2. Dennard scaling, from Robert Dennard’s 1974 IBM paper, said that as transistors shrink you can lower V and C in step, so power per square millimetre stays constant while frequency rises. That held for thirty years.
3. It broke down around 2005 to 2006. Threshold voltage could not keep falling without leakage current exploding, so supply voltage stalled near 1 volt, so power density started rising with every shrink.
4. Intel cancelled its 4 GHz Pentium 4 in October 2004 and topped out at 3.8 GHz in November 2004. IBM’s POWER4 in 2001 was the first mainstream dual-core chip; AMD’s Athlon 64 X2 and Intel’s Pentium D both arrived in May 2005. That is the exact moment the industry turned.
5. Simultaneous multithreading, stated honestly. SMT duplicates the architectural state, meaning registers, program counter and flags, so one core presents two logical processors to the operating system. It does not duplicate execution units, caches or TLBs. Those are shared.
 1. The gain comes from filling issue slots that one thread leaves empty while it waits for memory. Typical real throughput gain is 10 to 30 percent, not 100 percent.
 2. It can be negative. Two threads with large working sets thrash the shared L1 and L2 and both run slower than one would alone.
 3. Intel shipped Hyper-Threading on Xeon in February 2002 and on the 3.06 GHz Pentium 4 in November 2002.
 4. It has a security cost. Shared resources leak timing information, which produced attacks including PortSmash in 2018 and the MDS family in 2019. OpenBSD disabled SMT by default in 2018.
 5. Intel dropped SMT entirely on Arrow Lake in October 2024. The Core Ultra 9 285K has 24 cores and exactly 24 threads.

6. Cache coherence. MESI comes from Mark Papamarcos and Janak Patel, 1984, and is often called the Illinois protocol. Real chips extend it: MOESI adds Owned, used by AMD, and MESIF adds Forward, used by Intel. Large chips use a directory rather than broadcast snooping, because broadcast does not scale past a few dozen cores.
7. Memory ordering is a separate problem from coherence. Coherence is about one address. Ordering is about the visible order of accesses to different addresses.
 1. x86-64 uses Total Store Order. Loads are not reordered with loads, stores are not reordered with stores, but a store followed by a load to a different address may appear reordered because of the store buffer.
 2. ARM64 is weakly ordered. Almost any reordering is allowed unless you write a barrier.
 3. Barriers: MFENCE, LFENCE and SFENCE on x86-64; DMB, DSB and ISB on ARM64. In portable code, use the C11 or C++11 atomics with an explicit memory order rather than inline assembly.
 4. This is why lock-free code that works on an Intel laptop can fail on an Apple silicon Mac. The bug was always there. x86-64's stronger model was hiding it.
8. False sharing. Two threads write two different variables that happen to sit in the same 64-byte cache line. There is no logical sharing at all, but the line ping-pongs between cores at tens of cycles per bounce.


```

struct bad { long a; long b; };           /* same line */
struct good {
    long a; char pad[56];                /* 8 + 56 = 64 */
    long b;
};
      
```
9. In C++, `alignas(64)` or `std::hardware_destructive_interference_size` expresses this properly. On Linux, `perf c2c record` and `perf c2c report` find false sharing directly by address.
10. Amdahl's law was presented at the 1967 AFIPS Spring Joint Computer Conference. John Gustafson's 1988 rebuttal points out that in practice people grow the problem to fit the machine, and for a fixed time budget rather than a fixed problem size the scaling looks much better. Both are correct; they answer different questions.
11. Tools: `nproc` and `lscpu` show cores and threads, `taskset` pins a process to specific cores, and `numactl --hardware` shows memory locality on multi-socket machines.

■ 10.10.6 WORDS - remember these

1. Core — one complete CPU on the die — an independent instruction stream with its own registers, L1 and L2.
2. Power wall — the point where heat stopped the clock rising — the end of frequency scaling around 2005 due to power density.
3. Dennard scaling — shrinking used to be free — constant power density under scaling, broken since roughly 2006.
4. SMT — one core pretending to be two — simultaneous multithreading, duplicated architectural state over shared execution resources.
5. Cache coherence — everyone agrees what a given address holds — MESI and its relatives, enforced by snooping or a directory.
6. Memory ordering — the agreed order of unrelated accesses — TSO on x86-64, weak on ARM64, controlled by barriers.
7. False sharing — accidental fighting over one line — unrelated variables in the same 64-byte cache line bouncing between cores.

8. Amdahl's law — the serial part sets the ceiling — speedup equals $1 / ((1 - p) + p/n)$.

10.11 Caches from the CPU side

■ 10.11.1 PLAIN – in simple words

1. Main memory is desperately slow compared to the CPU. Fetching from it takes the time of hundreds of instructions.
2. So the CPU keeps small, fast copies of recently used memory close by. Those are **caches**.
3. There are usually three levels. L1 is tiny and instant, L2 is bigger and slower, L3 is much bigger and slower again.
4. Nothing is copied one byte at a time. Memory is moved in blocks of 64 bytes, called a **cache line**.
5. So if you read one byte, you get its 63 neighbours for free. That is why walking through an array in order is enormously faster than jumping about.
6. Caches work because of two habits real programs have. If you used something recently you will probably use it again soon, and if you used something you will probably use the thing next to it.
7. Those two habits have names: temporal locality and spatial locality. Almost all of computer performance rests on them.

■ 10.11.2 PLAIN – a picture in your head

1. Picture a library with a huge basement and a small desk.
2. You cannot work in the basement. You fetch books up to the desk.
3. Fetching one book takes twenty minutes, so you never fetch one book. You fetch the whole shelf section it sits in. That section is the cache line.
4. Your desk holds four books. The table behind you holds fifty. The room next door holds a thousand. Those are L1, L2 and L3.
5. When your desk is full and you need a new book, you must put one back. Which one you choose is the replacement policy.
6. If you always ask for books that are near each other, you almost never go to the basement. If you ask for random books, you go every time.

Where this comparison breaks: you decide what to fetch, and the CPU mostly does not. The hardware guesses your next request and fetches it before you ask. That guessing is prefetching, and when it works you never see the basement at all.

■ 10.11.3 PLAIN – a worked example

1. Real figures for AMD Zen 5, the Ryzen 9000 desktop series from 2024.

Level	Size	Latency	Time at 5.7 GHz
L1 instruction	32 KB per core	-	-
L1 data	48 KB per core	4 cycles	0.70 ns
L2	1 MB per core	14 cycles	2.46 ns
L3	32 MB per die	about 50 cycles	8.8 ns
DDR5-6000	32 GB	-	about 75 ns

The L1 and L2 cycle counts are AMD's published figures. The L3 figure is approximate: AMD quoted 50 cycles for Zen 4, and independent measurements of Zen 5 land close to it but vary with the part and the memory clock.

2. Read the last two rows together. Going to memory instead of L3 costs about nine times longer. Going to memory instead of L1 costs over a hundred times longer.
3. Now measure it yourself. Walk a large array in order, then in random order.

```
/* sequential: hardware prefetch works, ~1 to 2 ns per element */
for (i = 0; i < n; i++) sum += a[i];

/* random: every access is a cache miss and a TLB risk */
for (i = 0; i < n; i++) sum += a[idx[i]]; /* idx shuffled */
```

4. On a current desktop, with an array well over the L3 size, the sequential loop typically runs at roughly 1 to 2 nanoseconds per element and the random loop at roughly 70 to 120 nanoseconds per element. Those are approximate and machine dependent, but the ratio, somewhere around 50 to 100 times, is consistent everywhere.
5. Same instructions. Same number of additions. Same array. The only difference is the order of the addresses.
6. This is the single most important performance fact in the chapter.

■ 10.11.4 PLAIN – what is really happening inside

1. When the core wants an address, the cache must answer “do I have it” in one or two cycles. It cannot search. It must go straight to one place.
2. So the address is cut into three fields.
 1. The **offset**: which byte inside the 64-byte line.
 2. The **index**: which set of the cache to look in.
 3. The **tag**: the rest of the address, stored alongside the data to prove which line this actually is.
3. **Direct-mapped** means one place per index. Fast and simple, but two hot addresses that share an index knock each other out forever.
4. **Fully associative** means a line may sit anywhere. No conflicts, but you must compare every tag, which is impossibly expensive at any size.
5. **Set-associative** is the compromise, and it is what everyone uses. Each index selects a small set of, say, 8 or 12 slots, and only those tags are compared. You get most of the conflict resistance for a small cost.
6. When a set is full, something must go. The replacement policy picks the victim, usually an approximation of least recently used.
7. On a write, two choices. **Write-through** sends every write onward immediately, which is simple but floods the bus. **Write-back** marks the line dirty and only writes it out when it is evicted. Every modern data cache is write-back.

■ 10.11.5 TECHNICAL – the engineer’s version

1. Work the address split by hand for Zen 5’s L1 data cache: 48 KB, 12-way set associative, 64-byte lines.
 1. Lines total: $48 \times 1024 / 64 = 768$.
 2. Sets: $768 / 12 = 64$.
 3. Offset bits: $\log_2(64) = 6$.
 4. Index bits: $\log_2(64) = 6$.
 5. Tag bits: everything above bit 11.
2. Take the address 0x12345678 and split it.

```
0x12345678 = 0001 0010 0011 0100 0101 0110 0111 1000
               |<----- tag ----->|index | off |
```

```

tag    = 0x12345   (bits 31..12)
index  = 011001b = set 25   (bits 11..6)
offset= 111000b = byte 56  (bits 5..0)

```

3. Now a direct-mapped counter-example: a 32 KB direct-mapped cache with 64-byte lines has 512 lines, so 9 index bits. Any two addresses exactly 32 KB apart share an index. A loop touching two arrays that happen to be 32 KB apart will miss on every single access. That is a conflict miss, and it is why associativity exists.
4. The three kinds of miss, called the three Cs after Mark Hill's 1987 thesis: compulsory (first ever touch), capacity (working set exceeds the cache) and conflict (fits, but the sets collide).
5. Replacement policies in shipping hardware: true LRU up to about 4 ways; tree-based pseudo-LRU beyond that; RRIP and its dynamic variant from a 2010 ISCA paper for last-level caches, which resist thrashing far better; pure random in some Arm L2 designs, chosen for its tiny cost.
6. Write policy detail: write-back with write-allocate is standard for data caches. A dirty bit per line tracks whether a write-back is needed. Stores to memory marked write-combining, typically video memory, bypass the cache through separate combining buffers.
7. Hardware prefetchers, per level and multiple per level. Next-line, stride detectors that spot a constant step, stream prefetchers that follow several sequences at once, and region-based prefetchers. They can be counter-productive on pointer-chasing code, and they can be disabled through model-specific registers on both Intel and AMD parts.
8. Cache line size is 64 bytes on x86-64 and on most Arm cores. Apple's M series uses 128 bytes, which is a real portability trap for anyone padding structures by hand.
9. Inclusion policy is an implementation detail, not a standard. Intel's ring-based designs historically used an inclusive L3; AMD's L3 is a victim cache, holding only lines evicted from L2, which is why AMD's L3 is quoted per core complex die rather than per chip.
10. Tools: `lscpu -C` prints every level with size, ways and line size; `getconf LEVEL1_DCACHE_LINESIZE` prints the line size; `perf stat -e cache-references,cache-misses,LLC-load-misses` measures the real miss rate; `valgrind --tool=cachegrind` simulates it per line of source code.

■ 10.11.6 WORDS – remember these

1. Cache line — the block memory moves in — 64 bytes on x86-64 and most Arm, 128 on Apple silicon.
2. Tag, index, offset — proof, shelf, and position on the shelf — the three fields an address is split into for lookup.
3. Set-associative — a few slots per index — N-way lookup comparing N tags in parallel.
4. Conflict miss — two hot addresses on the same shelf — a miss that would not happen in a fully associative cache of the same size.
5. Write-back — write later, when evicted — a dirty bit marks lines needing a write-out, as opposed to write-through.
6. Prefetching — fetching before you ask — hardware or software issuing loads ahead of demand based on detected patterns.
7. Locality — you reuse things, and their neighbours — temporal and spatial locality, the reason caches work at all.

10.12 Interrupts, exceptions and privilege

■ 10.12.1 PLAIN – in simple words

1. The loop of fetch, decode, execute never stops on its own. Something must be able to break into it.
2. Two kinds of thing break in.
3. An **interrupt** comes from outside: a key was pressed, a network packet arrived, a timer expired. It has nothing to do with the current instruction.
4. An **exception** comes from inside: a divide by zero, an instruction the chip does not know, a memory address that is not mapped.
5. Either way the CPU stops, remembers exactly where it was, jumps to a fixed handler routine, runs it, and then goes back.
6. It also changes mode. Handlers run with more power than ordinary programs.
7. That mode change is the whole basis of operating system security. Your program cannot touch the disk directly. It must ask, and asking means deliberately triggering a controlled jump into the more powerful mode.

■ 10.12.2 PLAIN – a picture in your head

1. Picture a receptionist working through a pile of forms.
2. The phone rings. They put a bookmark in the pile, note exactly where they were, answer the phone, deal with it, and return to the exact same form.
3. The phone number is not decided on the spot. There is a printed list on the wall: “for fire, call this room; for deliveries, call that room”. That list is the interrupt vector table.
4. Some rooms need a key the receptionist does not have. To get in, they must go through a specific door with a guard, who checks them and hands over the key on the way in and takes it back on the way out.
5. That guarded door is a system call.

Where this comparison breaks: the receptionist chooses when to look up. A CPU can be interrupted between any two instructions, whether it wants to be or not, and the hardware, not the software, decides when.

■ 10.12.3 PLAIN – a worked example

1. Print thirteen characters to the screen. On Linux, x86-64.

```

mov rax, 1          ; 1 = the write system call
mov rdi, 1          ; file1 = standard output
lea rsi, [msg]      ; address of the text
mov rdx, 13         ; how many bytes
syscall            ; cross into the kernel
```

2. The same thing on Linux, ARM64.

```

mov x8, #64         ; 64 = the write system call
mov x0, #1          ; file 1 = standard output
adr x1, msg         ; address of the text
mov x2, #13         ; how many bytes
svc #0             ; cross into the kernel
```

3. Notice the shape is identical. Put a number saying which service, put the arguments in agreed registers, execute one special instruction.
4. Notice the register names differ, the call numbers differ, and the instruction differs. All three are conventions of that operating system on that architecture, not laws of the machine.

5. The `syscall` and `svc` instructions do almost nothing themselves. They save the return address, raise the privilege level, and jump to one fixed address the kernel installed at boot. The kernel does the rest.

■ 10.12.4 PLAIN – what is really happening inside

1. Between instructions, the core checks whether an interrupt is pending. If interrupts are enabled and one is pending, it acts.
2. It finishes the instructions already in flight and discards the speculative ones, so the machine has a clean, precise state.
3. It looks up the vector number in a table of handler addresses whose location is held in a special register.
4. It pushes enough state to come back: at minimum the return address and the flags, and on x86 also the stack and code segment selectors and, for some faults, an error code.
5. It raises the privilege level and jumps to the handler.
6. The handler immediately saves everything else it will use, because the hardware only saved the minimum.
7. When finished, a return-from-interrupt instruction restores the saved state and drops the privilege level in one atomic step.
8. All of this costs real time. A modern system call is on the order of tens to a few hundred nanoseconds, and the Meltdown mitigations of 2018 made it noticeably more expensive by forcing a page-table switch on every crossing.

■ 10.12.5 TECHNICAL – the engineer's version

1. x86-64 uses the Interrupt Descriptor Table, located by the IDTR register. It has 256 entries. Vectors 0 to 31 are architecturally defined exceptions; 32 to 255 are available for devices and software.

Vector	Name	Cause
0	#DE	Divide error
6	#UD	Invalid opcode
13	#GP	General protection fault
14	#PF	Page fault

2. Exceptions are classified as faults, which re-run the instruction after the handler, traps, which continue after it, and aborts, which do not return. A page fault is a fault; that is exactly how demand paging works.
3. ARM64 uses a vector table located by `VBAR_EL1`, with 16 entries of 128 bytes each.
 1. The 16 entries are four exception types, which are synchronous, IRQ, FIQ and SError, crossed with four sources.
 2. The four sources are the current exception level using `SP0`, the current level using `SPx`, a lower level in `AArch64`, and a lower level in `AArch32`.
 3. State is saved into `ELR_EL1` for the return address, `SPSR_EL1` for the saved processor state, and `ESR_EL1` for the syndrome describing why.
4. Privilege levels. x86-64 defines rings 0 to 3. In practice only ring 0, the kernel, and ring 3, user code, are used; ring 1 and 2 are historical. Virtualization adds VMX root mode below ring 0, informally called ring -1, and System Management Mode sits outside all of it.

5. ARM64 defines exception levels instead: EL0 for applications, EL1 for the kernel, EL2 for a hypervisor, EL3 for the secure monitor. Higher number means more privilege, the opposite direction to x86 rings.
6. SYSCALL on x86-64 was introduced by AMD and is deliberately minimal.
 1. It saves RIP into RCX and RFLAGS into R11, then masks RFLAGS using IA32_FMASK.
 2. It loads the code and stack selectors from IA32_STAR and jumps to the address held in IA32_LSTAR.
 3. It performs no memory reads and no descriptor table lookup. That is precisely why it is fast compared with the old `int 0x80` gate.
7. Because SYSCALL destroys RCX, the Linux x86-64 system call ABI passes its fourth argument in R10 rather than RCX. The number goes in RAX, arguments in RDI, RSI, RDX, R10, R8, R9, and the result comes back in RAX. On ARM64 the number goes in X8, arguments in X0 to X5, result in X0.
8. Kernel page-table isolation, the Meltdown fix, unmaps almost all kernel memory while user code runs, so each crossing writes CR3 and disturbs the TLB. On processors without PCID tagging the cost is severe. This is a concrete case where a security fix has a permanent, measurable performance price.
9. The vDSO exists to avoid the crossing entirely for cheap, common calls. Linux maps a small shared page into every process so that `clock_gettime` and similar can read a kernel-updated timestamp without a system call.
10. Tools: `strace ./program` traces every system call and its arguments, `cat /proc/interrupts` shows interrupt counts per CPU per device, and `perf trace` gives timing per call.

■ 10.12.6 WORDS – remember these

1. Interrupt — something outside wants attention — an asynchronous event from a device, delivered between instructions.
2. Exception — the current instruction went wrong — a synchronous fault, trap or abort caused by execution itself.
3. Vector table — the printed list of who to call — IDT on x86-64, the table at `VBAR_EL1` on ARM64.
4. Context switch — putting a bookmark in and picking up another book — saving and restoring register state to change what is running.
5. Privilege ring — how much the running code is allowed to do — rings 0 to 3 on x86-64, exception levels EL0 to EL3 on ARM64.
6. System call — knocking on the kernel's door on purpose — SYSCALL on x86-64, SVC on ARM64, with the number in a register.
7. Page fault — the address had no memory behind it — a fault that lets the kernel map a page and re-run the instruction.

10.13 Reading a real spec sheet

■ 10.13.1 PLAIN – in simple words

1. A CPU page is a wall of numbers. Almost all of them are simple once you know what each one is measuring.
2. Cores means how many complete CPUs are on the chip. Threads means how many programs the operating system can schedule at once, which may be the same number or twice it.
3. Base clock is what the chip guarantees under a heavy load on all cores. Boost clock is the best it will ever do, briefly, on one core, when cool.
4. Cache numbers tell you how much fast memory sits between the cores and main memory, at each of three levels.
5. TDP is a cooling target in watts. It is not the power the chip draws.
6. Process node, such as “3 nm”, is a marketing name for a manufacturing generation, not a measurement of anything on the chip.
7. Socket says which motherboard it physically fits.
8. Memory channels and PCIe lanes say how much data can get in and out.

■ 10.13.2 PLAIN – a picture in your head

1. Compare it to reading a car’s specification.
2. Cores are cylinders. Threads are how many drivers the dashboard pretends there are.
3. Base clock is the speed you can hold all day up a hill. Boost clock is what the speedometer reaches downhill with a tailwind, for thirty seconds.
4. Cache is the fuel already in the lines. TDP is the radiator’s rating.
5. PCIe lanes are the number of doors for luggage.

Where this comparison breaks: a car’s top speed is a property of the car, but a CPU’s boost clock is a property of the car, the road, the weather and the driver together. Cooling, motherboard power delivery and even the individual silicon sample all change it.

■ 10.13.3 PLAIN – a worked example

1. Take a current desktop chip: the AMD Ryzen 9 9950X3D, launched on 12 March 2025. Every number decoded.

Item	Value	What it means
Cores / threads	16 / 32	16 real cores, SMT on
Base clock	4.3 GHz	Guaranteed all-core floor
Boost clock	5.7 GHz	Best case, one core, cool
L1 cache	80 KB per core	32 KB code, 48 KB data
L2 cache	1 MB per core	16 MB in total
L3 cache	128 MB	96 MB stacked, 32 MB plain
TDP	170 W	Cooling design target
Socket	AM5	LGA 1718 mainboard fit
Memory	Dual channel DDR5	Two 64-bit paths
PCIe	5.0, 24 usable lanes	Graphics and storage

2. The 128 MB of L3 is not one pool. This chip has two eight-core dies. One has a 64 MB cache die stacked on top of it, giving that die 96 MB. The other has the normal 32 MB.
3. So threads on the wrong die do not get the big cache. That is why the operating system scheduler has to be told which die is which.

4. Now the phone side: the Qualcomm Snapdragon 8 Elite Gen 5, announced on 24 September 2025.

Item	Value	What it means
Prime cores	2 at up to 4.6 GHz	Third-generation Oryon
Perf cores	6 at up to 3.63 GHz	Same design, lower clock
Threads	8	No SMT, one per core
Process	3 nm class	TSMC generation name
GPU	Adreno at 1.2 GHz	Sliced architecture
NPU	Hexagon	Qualcomm claims +37%
Memory	LPDDR5X	On-package, no slots
Modem	Snapdragon X85	Up to 12.5 Gbps down

5. Notice what is missing from the phone sheet: no TDP, no socket, no memory channels, no PCIe lanes. A phone chip is soldered, its memory is stacked on the package, and its power budget is a thermal envelope of a few watts, not a published number.

■ 10.13.4 PLAIN - what is really happening inside

1. TDP deserves a paragraph of its own, because it is widely misread.
2. TDP is the heat, in watts, that the cooler must be able to remove for the chip to hold its rated behaviour. It is a design target for the cooling system.
3. AMD's socket AM5 parts will draw up to 1.35 times the TDP in package power. So a 170 W part draws up to about 230 W.
4. Intel now publishes two numbers instead: base power and maximum turbo power. On the Core Ultra 9 285K those are 125 W and 250 W.
5. Boost clock is a ceiling, not a promise. It requires a light load, a good core, headroom in the power budget, and cool silicon. All-core sustained clocks under a heavy load are typically several hundred megahertz lower.
6. Cores are no longer identical. Intel's desktop parts mix performance cores and efficient cores with different clocks and different capabilities. Phone chips have done this since Arm's big.LITTLE arrangement in 2011.
7. "Cores plus threads" on a mixed chip is genuinely confusing. The Core Ultra 9 285K has 8 performance cores plus 16 efficient cores, which is 24 cores, and because SMT was removed it has exactly 24 threads.

■ 10.13.5 TECHNICAL - the engineer's version

1. The two current desktop parts side by side, from vendor specifications.

Spec	9950X3D	Core Ultra 9 285K
Launched	March 2025	Q4 2024
Cores	16 uniform	8 P plus 16 E
Threads	32	24
Base clock	4.3 GHz	3.7 GHz P, 3.2 GHz E
Max turbo	5.7 GHz	5.7 GHz
L2 total	16 MB	40 MB
L3	128 MB	36 MB
Power	170 W TDP	125 W base, 250 W max
Socket	AM5	FCLGA1851

Spec	9950X3D	Core Ultra 9 285K
Memory	DDR5, 2 channels	DDR5-6400, 2 channels
PCIe	5.0, 24 usable	5.0 and 4.0, 24 lanes

- Process nodes: the 9950X3D uses TSMC N4P for its core dies with a separate 6 nm class I/O die; the 285K uses a TSMC N3B compute tile in a multi-tile package. In both cases the number is a process generation name. Chapter 3 covered why no dimension on the chip actually measures 3 or 4 nanometres.
- The mobile parts for comparison, from vendor material.

Spec	Snapdragon 8E Gen 5	Apple A19 Pro
Announced	24 Sept 2025	9 Sept 2025
CPU	2 at 4.6, 6 at 3.63	2 at 4.26, 4 at 2.6
Process	3 nm class	TSMC N3P
Memory	LPDDR5X on package	12 GB LPDDR5X
Bandwidth	not published	about 76.8 GB/s
NPU	Hexagon	16-core Neural Engine

- Note that Apple publishes core counts and little else; Qualcomm publishes clocks; neither publishes sustained power. Comparing them properly needs measurement, not spec sheets.
- Memory bandwidth is worth computing rather than reading. Dual-channel DDR5-6000 gives 2 channels times 8 bytes times 6000 million transfers per second, which is 96 GB/s peak. Real achieved bandwidth is typically 70 to 85 percent of that.
- PCIe bandwidth likewise: PCIe 5.0 runs at 32 gigatransfers per second per lane, and with 128b/130b encoding that is about 3.94 GB/s per lane per direction, so a 16-lane graphics slot is about 63 GB/s each way.
- Tools: `lscpu`, `lscpu -C`, `dmidecode -t processor` for socket and package data, `sudo lspci -vv` for actual negotiated PCIe link width and speed, and `turbostat` for live clock, power and temperature.

■ 10.13.6 WORDS – remember these

- Base clock — the speed it can always hold — guaranteed frequency at rated power with all cores loaded.
- Boost clock — the best it ever does — opportunistic maximum for a light load with thermal and power headroom.
- TDP — how much heat the cooler must remove — thermal design power, a cooling target, not consumption.
- PPT, PL1, PL2 — the real power limits — package power tracking on AMD, base and turbo power limits on Intel.
- Socket — the physical fit — LGA 1718 for AM5, FCLGA1851 for Arrow Lake.
- Memory channel — one independent 64-bit path to DRAM — two channels is standard on desktop, more on workstation and server.
- PCIe lane — one serial link pair for expansion — PCIe 5.0 gives about 3.94 GB/s per lane per direction.
- System on chip — everything in one package — CPU, GPU, NPU, modem, memory controller and I/O on one piece of silicon.

10.14 How fast is a CPU really

■ 10.14.1 PLAIN – in simple words

1. Speed is not gigahertz. Gigahertz is how many times per second the clock ticks, and that is only half the story.
2. The other half is how much work the chip gets done per tick.
3. A chip at 3 GHz that does two instructions per tick beats a chip at 4 GHz that does one.
4. Work per tick is called **instructions per cycle**, or IPC.
5. Time taken equals the number of instructions, times cycles per instruction, divided by the clock speed. Those three numbers are the whole of it.
6. A compiler that emits fewer instructions helps. A wider chip that runs more per cycle helps. A higher clock helps. All three multiply together.
7. So you cannot compare two different chips by clock speed alone, and you certainly cannot compare two different instruction sets that way.

■ 10.14.2 PLAIN – a picture in your head

1. Think of a factory line. The clock is how often the conveyor advances.
2. IPC is how many items each advance actually delivers.
3. A line that advances four times a second but usually delivers nothing is worse than one that advances three times a second and always delivers two.
4. Turbo is the line running fast for a short burst while everything is still cool, then slowing down once the machines heat up.
5. Thermal throttling is the safety cut-out that slows the line before anything melts.

Where this comparison breaks: a factory delivers nothing when starved of parts, and so does a CPU, but the starvation is invisible. A core waiting on memory still ticks its clock and still reports full frequency. It just retires almost no instructions. That is why frequency graphs look healthy while programs run slowly.

■ 10.14.3 PLAIN – a worked example

1. Take a program of exactly 1,000,000,000 instructions.
2. CPU A: 4.0 GHz, IPC 1.0. $\text{Time} = 1e9 / (4e9 \times 1.0) = 0.250$ seconds.
3. CPU B: 3.0 GHz, IPC 1.8. $\text{Time} = 1e9 / (3e9 \times 1.8) = 0.185$ seconds.
4. CPU B is 26 percent faster while being 25 percent slower in gigahertz.
5. This is not a made-up case. It is exactly what happened in 2006. The Pentium 4 670 ran at 3.8 GHz. The Core 2 Duo E6600 ran at 2.4 GHz and beat it comfortably per core, because its IPC was far higher.
6. Now add the third term. If a better compiler removes 20 percent of the instructions, CPU A takes 0.200 seconds without any hardware change.
7. Real IPC values worth carrying around: pointer-chasing code that misses cache constantly runs at 0.1 to 0.3; ordinary integer application code runs at roughly 1 to 2.5 on a modern wide core; tight vectorized numeric loops can exceed 4.

■ 10.14.4 PLAIN – what is really happening inside

1. Turbo works like a budget. The chip watches its power draw, its current draw and its temperature, thousands of times a second.
2. While all three are under their limits, it raises the clock. When any one hits its limit, it lowers it.
3. So the same chip in a small case with a weak cooler is genuinely a slower chip than in a big case with a good one. The number on the box does not change; the achieved clock does.
4. Not all cores are equal even within one chip. Manufacturing variation means some cores tolerate higher clocks, and the firmware knows which, and sends single-threaded work there.
5. Thermal throttling is the last-resort mechanism. If temperature reaches the limit, typically around 95 to 100 degrees Celsius, the clock drops sharply until it recovers.
6. This is why a benchmark run for 10 seconds and a benchmark run for 10 minutes give different answers, and why laptop reviews measure both.

■ 10.14.5 TECHNICAL – the engineer’s version

1. The performance equation, often called the iron law of processor performance, is: $\text{CPU time} = \text{instruction count} \times \text{cycles per instruction} \times \text{clock period}$.
2. IPC is the reciprocal of CPI. Each of the three terms is owned by a different party: the compiler and program own instruction count, the microarchitecture owns CPI, and the process and power budget own the clock.
3. Measuring it, on Linux.

```
perf stat -e cycles,instructions,branch-misses ./program
# prints, among other lines:
# 1,842,113,904 cycles
# 3,120,884,551 instructions # 1.69 insn per cycle
```

4. That “insn per cycle” figure counts retired instructions, meaning the ones that actually completed. Speculative work that was thrown away is not counted, which is the correct definition.
5. IPC is not comparable across instruction sets. One x86-64 instruction may do a load, an arithmetic operation and an addressing calculation, while the ARM64 equivalent needs two or three instructions. Comparing IPC between them measures encoding density, not speed.
6. IPC is also not comparable across programs. A memory-bound program has low IPC because it is waiting, and no amount of core width fixes it.
7. Benchmark categories, and what each is honest about.
 1. SPEC CPU 2017: real compiled applications, with separate integer and floating point suites and separate rate and speed variants. Rate measures throughput with many copies; speed measures one job’s time. It is the closest thing to a standard, and results must be submitted with full disclosure of compiler flags.
 2. Geekbench 6 and Cinebench 2024: quick, widely quoted, and dominated by short bursts, so they favour high boost clocks over sustained ability.
 3. Microbenchmarks: measure one instruction’s latency or throughput. Useful for engineers, useless as a purchase guide.
 4. Your own workload, timed: always the best answer if you can get it.
8. Turbo control is exposed as real, documented limits. Intel defines PL1, the long-term power limit, PL2, the short-term limit, and tau, the time constant over which the average is taken. AMD defines PPT for package power, TDC for sustained current and EDC for peak current, with Precision Boost 2 adjusting clocks continuously against all three plus temperature.
9. Where experts disagree: whether single-number benchmark scores are useful at all. One camp

says a composite score is the only practical way for a buyer to compare; the other says composites hide the fact that different workloads rank chips in different orders. Both are right in their own context, and the safe habit is to look at the sub-scores.

10. Tools: `turbostat` on Intel and `zenmonitor` or `ryzen_monitor` on AMD show live per-core clock, package power and temperature. `s-tui` shows throttling under load. `stress-ng` generates the load.

■ 10.14.6 WORDS – remember these

1. IPC — how much gets done per tick — instructions retired per cycle, measured by hardware performance counters.
2. CPI — the same thing upside down — cycles per instruction, the reciprocal of IPC.
3. Iron law — the only performance formula you need — time equals instruction count times CPI times clock period.
4. Retired instruction — one that really counted — an instruction that committed architectural state, excluding discarded speculation.
5. Turbo — a temporary speed rise — opportunistic frequency increase within power, current and thermal limits.
6. Thermal throttling — slowing down to survive — automatic clock reduction when a temperature limit is reached.
7. SPEC CPU 2017 — the serious benchmark — a suite of real applications with mandatory disclosure of build settings.

10.98 Common wrong ideas

1. Wrong: a CPU understands your program. Right: it matches bit patterns to control signals. Nothing is understood at any point.
2. Wrong: a higher clock speed means a faster chip. Right: time equals instruction count times cycles per instruction divided by clock speed, and the middle term differs by more than 2 times between designs.
3. Wrong: instructions and data are stored differently. Right: they are the same bytes in the same memory. Only permission bits and where the program counter points make the difference.
4. Wrong: more cores means proportionally more speed. Right: Amdahl's law caps you at 1 divided by the serial fraction, so 5 percent serial code can never go more than 20 times faster.
5. Wrong: hyper-threading doubles performance. Right: it duplicates architectural state only, and the typical real throughput gain is 10 to 30 percent, sometimes negative.
6. Wrong: RISC chips are simple and CISC chips are complex. Right: both decode into internal micro-operations and both run out of order. The real surviving differences are decode complexity, memory operands and the memory ordering model.
7. Wrong: a deeper pipeline is always faster. Right: the Pentium 4 went from 20 to 31 stages, was cancelled at 3.8 GHz, and was beaten by a 14-stage design in 2006.
8. Wrong: microcode is what runs your program. Right: most instructions never reach the microcode sequencer. It handles complicated cases and provides a patching mechanism.
9. Wrong: cache makes memory faster. Right: cache makes repeated and nearby access faster. Truly random access over a large array gets almost no benefit at all.
10. Wrong: TDP is how much power the chip uses. Right: TDP is a cooling design target. An AMD 170 W part draws up to about 230 W at the package.

10.99 Chapter summary in 20 lines

1. A CPU is one loop repeated forever: fetch, decode, execute, repeat.
2. Its parts are a control unit, an ALU, a register file, a program counter, an instruction register, caches, interconnect and a memory controller.
3. The stored-program idea puts instructions and data in the same memory, which is why one machine runs any program.
4. That same idea permits buffer overflows, self-modifying code and JIT compilation, and it is why the NX permission bit had to be invented.
5. Real CPUs are modified Harvard: one memory underneath, split L1 instruction and data caches on top.
6. An instruction is a number split into fields: opcode, operands, immediates. Addressing modes are the rules that turn those fields into an address.
7. An ISA is a contract, not a design. x86-64 dates from 2000, ARM64 from 2011, RISC-V began at Berkeley in 2010 and is open.
8. Registers are the fastest storage that exists because they are tiny, near, many-ported and addressed by three to five bits.
9. Microcode turns one complex instruction into several micro-operations, and because it is stored rather than wired, it can be patched at boot.
10. That patching mechanism carried the 2018 Spectre mitigations, which added new control registers to a shipped architecture by firmware update.
11. Pipelining overlaps stages so one instruction completes per cycle instead of one every five, and it is why clock speeds could rise at all.
12. Hazards are structural, data and control. Forwarding fixes most data hazards; branch prediction fixes most control hazards.
13. Very deep pipelines lose more on every mispredict than they gain in clock. The 31-stage Pentium 4 Prescott is the proof.
14. Superscalar, out-of-order execution, register renaming and the reorder buffer extract parallelism while still showing software a tidy, ordered machine.
15. Modern TAGE-style branch predictors are right the great majority of the time, and each miss still costs about 16 to 20 cycles.
16. SIMD does one instruction to many numbers at once, from MMX in 1997 to AVX-512 and Arm SVE. Fixed-function units such as AES-NI and NPUs go further and put whole algorithms in silicon.
17. Clocks stopped rising around 2005 when Dennard scaling ended, so chips went multicore, and Amdahl's law then set the ceiling.
18. Cache coherence keeps cores agreeing about one address; memory ordering and barriers govern the visible order across different addresses.
19. Caches work only because programs reuse data and touch neighbours. Sequential access can be 50 to 100 times faster than random access.
20. Judge a CPU by instruction count times cycles per instruction times clock period, measured on your own workload, not by the number on the box.

The Memory Hierarchy – Cache, RAM and Virtual Memory

11.0 What this chapter gives you

1. You will be able to draw the whole memory ladder, from registers down to tape, with real sizes, real waiting times and real prices.
2. You will be able to define temporal and spatial locality and show, with a measured number, why reading an array the wrong way is about nine times slower on the same machine.
3. You will be able to read a memory stick's label, say what DDR5-6000 CL30 means, and work out its true delay in nanoseconds by hand.
4. You will be able to trace a physical address into channel, rank, bank, row and column, and say why memory delay is a spread of values, not one value.
5. You will be able to translate a real 64-bit virtual address into a physical one, step by step, through four levels of page table.
6. You will be able to tell a minor page fault from a major one and from an invalid one, and give the cost of each in nanoseconds.
7. You will be able to read a real process memory map and name every region in it, including where a memory leak would show up.
8. You will be able to make three specific code changes that gave measured speed-ups of 9.0 times, 8.2 times and 3.5 times on the machine used to write this chapter.

11.1 Why a hierarchy exists at all

■ 11.1.1 PLAIN – in simple words

1. A computer needs to remember things. Some things it needs right now, some in a moment, some next year.
2. If we could buy one kind of memory that was huge, instant and cheap, there would be no chapter here. We cannot.
3. Memory that answers in under a nanosecond (a billionth of a second) is very expensive per byte and takes a lot of chip space.
4. So engineers build a ladder. Tiny and instant at the top. Huge and slow at the bottom.
5. Almost everything in this chapter is one idea repeated: guess what will be needed soon, and move it up the ladder before it is asked for.

■ 11.1.2 PLAIN – a picture in your head

1. Think of a student studying at a desk.
2. In your hand is one pen. That is a register. You can use it with no delay.
3. On the desk are three open books. That is cache. Reaching one takes a second.
4. On the shelf behind you are two hundred books. That is main memory. You must stand up and walk over.
5. In the university library are two million books. That is the disk. You must walk across campus and wait at a counter.
6. Nobody carries two million books to their desk. Nobody studies with an empty desk either. You keep the right small set close.

Where this comparison breaks: the student decides what to fetch. In a computer the hardware guesses, automatically, with no help from the program. Also, a book on the shelf is still one book. In a computer the same data can exist in four places at once, and keeping those copies agreeing is a real problem that Chapter 10 dealt with under the name cache coherence.

■ 11.1.3 PLAIN – a worked example

1. Take one real machine: the virtual server used to measure every benchmark in this chapter. It runs an Intel Xeon at 2.1 GHz.
2. One clock tick at 2.1 GHz lasts about 0.48 nanoseconds.
3. We measured how long one random memory read takes, for different amounts of data being walked over. Each read had to finish before the next one started, so no overlapping could hide the delay.

Data being walked	Measured time per read
16 KB (fits in L1)	1.79 ns
256 KB (fits in L2)	5.74 ns
8 MB (fits in L3)	37.65 ns
128 MB (goes to RAM)	143.37 ns
512 MB (goes to RAM)	179.76 ns

4. So the same instruction, `load` from memory, took anywhere from 1.79 to 179.76 nanoseconds. A factor of 100, decided entirely by where the data was. ### 11.1.4 PLAIN - what is really happening inside
5. Two different physical things are being traded off. Cost per bit, and time to answer.
6. Fast memory uses a circuit called SRAM, which needs six transistors to hold one bit. It holds its value as long as power is on, and answers in one or two clock ticks.
7. Main memory uses DRAM, which needs one transistor and one tiny capacitor per bit. Far smaller, far cheaper per bit.
8. But the DRAM capacitor leaks. Every cell must be read and rewritten thousands of times a second, which is called refresh, and reading it is destructive, so it must be written back.
9. Spinning disks and tape store magnetic patterns, and something physical has to move before you get an answer. Movement is measured in milliseconds. ### 11.1.5 TECHNICAL - the engineer's version
10. The full ladder, with figures accurate for a mid-2026 desktop or small server. Prices are retail per gigabyte and are unusually high because of the 2025 to 2026 memory shortage.

Level	Typical size	Latency
Register	32 x 64 bits	0 cycles
L1 data cache	32 to 64 KB per core	4 to 5 cycles
L2 cache	1 to 3 MB per core	14 to 20 cycles
L3 cache	24 to 260 MB shared	40 to 90 cycles
DRAM (DDR5)	16 to 512 GB	70 to 120 ns
NVMe SSD	0.5 to 8 TB	40 to 100 us
Hard disk	4 to 30 TB	5 to 10 ms
LTO-10 tape	30 TB per cartridge	30 to 90 s

2. The same ladder priced. Figures dated August 2026, taken from public price trackers. Treat them as approximate and expect them to move.

Level	Price per GB	Note
SRAM in cache	not sold separately	about 100x DRAM area
DDR5 DIMM	about 16.20 USD	was about 3 USD in 2024
DDR4 DIMM	about 7.64 USD	end-of-life squeeze
Consumer NVMe SSD	about 0.09 USD	1 TB near 90 USD
Nearline hard disk	0.014 to 0.030 USD	up about 50% in 2026
LTO-10 tape media	0.003 to 0.010 USD	drive costs thousands

3. The 2025 to 2026 price situation is worth stating plainly, because it is recent and it distorts every rule of thumb. TrendForce reported DDR5 spot prices rising 307 percent between September and November 2025, driven by AI datacentre demand. A 32 GB DDR5 kit that sold near 95 USD before the shortage was 380 to 589 USD in August 2026.
4. Established fact: the shortage happened and prices roughly doubled to quadrupled. Active forecast, not fact: several analysts expect no meaningful relief before late 2027. Treat that as a prediction.
5. The classic scaling rule, sometimes called the memory wall, is that CPU speed improved far faster than DRAM latency for about thirty years. DRAM bandwidth improved greatly. DRAM latency barely improved at all. Section 11.3 gives the numbers that prove it.
6. Tools that show you the real ladder on your own machine: `lscpu` and `getconf -a` on Linux, `sysctl -a | grep cache` on macOS, and the free Intel Memory Latency Checker for detailed per-level figures.

```

smaller, faster, dearer per byte
^
| registers    < 1 ns    bytes
| L1 cache    ~ 1 ns    tens of KB
| L2 cache    ~ 5 ns    a few MB
| L3 cache    ~ 40 ns   tens of MB
| DRAM        ~ 90 ns   tens of GB
| NVMe SSD    ~ 60 us   terabytes
| hard disk   ~ 8 ms    tens of TB
v tape        ~ 60 s    petabytes
bigger, slower, cheaper per byte

```

■ 11.1.6 WORDS – remember these

1. Memory hierarchy — the ladder of stores — a multi-level storage system trading capacity against access latency.
2. Latency — the waiting time before data arrives — time from request issue to first data return, measured in cycles or nanoseconds.
3. SRAM — fast memory that needs power — static random access memory, typically a six-transistor cell, no refresh needed.
4. DRAM — cheap main memory — dynamic random access memory, one transistor and one capacitor per cell, requires periodic refresh.
5. Memory wall — CPUs got fast, memory did not — the growing gap between processor cycle time and DRAM access latency.

11.2 Locality: why the whole idea works

■ 11.2.1 PLAIN – in simple words

1. The ladder only helps if we can guess what a program will want next.
2. Luckily, programs are extremely predictable in two specific ways.
3. First: if a program used something a moment ago, it will very likely use it again soon. That is called **temporal locality**, meaning locality in time.
4. Second: if a program used something, it will very likely use the thing sitting right next to it. That is **spatial locality**, meaning locality in space.
5. Because of these two habits, a small fast store holding recently used data and its neighbours will satisfy most requests.
6. Locality is not a law. It is a very strong statistical habit of real code. Code that breaks it runs slowly, and the machine cannot help you.

■ 11.2.2 PLAIN – a picture in your head

1. Picture a cook in a kitchen with a small counter next to the stove.
2. Temporal locality: the salt gets used every two minutes, so it stays on the counter all evening.
3. Spatial locality: when the cook fetches the tin of tomatoes from the store cupboard, they carry the whole tray it sits on, because the other tins on that tray will probably be needed too.
4. Carrying the whole tray is exactly what a computer does. It never fetches one byte from memory. It fetches a fixed-size block, normally 64 bytes, called a cache line.
5. If the cook only ever needs one tin from each of fifty different trays, carrying trays is pure waste, and the evening goes badly.

Where this comparison breaks: the cook knows the recipe in advance. The hardware does not know your program. It uses fixed rules, mainly “keep what was just used” and “fetch the neighbours”. Modern chips also add a prefetcher that spots simple patterns such as “every 64 bytes forward”, but it is a pattern detector, not a mind reader.

■ 11.2.3 PLAIN – a worked example

1. Here is the demonstration everybody should do once. A square grid of numbers, added up two different ways.
2. In C, a two-dimensional array is stored row by row. The whole first row sits in memory, then the whole second row, and so on. That is called row-major order and it is part of the C language definition.
3. Both read exactly the same 16,777,216 numbers and produce the same answer.

```
double a[4096][4096];          /* 128 MB of doubles */

/* version 1: along the rows, memory order */
for (i = 0; i < 4096; i++)
    for (j = 0; j < 4096; j++)
        s1 += a[i][j];

/* version 2: down the columns, jumping 32 KB each step */
for (j = 0; j < 4096; j++)
    for (i = 0; i < 4096; i++)
        s2 += a[i][j];
```

4. Measured on the 2.1 GHz Xeon used for this chapter, compiled with gcc -O2, three runs each.

Order	Time	Ratio
Along rows	0.0182 to 0.0209 s	1.0x
Down columns	0.1837 to 0.1889 s	9.0x to 10.4x

5. So the column version was about nine to ten times slower, on the same data, in the same program, with the same compiler flags.
6. Why. Each row is 4096 doubles, which is 32,768 bytes. Stepping down a column moves 32,768 bytes at a time.
7. A cache line is 64 bytes and holds 8 doubles. Walking a row uses all 8 of them before moving on, so one memory fetch serves 8 additions.
8. Walking a column uses 1 of the 8 and throws the rest away, so it needs 8 times as many fetches. It also touches a new 4 KB page every single step, which wrecks address translation as well. Section 11.6 explains that part. ### 11.2.4 PLAIN - what is really happening inside
9. When the CPU asks for an address, the cache checks whether the 64-byte block containing it is already held.
10. If it is there, that is a hit. If it is not, that is a miss, and the whole 64-byte block is fetched from the level below, and something already in the cache is thrown out to make room.
11. Row order: byte 0 misses, and the fetch brings bytes 0 to 63. The next seven reads all hit. One miss buys eight uses.
12. Column order: byte 0 misses and brings bytes 0 to 63, but the program's next read is 32,768 bytes away, so the other 63 bytes are never touched before being evicted. Every read is a miss.
13. On top of that, the hardware prefetcher watches the address stream. A forward walk with a small constant step is exactly the pattern it detects, so it starts fetching the next lines before they are asked for. Row order gets that free help. Column order, jumping 32 KB, usually does not. ### 11.2.5 TECHNICAL - the engineer's version
14. Denning formalized locality as the working set model in 1968, defining the working set $W(t, T)$ as the set of pages referenced in the last T units of virtual time. It remains the standard framework for replacement policy.
15. Effective access time with a two-level model: $T_{\text{eff}} = h * T_{\text{fast}} + (1 - h) * T_{\text{slow}}$, where h is the hit rate.
16. Put real figures in. With L1 at 1.79 ns, DRAM at 143 ns and a 95 percent hit rate: $0.95 * 1.79 + 0.05 * 143 = 8.85$ ns. At 99 percent it is 3.20 ns. At 99.9 percent it is 1.93 ns. The tail dominates, which is why hit rate is quoted to two decimal places.

17. Cache line size is 64 bytes on essentially all x86-64 and on Apple silicon the L1 line is 64 bytes with a 128-byte L2 line. Verify with `getconf LEVEL1_DCACHE_LINESIZE` on Linux or `sysctl hw.cachelinesize` on macOS.
18. Row-major storage for multidimensional arrays is a **standard**, fixed by the C and C++ language definitions. Column-major is the standard in Fortran, MATLAB, R and Julia. NumPy defaults to row-major but supports both through the `order` argument, so the same loop can be fast or slow depending on how the array was created.
19. The measured 9.0x to 10.4x ratio above is specific to this machine, this array size and this compiler. On a machine with a larger L3 and a smaller array the ratio shrinks toward 1. On a machine with slower DRAM it grows. Do not quote a single universal number.
20. Measure it rather than guess. On Linux:

```
perf stat -e cache-references,cache-misses,\
LLC-load-misses,dTLB-load-misses ./program
```

On macOS, use Instruments with the Counters template, or `xcrun xctrace`.

■ 11.2.6 WORDS – remember these

1. Temporal locality — used once, used again soon — the probability that a referenced address is referenced again within a short interval.
2. Spatial locality — neighbours get used together — the probability that addresses near a referenced address are referenced soon.
3. Cache line — the block that always moves together — the minimum unit of transfer between cache levels, 64 bytes on x86-64 and Arm.
4. Working set — what a program is using just now — the set of pages referenced in a sliding time window, from Denning 1968.
5. Prefetcher — the hardware guesser — a unit that detects address stride patterns and issues loads before the demand request.

11.3 RAM modules in the real world

■ 11.3.1 PLAIN – in simple words

1. Main memory comes on a green stick you push into a slot on the motherboard. The stick is called a **DIMM**, short for dual in-line memory module.
2. Laptops use a shorter version called a **SO-DIMM**, where SO means small outline.
3. A group of chips that answers as one unit is called a **rank**. A stick can have one rank or two, and a two-rank stick has chips on both sides, or stacked.
4. The motherboard has a small number of paths to memory, called **channels**. Two channels means two separate roads, not a wider road.
5. Filling both channels roughly doubles how much data per second you can move. It does not shorten the wait for a single item.
6. A label like DDR5-6000 CL30 tells you two things: how many transfers per second, and how many clock ticks you wait for the first one.

■ 11.3.2 PLAIN – a picture in your head

1. Think of a warehouse with numbered aisles, shelves and boxes.
2. A channel is a whole loading dock with its own truck. Two channels means two docks working at the same time.

3. A rank is one full team of workers. Two ranks means two teams that share the dock and take turns, so while one team is putting a pallet away, the other can be fetching.
4. A bank is an aisle. Only one shelf per aisle can be pulled out at a time, but different aisles can be busy at once.
5. A row is one shelf. When a worker pulls a shelf out, it sits on the trolley, and taking more boxes off the same shelf is quick.
6. Adding a second dock doubles throughput. It does not make one single box arrive any sooner. That is the difference between bandwidth and latency, and it is the single most misunderstood thing about RAM.

Where this comparison breaks: shelves in a real warehouse do not forget their contents. DRAM rows do. Reading a row destroys it and it must be written back, and every row must be refreshed every 32 or 64 milliseconds or the data fades away. There is no warehouse equivalent of that.

■ 11.3.3 PLAIN – a worked example

1. Take a common 2026 desktop kit: 2 sticks of 16 GB DDR5-6000, timings written as 30-36-36-76.
2. Those four numbers are, in order, CL, tRCD, tRP and tRAS, all counted in clock ticks.
3. DDR5-6000 means 6000 million transfers per second. DDR means double data rate: data moves on both the rise and the fall of the clock.
4. So the clock itself runs at 3000 MHz, which is 3000 million ticks a second.
5. CL30 means 30 ticks of waiting. $30 \times 0.3333 = 10.0$ nanoseconds.
6. The general formula, worth memorizing:

$$\text{true latency in ns} = \text{CL} * 2000 / \text{data rate in MT/s}$$

7. Now the shock. Apply the same formula across twenty-five years.

Module and timing	CAS in ns
PC133 SDRAM, CL3	22.5
DDR-400, CL3	15.0
DDR2-800, CL5	12.5
DDR3-1600, CL9	11.25
DDR4-3200, CL16	10.0
DDR5-6000, CL30	10.0
DDR5-8000, CL38	9.5

8. Transfer rate went up about 60 times. The wait for the first byte went from 22.5 ns to about 10 ns, roughly a factor of two, in twenty-five years.
9. Bandwidth for the kit: $6000 \text{ MT/s} \times 8 \text{ bytes per transfer} = 48 \text{ GB/s per stick}$. Two sticks in two channels = 96 GB/s.

■ 11.3.4 PLAIN – what is really happening inside

1. A DDR5 stick is 64 data wires wide, but DDR5 splits them into two independent halves of 32 wires each, called sub-channels. DDR4 did not do this.
2. A burst moves 16 transfers in a row on DDR5. On a 32-wire sub-channel that is $16 \times 4 \text{ bytes} = 64 \text{ bytes}$, which is exactly one cache line. The design was chosen to match.
3. Ranks matter because a rank cannot do two things at once. With two ranks the controller can be closing a row on one while reading from the other, which raises useful throughput by roughly 5 to 15 percent in practice.

4. **XMP** and **EXPO** are little profiles stored in a small chip on the stick. They tell the motherboard “you may run me at 6000 with these timings”, which is faster than the guaranteed baseline the standard defines.
5. **ECC** memory adds extra chips holding check bits. If one bit flips, from a cosmic ray or a marginal cell, the controller repairs it and carries on and logs it. If two bits flip, it detects the error and halts rather than returning wrong data.

■ 11.3.5 TECHNICAL – the engineer’s version

1. DDR generations, with JEDEC standard numbers and publication years.

Generation	Standard, year	Rates and voltage
DDR	JESD79, 2000	200–400 MT/s, 2.5 V
DDR2	JESD79-2, 2003	400–1066 MT/s, 1.8 V
DDR3	JESD79-3, 2007	800–2133 MT/s, 1.5 V
DDR4	JESD79-4, 2012	1600–3200 MT/s, 1.2 V
DDR5	JESD79-5, 2020	3200–6400 MT/s, 1.1 V
DDR5 rev C	JESD79-5C, 2024	up to 8800 MT/s

2. JESD79-5C was published on 17 April 2024. Besides extending timings to 8800 MT/s it added PRAC, per-row activation counting, which counts activations at wordline granularity so the system can react to rowhammer-style attacks. It also deprecated PASR, partial array self refresh, on security grounds.
3. DDR5 structural changes against DDR4: two 32-bit sub-channels instead of one 64-bit channel, 32 banks in 8 bank groups instead of 16 banks in 4 groups, 16n prefetch and burst length 16 instead of 8n and BL8, on-die ECC as standard, and the voltage regulator moved onto the module as a PMIC fed from 5 V.
4. On-die ECC is not the same as module ECC. On-die ECC protects the internal DRAM array only. It does not protect the bus between module and CPU. A non-ECC DDR5 stick has on-die ECC and is still not an ECC module.
5. Module ECC is SECDED: single error correct, double error detect. It widens the module from 64 to 72 data bits, 8 check bits per 64 data bits. Linux reports corrected and uncorrected counts through EDAC, readable under `/sys/devices/system/edac/mc/` or with `edac-util -v`.
6. XMP is an Intel **convention**, not a JEDEC standard. XMP 3.0 arrived with DDR5 in 2021 and provides five profile slots, three vendor-written and two user-writable. EXPO, AMD Extended Profiles for Overclocking, was announced in 2022 alongside Socket AM5 and Ryzen 7000. Both write into the module’s SPD, serial presence detect, chip. Running either is technically operating the part outside its JEDEC-guaranteed bin.
7. LPDDR is a separate family for phones and thin laptops, physically soldered rather than socketed, with lower voltage and aggressive power-down states. JEDEC published the first LPDDR6 standard on 10 July 2025, specifying 10,667 to 14,400 MT/s and a new arrangement of four 24-bit sub-channels, in place of DDR5’s two 32-bit sub-channels, for lower latency and more concurrency.
8. HBM, high bandwidth memory, stacks DRAM dies vertically and connects them with through-silicon vias to a very wide, short bus. JEDEC released JESD270-4, the HBM4 standard, on 16 April 2025: a 2048-bit interface, up to 8 Gb/s per pin, up to 2 TB/s per stack, 32 channels with 2 pseudo-channels each, 4-high to 16-high stacks, and up to 64 GB per stack using 32 Gb dies.
9. CXL, Compute Express Link, adds a third option: memory on the far side of a PCIe-style link. CXL 1.0 and 2.0 built on PCIe 5.0 in 2019 and 2020, CXL 3.0 on PCIe 6.0 in 2022, with 3.1 in 2023

and 3.2 in 2024. Type 3 devices are memory expanders. They add capacity and cost roughly 150 to 250 ns extra latency, so they sit between DRAM and SSD on the ladder.

10. Inspect real modules: `sudo dmidecode -t memory` on Linux, `sudo decode-dimms` for raw SPD contents, and `system_profiler SPMemoryDataType` on macOS.

■ 11.3.6 WORDS – remember these

1. DIMM — the memory stick — dual in-line memory module, a 64-bit-wide printed circuit board carrying DRAM devices.
2. Rank — one full team of chips — a set of DRAM devices sharing a chip-select and responding together to form the full data width.
3. Channel — one road to memory — an independent memory controller port with its own command and data bus.
4. CAS latency — ticks of waiting for the first byte — column address strobe delay in clock cycles, from column command to first data.
5. ECC — memory that repairs itself — error correcting code memory, normally SECDED using 8 check bits per 64 data bits.
6. HBM — memory stacked on the GPU package — high bandwidth memory, a 1024 or 2048-bit stacked DRAM connected by through-silicon vias.

11.4 The memory controller

■ 11.4.1 PLAIN – in simple words

1. Something has to turn “read address 4,297,318,416” into the right electrical commands on the right wires. That thing is the **memory controller**.
2. Today it is inside the processor itself, on the same piece of silicon. That change alone removed a large chunk of memory delay.
3. The controller takes an address and splits it into pieces: which channel, which rank, which bank, which row, which column.
4. It also keeps a queue of pending requests and is allowed to serve them out of order, choosing whichever one it can answer fastest.
5. So memory latency is not a number. It is a spread of numbers.

■ 11.4.2 PLAIN – a picture in your head

1. Back to the warehouse. A worker has one shelf pulled out onto the trolley at a time, per aisle.
2. If your box is on the shelf already out, you get it fast. That is a **row buffer hit**.
3. If the wrong shelf is out, they must push it back in before pulling yours. Slowest. That is a **row buffer conflict**.
4. The supervisor sees twenty orders at once and deliberately serves all the orders for the shelf already out, before switching. That is scheduling.
5. This is why the same order can take one, two or three times as long, depending purely on what came before it.

Where this comparison breaks: the supervisor cannot delay an order forever. Real controllers have fairness and starvation limits, and they must interrupt everything periodically for refresh, which has no warehouse equivalent. ### 11.4.3 PLAIN - a worked example

1. Take DDR5-6000 with timings 30-36-36-76. Convert to nanoseconds first.
2. CL 30 ticks = 10.0 ns. tRCD 36 ticks = 12.0 ns. tRP 36 ticks = 12.0 ns.

3. Now three cases for one read, at the DRAM chip itself.

Case	Sum of timings	Time
Row already open	CL	10.0 ns
No row open	tRCD + CL	22.0 ns
Wrong row open	tRP + tRCD + CL	34.0 ns

4. So the chip alone can answer in 10 ns or 34 ns for identical instructions.
5. Which is why the measured figure in section 11.1 was 143 ns and not 10 ns. Most of the delay is not in the DRAM chip at all. ### 11.4.4 PLAIN - what is really happening inside
6. The controller receives a physical address, for example 0x0001_62B8_D010, and chops the bits into fields. A rough example layout for a two-channel DDR5 system, from the top of the address downward: row bits, then bank group and bank bits, then rank, then channel, then column, then the byte offset inside the burst.
7. The exact chopping is an **implementation detail**, different on every CPU family, and often deliberately scrambled by XOR-ing bits together.
8. Why scrambled: if consecutive cache lines all landed in the same bank, a simple sequential walk would hit one bank over and over and get no parallelism. Interleaving spreads consecutive lines across channels and banks so they can be served at once.
9. Having chosen a bank, the controller issues ACTIVATE with a row address. That copies the whole row, typically 8 or 16 kilobits, into the sense amplifiers, which form the row buffer.
10. If it needs a different row in that bank, it must first issue PRECHARGE, which writes the buffer back into the cells and clears the sense amps.
11. The scheduler picks among queued requests. The classic policy is FR-FCFS, first-ready first-come-first-served: prefer any request that hits an open row, otherwise take the oldest.

■ 11.4.5 TECHNICAL - the engineer's version

1. History. AMD integrated the memory controller onto the CPU die with the Athlon 64, launched September 2003. Intel followed with Nehalem in November 2008, retiring the front-side bus. Before that, every memory access crossed a separate northbridge chip.
2. Consequence: memory latency became a property of the CPU, and multi-socket machines became NUMA, non-uniform memory access, where local DRAM is faster than DRAM attached to another socket. Typical remote penalty is 1.4x to 2.2x on latency. Inspect with `numactl --hardware` and `lstopo`.
3. Key DDR timing parameters, and what each one blocks.

Symbol	Meaning	DDR5-6000 30-36-36-76
tCL	column to data	10.0 ns
tRCD	activate to column	12.0 ns
tRP	precharge to activate	12.0 ns
tRAS	activate to precharge	25.3 ns
tREFI	refresh interval	3.9 us typical
tRFC	refresh duration	195 to 410 ns

4. Row buffer hit rate on real workloads is commonly 20 to 60 percent for general code, and much higher for streaming loops. It is a first-order term in memory performance, and it is why address interleaving schemes are tuned so carefully.

5. Latency is a distribution. On a loaded server, measured DRAM read latency commonly runs 75 ns at the median and 300 ns or worse at the 99th percentile, because queueing delay grows sharply as utilization approaches the bandwidth limit. This follows ordinary queueing theory: as utilization goes to 1, queue length goes to infinity.
6. Tools: Intel Memory Latency Checker prints a full loaded-latency curve. `perf stat -e un-core_imc/cas_count_read/` gives DRAM traffic on Intel. `pcm-memory` from Intel PCM shows per-channel bandwidth live. On AMD, use `amd_uprof`.
7. Rowhammer is the security consequence of all this. Repeatedly activating one row disturbs charge in neighbours, and since the 2014 Kim et al. paper it has been a real attack. Mitigations include target row refresh, and from JESD79-5C in 2024, PRAC.

■ 11.4.6 WORDS – remember these

1. Memory controller — the part that talks to the sticks — the integrated circuit block that translates addresses into DRAM commands and schedules them.
2. Row buffer — the shelf currently pulled out — the sense amplifier array holding one open DRAM row, typically 1 to 2 KB per device.
3. Row hit — asking for something already open — a column access to the currently activated row, costing only tCL.
4. Bank conflict — the wrong row is open — an access requiring precharge then activate before the column read, costing tRP + tRCD + tCL.
5. NUMA — some memory is further away — non-uniform memory access, where latency depends on which socket owns the DRAM.

11.5 The address space

■ 11.5.1 PLAIN – in simple words

1. An address is just a number that names a location. Nothing more.
2. If addresses are 64 bits, there are about 18.4 quintillion of them.
3. Every program believes it has the whole range to itself, starting near zero, with nobody else in it. That belief is a carefully maintained lie.
4. The addresses a program uses are **virtual addresses**. They are private invented numbers.
5. The addresses on the actual wires to the memory sticks are **physical addresses**. There is exactly one set of those in the machine.
6. This is why one program cannot read another's data by guessing. Its numbers simply do not refer to the same places.

■ 11.5.2 PLAIN – a picture in your head

1. Think of a large office building where every company has its own internal room numbering: Room 1, Room 2, Room 3.
2. Three companies all have a Room 1. They are three different physical rooms.
3. At every door there is a translator holding a small book: “for this company, Room 1 means B4-102”.
4. If a company asks for Room 9 and the book has no entry, the translator refuses to open anything and reports it.
5. Two companies can share a meeting room by both having book entries pointing at the same physical room. That is shared memory.

Where this comparison breaks: the translation happens billions of times a second, in hardware, in about a nanosecond when cached. And the book is not one book but a tree of books, which is section 11.6. ### 11.5.3 PLAIN - a worked example

1. Run two copies of the same program at the same time. Print the address of a variable in each.
2. Either way, neither number is a real place in the memory sticks.
3. Here are the real virtual addresses printed by a small test program on the Linux machine used for this chapter.

```
big heap block at 0x7f19b4bff010
small heap block at 0x55a23da282a0
```

4. The big block sits high, around 0x7f19..., because a large allocation is served by a fresh mapping in the middle of the address space.
5. The small block sits lower, around 0x55a2..., in the classic heap area just above the program’s own code. ### 11.5.4 PLAIN - what is really happening inside
6. Nothing in the CPU has 64 real address wires going to memory. That would be both useless and expensive.
7. Current x86-64 processors implement 48 bits of virtual address, extended to 57 bits on newer server parts. The rest of the bits are required to be copies of the top implemented bit.
8. An address obeying that rule is called **canonical**. A non-canonical address causes a fault immediately, before any translation is attempted.
9. So the usable space splits into two lumps: one at the very bottom starting at zero, and one at the very top ending at all ones. The vast middle is a hole.
10. The machine used for this chapter reports 52 bits of physical address and 57 bits of virtual address, which you can read with `lscpu`. ### 11.5.5 TECHNICAL - the engineer’s version
11. x86-64 canonical form is a **standard**, defined by the architecture. With 48-bit addressing, bits 48 to 63 must all equal bit 47. With LA57 enabled, bits 57 to 63 must all equal bit 56.
12. Address space sizes:

Mode	Virtual bits	Total space
x86-64 classic	48	256 TiB
x86-64 with LA57	57	128 PiB
Arm64 typical	48	256 TiB
Arm64 with LVA	52	4 PiB

3. Intel published the 5-level paging specification in 2016, submitting Linux patches on 8 December 2016. Support landed in Linux 4.14 and was enabled by default in Linux 5.5. Hardware support

arrived with Ice Lake server parts, and AMD supports it on EPYC 9004 and 8004 and Threadripper PRO 7000 WX. It is switched on by bit 12 of CR4, named LA57.

- Standard Linux x86-64 split with 4-level paging:

Region	Range
User space	0 to 0x00007fff_ffffffff
Non-canonical hole	0x00008000.. to 0xffff7ff..
Kernel space	0xffff8000_00000000 up

- That gives 128 TiB to user space and 128 TiB to the kernel. With LA57 both become 64 PiB. A process only gets the larger space if it explicitly asks, by passing an mmap hint above 47 bits, so that old programs storing tag bits in the top of pointers do not break. That compatibility hack is a real and current design decision, not history.
- Observe the layout of any Linux process with `cat /proc/PID/maps`. On macOS use `vmmap PID`. On Windows use VMMap from Sysinternals.

The honest version: “programs never see physical addresses” is a simplification. The kernel routinely works with them, drivers must hand real physical addresses to devices doing direct memory access, and many small embedded microcontrollers have no MMU at all, so every address is physical. The accurate statement is that user-space code on a general-purpose operating system with an MMU sees only virtual addresses.

■ 11.5.6 WORDS – remember these

- Address — a number naming a location — an index into an address space, 64 bits wide on modern general-purpose CPUs.
- Virtual address — the private number a program uses — the address produced by the program, subject to MMU translation.
- Physical address — the real number on the wires — the address presented to the memory controller after translation.
- Canonical address — an address with a legal top half — one whose unused high bits are sign-extended copies of the highest implemented bit.
- LA57 — the switch for 57-bit addressing — CR4 bit 12, enabling 5-level paging and a 128 PiB virtual space.

11.6 Virtual memory, explained slowly

■ 11.6.1 PLAIN – in simple words

- The machine cannot keep a note for every single byte saying where it really lives. That table would be bigger than the memory itself.
- So memory is cut into fixed-size blocks called **pages**. The usual size is 4 kilobytes, which is 4096 bytes.
- Now the note only has to say “virtual page 17 lives in physical frame 90210”. One entry covers 4096 bytes.
- The collection of those notes is the **page table**, and every process has its own.
- The hardware unit that does the lookup is the **memory management unit**, or MMU. It sits between the core and the caches.

6. Looking up a table in memory on every access would be hopeless, so the MMU keeps a small very fast cache of recent translations, called the **translation lookaside buffer**, or TLB. ### 11.6.2 PLAIN - a picture in your head
7. Imagine a huge hotel with a million rooms and a guest list.
8. A flat guest list with one line per room would be a book of a million lines, which you would have to carry everywhere.
9. Instead the hotel uses a tree. One thin card lists the buildings. Each building has a card listing floors. Each floor has a card listing corridors. Each corridor has a card listing rooms.
10. To find a guest you read four cards. But you only need cards for parts of the hotel that actually have guests.
11. The receptionist remembers the last few guests they looked up and answers those instantly without touching any card. That is the TLB.

Where this comparison breaks: the tree is walked by hardware, in silicon, not by anyone reading. And a missing card does not mean “no such guest”. It can mean “that guest is out, fetch them from storage”, which is a page fault. ### 11.6.3 PLAIN - a worked example

1. This is a real translation, measured on the Linux machine used for this chapter by reading `/proc/self/pagemap` as root.
2. The virtual address was `0x7f1e1a9ff010`. On x86-64 with 4-level paging, a 64-bit address is chopped like this:

```
bits 63..48 must copy bit 47 (canonical check)
bits 47..39 index into level 4 table (PML4)
bits 38..30 index into level 3 table (PDPT)
bits 29..21 index into level 2 table (PD)
bits 20..12 index into level 1 table (PT)
bits 11..0  byte offset inside the 4 KB page
```

3. Each index is 9 bits, so each table has $2^9 = 512$ entries. Each entry is 8 bytes, so each table is exactly $512 \times 8 = 4096$ bytes: one page. That is not a coincidence, it is the design.
4. Chopping `0x7f1e1a9ff010` gives:

Field	Value
PML4 index	254
PDPT index	120
PD index	212
PT index	511
Byte offset	16

5. The walk: read entry 254 of the level 4 table to get the address of a level 3 table. Read entry 120 of that to get a level 2 table. Read entry 212 of that to get a level 1 table. Read entry 511 of that to get the frame number.
6. Physical address = frame number $\times$ 4096 + offset = $0x162B8D000 + 0x10 = 0x162B8D010$.
7. Now look at the next two pages of the same allocation, which are next to each other in virtual space:

Virtual address	Frame number
0x7fle1a9ff010	0x162B8D
0x7fle1aa00010	0x1C18DD
0x7fle1aa01010	0x13BA83

8. Three pages that are neighbours in the program's view live in three scattered, unrelated places in the actual memory chips. ### 11.6.4 PLAIN - what is really happening inside
9. A register in the CPU holds the physical address of the top-level table. On x86-64 it is called CR3. On Arm64 it is TTBR0 and TTBR1.
10. Switching from one process to another means, at heart, writing a new value into that register. Everything the old process could reach becomes unreachable in one instruction.
11. On every memory access the MMU first asks the TLB. If the translation is there, done, no table reading at all.
12. If it is not, the hardware itself walks the tree. On x86-64 that is done by a hardware page walker, with no software involved and no interrupt.
13. Each entry it reads is 64 bits and carries more than an address. It carries flag bits: present, writable, user-accessible, accessed, dirty, and no-execute.
14. If the present bit is 0, the walker gives up and raises a **page fault**, which is an exception handled by the operating system. Section 11.7 covers what the operating system does next.
15. On success the translation is written into the TLB, and a few clock ticks later the data arrives. ### 11.6.5 TECHNICAL - the engineer's version
16. Page size 4096 bytes is a **convention** so widespread it feels like a standard, but it is not universal. It came from the VAX-11/780 in 1978 and was carried into x86 in 1985 with the 80386. Apple silicon uses 16 KB pages on macOS and iOS. Confirm with `getconf PAGE_SIZE`, which prints 4096 on x86-64 Linux and 16384 on an Apple silicon Mac.
17. Historical note worth knowing: the first machine with what we now call virtual memory was the Atlas, built by a joint University of Manchester and Ferranti team led by Tom Kilburn, and inaugurated in December 1962. They called it the **one-level store**. It had a 16,000 word core store and a 96,000 word drum, a block size of 512 words, and a replacement policy called the learning program that estimated reuse cycles per page and evicted the one predicted to be needed furthest in the future.
18. Page table levels on x86-64:

Levels	Name	Virtual bits
4	PML4, PDPT, PD, PT	48
5	PML5 added on top	57

4. Page table entry layout on x86-64, 64 bits wide:

Bit	Name	Meaning
0	P	present
1	R/W	writable
2	U/S	user accessible
5	A	accessed

Bit	Name	Meaning
6	D	dirty
7	PS	page size, huge page
51..12	PFN	frame number
63	NX	no execute

5. TLB structure on recent parts. AMD Zen 4 has a 72-entry fully associative L1 data TLB and a 3072-entry L2 data TLB. Intel Golden Cove has a 96-entry L1 data TLB and a 2048-entry shared L2 TLB. Figures for Zen 5 were not fully published at the time of writing, so treat any specific number for it as uncertain.
6. TLB reach, meaning how much memory the TLB can cover at once, is the number that matters:

TLB and page size	Reach
64 entries x 4 KB	256 KB
2048 entries x 4 KB	8 MB
2048 entries x 2 MB	4 GB
2048 entries x 1 GB	2 TB

7. Measure TLB behaviour with `perf stat -e dTLB-load-misses,dtlb_load_misses.walk_active ./prog` on Linux. Correlate with `perf stat -e page-faults`.
8. Every virtual machine adds a second layer. The guest's page tables map guest virtual to guest physical, and a second set, called extended page tables on Intel and nested page tables on AMD, maps guest physical to host physical. A full nested walk can require up to 24 memory accesses on a 4-level by 4-level system, which is why the measured latencies in this chapter, taken inside a virtual machine, are higher than bare metal.

```

virtual address 0x7f1e1a9ff010
|
| CR3 -> level 4 table
+--> [254] --> level 3 table
      +--> [120] --> level 2 table
            +--> [212] --> level 1 table
                  +--> [511] --> frame 0x162B8D
                                     |
physical address = 0x162B8D000 + 0x010 -----+
                  = 0x162B8D010

```

■ 11.6.6 WORDS – remember these

1. Page — a fixed block of a program's memory — the unit of virtual to physical mapping, normally 4 KB, or 16 KB on Apple silicon.
2. Frame — a fixed block of real memory — a physical page-sized region of DRAM, identified by a page frame number.
3. Page table — the map from pages to frames — a radix tree of 512-entry tables walked by hardware.
4. MMU — the translator — memory management unit, the hardware performing address translation and permission checks.
5. TLB — the translator's memory of recent answers — translation lookaside buffer, a small associative cache of page table entries.
6. CR3 — where the map starts — the x86-64 control register holding the physical address of the top-level page table; TTBR0 and TTBR1 on Arm64. ## 11.7 Paging and swapping

■ 11.7.1 PLAIN – in simple words

1. A **page fault** is the hardware saying “I cannot translate this address, you deal with it”. Control jumps into the operating system.
2. That is not an error. It is the normal way memory gets allocated, and there are three kinds, which you must be able to tell apart.
3. A **minor fault** means the data is already in RAM somewhere, and the kernel only has to write a page table entry. Fast: microseconds.
4. A **major fault** means the data must be read from disk first. Slow: tens to thousands of microseconds.
5. An **invalid fault** means the address is genuinely not yours. The program is killed, with a message like segmentation fault.
6. When RAM runs short, the kernel writes some pages out to disk and takes the RAM back. That is **swapping**.
7. If it has to keep bringing them straight back, the machine spends all its time moving pages and none doing work. That is **thrashing**, and it is why a full machine feels frozen rather than merely slow.

■ 11.7.2 PLAIN – a picture in your head

1. Picture a small desk and a big filing cabinet behind you.
2. A document that is in the room but under a pile: you find it in a second. That is a minor fault.
3. A document in the cabinet: you stand, walk, open a drawer, search. That is a major fault, and it takes a thousand times longer.
4. Now imagine the desk holds four documents but the task needs five, and you cycle through all five over and over.
5. Every single step needs a trip to the cabinet, and you also have to file something away first to make room. You get nothing done.
6. Adding a bigger desk fixes it completely. Adding a faster cabinet barely helps.

Where this comparison breaks: the desk in a computer is also full of documents nobody asked for, kept just in case, which the kernel will silently throw away the instant they are needed. That is the page cache, and it is why “free memory” is such a misleading measurement.

■ 11.7.3 PLAIN – a worked example

1. Here are real numbers measured for this chapter, on the same 2.1 GHz Linux virtual machine.
2. A program mapped 512 MB of anonymous memory and wrote one byte in each 4 KB page, then wrote one byte in each page again.

Pass	Total	Per page
First touch, faults	0.220 to 0.271 s	1677 to 2071 ns
Second touch, no faults	0.002 to 0.003 s	18 to 20 ns

3. So a minor page fault cost about 100 times as much as a plain memory write.
4. $131,155 - 83 = 131,072$. And $512 \text{ MB} / 4 \text{ KB} = 131,072$. One fault per page, exactly.
5. Second experiment, showing that a promise is not memory:

```

at start          VmSize =   2.6 MB   VmRSS =   1.4 MB
after malloc 4 GB VmSize = 4098.6 MB  VmRSS =   1.7 MB
after touching 1 GB VmSize = 4098.6 MB VmRSS = 1025.7 MB
after touching 4 GB VmSize = 4098.6 MB VmRSS = 4097.7 MB

```

6. Asking for 4 GB moved resident memory by 0.3 MB. The memory appeared only when it was written to.
7. Now put a cost on a major fault. A minor fault is about 2 microseconds. A read from an NVMe SSD is about 60 microseconds. A read from a hard disk is about 8 milliseconds.
8. So one major fault to SSD costs roughly 30 minor faults. One major fault to a hard disk costs roughly 4,000 minor faults, or about 90,000 ordinary memory writes. ### 11.7.4 PLAIN - what is really happening inside
9. When you call `malloc` for a large block, the C library asks the kernel for a mapping. The kernel records “this range of addresses belongs to you” in a list of regions, and stops there.
10. Your first write to that region raises a page fault. The kernel finds a free physical frame, fills it with zeros for safety, writes the page table entry, and returns to the exact instruction that faulted, which now succeeds.
11. Reading a file works the same way. The kernel keeps recently used file contents in RAM in the **page cache**. If your fault can be satisfied from the page cache, it is minor. If not, real disk input starts and it is major.
12. When free memory gets low, a kernel thread scans and reclaims. It has two easy sources: clean file pages, which can simply be dropped because the copy on disk is identical, and clean anonymous pages, which are rare.
13. Dirty pages must be written first. File pages go back to their file. Anonymous pages have no file, so they go to **swap**: a dedicated partition or a file, called the page file on Windows.
14. Before writing anything out, modern systems try compressing instead. Squeezing three pages into one costs microseconds of CPU and saves milliseconds of disk. ### 11.7.5 TECHNICAL - the engineer’s version
15. Fault taxonomy in the terms the tools use:

Kind	Cause	Typical cost
Minor	anon first touch, COW, cache hit	1 to 3 us
Major	needs disk or swap read	50 us to 10 ms
Invalid	no mapping or bad access	SIGSEGV

2. Copy-on-write faults are minor but not cheap. Measured on this machine, a process with 1 GB resident forked, and the child then wrote every page: about 16 microseconds per page, against about 2 microseconds for a plain first-touch fault. That gap is larger than on bare metal because nested paging in a virtual machine makes TLB invalidation expensive. Treat 16 microseconds as this machine’s figure, not a universal one.
3. Overcommit is a **policy**, not a law. Linux exposes it through `/proc/sys/vm/overcommit_memory`: 0 is heuristic and the default, 1 always allows, 2 enforces a strict limit set by `overcommit_ratio`. With the default, `malloc` of more memory than you have usually succeeds.
4. Memory compression, with dates:

System	Feature	Since
macOS	Compressed Memory	10.9, 2013
Windows	Memory Compression	Windows 10, 2015
Linux	zswap	kernel 3.11, 2013
Linux	zram	mainline 3.14, 2014

5. macOS compressed memory was introduced in OS X 10.9 Mavericks, announced June 2013 and released October 2013, using the WKdm family of fast dictionary compressors. Typical ratio on real workloads is around 2 to 1. zswap and zram commonly use LZO or LZ4 and report similar ratios.
6. Reading memory pressure on macOS. Activity Monitor's Memory tab shows a Memory Pressure graph, which is driven by the kernel's own pressure state, not by free memory. Green means fine. Yellow means reclaim is working hard. Red means applications are about to be terminated. On the command line:

```
vm_stat 1          # page-level counters, 4096-byte units
memory_pressure    # prints the current pressure level
top -l 1 -s 0 -n 0 # PhysMem and compressor lines
footprint -p PID   # per-process accounting, macOS 11+
```

7. Reading memory pressure on Linux:

```
free -m           # the -/+ buffers view, and available
vmstat 1          # watch si and so: swap in and out per s
cat /proc/meminfo  # everything, in kB
cat /proc/pressure/memory # PSI: some/full stall percentages
sar -B 1          # pgpgin, pgpgout, majflt per second
```

8. On `free -m`, the column to trust is **available**, not free. On the machine used here: total 8023 MB, used 773 MB, free 6573 MB, buff/cache 910 MB, available 7249 MB. Available exceeds free because most of buff/cache can be reclaimed instantly.
9. Pressure Stall Information, added in Linux 4.20 in December 2018, is the best single signal. `/proc/pressure/memory` gives the percentage of time tasks were stalled waiting on memory. A full average above a few percent means real trouble, long before free memory looks alarming. ### 11.7.6 WORDS - remember these
10. Page fault — the hardware asking the kernel for help — an exception raised when translation or permission check fails.
11. Minor fault — fixable without disk — a fault resolved by mapping an existing or newly zeroed frame.
12. Major fault — needs disk input — a fault requiring a read from a file or from swap before it can be satisfied.
13. Demand paging — nothing until you touch it — allocating physical frames lazily on first access rather than at request time.
14. Page cache — file contents kept in spare RAM — the kernel cache of file pages, reclaimable on demand.
15. Swap — RAM contents parked on disk — backing store for anonymous pages; called the page file on Windows.
16. Thrashing — all paging, no progress — a state where the working set exceeds RAM and nearly every access faults. ## 11.8 Huge pages, memory protection and isolation

■ 11.8.1 PLAIN - in simple words

1. A 4 KB page is small. A program using 64 gigabytes needs 16 million translations.
2. The translator's fast memory, the TLB, holds only a couple of thousand. So big programs miss constantly.
3. The fix is bigger pages. On a normal PC you can have 2 megabyte pages and 1 gigabyte pages.
4. One 2 MB page replaces 512 small ones, so the same TLB now covers 512 times as much memory.

5. Separately, every page carries permission flags: may be read, may be written, may be executed as code.
6. A sane rule is that a page is never both writable and executable at once. That is called **W xor X**, written W^X .
7. Programs are also loaded at a different random address every run, so an attacker cannot know where anything is. That is **address space layout randomization**. ### 11.8.2 PLAIN - a picture in your head
8. Think of a warehouse index card system again. Small pages are like indexing every single box.
9. Huge pages are like indexing whole pallets. Far fewer cards, so the whole index fits in your pocket.
10. Permissions are like signs on a room: READ ONLY, NO ENTRY, STAFF ONLY.
11. W^X is the rule “a room may be a workshop or a library, never both”. An attacker who sneaks a fake instruction into a data room finds it cannot be executed there.
12. Randomization is renumbering all the rooms every morning. A burglar with yesterday’s map is lost.

Where this comparison breaks: the signs are checked by hardware on every single access, in parallel with the lookup, so they cost nothing. No human system can check every action for free. ### 11.8.3 PLAIN - a worked example

1. Measured on this machine: a random pointer chase over 512 MB, once with ordinary 4 KB pages and once with transparent huge pages of 2 MB.

Pages used	Time per access
4 KB pages	170.2 to 180.8 ns
2 MB huge pages	136.4 to 139.5 ns

2. The speed-up was 1.25 to 1.30 times. Not enormous, but free, on a workload the CPU was already handling as badly as possible.
3. The arithmetic behind it. A 2048-entry TLB with 4 KB pages covers $2048 \times 4 \text{ KB} = 8 \text{ MB}$. With 2 MB pages it covers $2048 \times 2 \text{ MB} = 4 \text{ GB}$.
4. Now the permission side. Read a real memory map. Note the last column of letters:

```
55a209398000-55a209399000 r--p /tmp/mb/maps2
55a209399000-55a20939a000 r-xp /tmp/mb/maps2
55a20939a000-55a20939b000 r--p /tmp/mb/maps2
55a20939c000-55a20939d000 rw-p /tmp/mb/maps2
55a23da28000-55a23da49000 rw-p [heap]
7f19b8c28000-7f19b8db0000 r-xp libc.so.6
7ffd3da2b000-7ffd3da4d000 rw-p [stack]
```

5. Every single region is either r-x or rw-. Not one is rwx. That is W^X being enforced, visible in one command.

■ 11.8.4 PLAIN - what is really happening inside

1. A huge page is not a special kind of memory. It is the page walk stopping early.
2. Normally the level 2 entry points at a level 1 table. If its page-size flag is set, the walker treats it as pointing directly at a 2 MB region and stops. One less level, one less table, one TLB entry for 2 MB.

- There are two ways to get them. Reserved pools, where you set aside a fixed number at boot and programs must ask explicitly. And transparent huge pages, where the kernel promotes ordinary allocations automatically and quietly.
- The no-execute flag is bit 63 of a page table entry. When set, any attempt to fetch an instruction from that page faults.
- Isolation needs no extra machinery. Process A's page table has entries for A's frames. Process B's has entries for B's. There is no address A can name that reaches B's memory, because a name is only meaningful through A's own table. ### 11.8.5 TECHNICAL - the engineer's version
- Page sizes by architecture:

Architecture	Base page	Large pages
x86-64	4 KB	2 MB, 1 GB
Arm64, 4 KB granule	4 KB	2 MB, 1 GB
Arm64, 16 KB granule	16 KB	32 MB
Arm64, 64 KB granule	64 KB	512 MB

- Apple silicon Macs and iOS devices use the 16 KB granule. `getconf PAGE_SIZE` returns 16384 there and 4096 on x86-64 Linux. Android has been moving apps and libraries to 16 KB page alignment since 2024 for the same reason. Support for 1 GB pages on x86-64 is reported by the `pdpe1gb` CPU flag in `/proc/cpuinfo`.
- Two mechanisms on Linux, and they behave very differently:

```
# transparent huge pages: automatic, best effort
cat /sys/kernel/mm/transparent_hugepage/enabled
# prints e.g. always [madvise] never
madvise(p, len, MADV_HUGEPAGE); # opt in from code

# hugetlbfs: reserved, guaranteed, never swapped
echo 512 > /proc/sys/vm/nr_hugepages
grep -i huge /proc/meminfo
mmap(..., MAP_HUGETLB | MAP_HUGE_2MB, ...);
```

- Where huge pages help most: large in-memory databases, JVM heaps, HPC working sets, and virtual machine guest memory, where they cut nested page walk cost sharply. Where they hurt: memory-tight desktops, workloads with many small sparse mappings, and latency-sensitive services that suffer from the kernel's page-compaction pauses. Oracle, PostgreSQL and Redis documentation all recommend disabling transparent huge pages while using explicit hugetlb pages instead. That disagreement between "automatic is good" and "automatic causes latency spikes" is real and unresolved.
- Protection history, with dates:

Feature	First shipped	Year
NX bit	AMD64, Athlon 64	2003
W^X default	OpenBSD 3.3	2003
DEP	Windows XP SP2	2004
ASLR design	PaX project	2001
ASLR default	OpenBSD 3.4	2003

6. Fuller ASLR timeline: PaX published the first design and implementation in July 2001. OpenBSD 3.4 in 2003 was the first mainstream operating system with it on by default. Linux enabled a weak form by default from kernel 2.6.12 in June 2005. Windows Vista added it for opted-in binaries, RTM November 2006. macOS randomized system libraries in 10.5 Leopard in October 2007, extended it to all applications in 10.7 Lion in July 2011, and added kernel randomization in 10.8 Mountain Lion in July 2012.
7. W^X is not free in practice. Just-in-time compilers must generate code and then run it. The modern answer is dual mapping: map the same physical pages twice, once writable and once executable, and never both at the same address. Apple silicon requires this and provides `pthread_jit_write_protect_np` to flip a thread between write mode and execute mode.
8. Meltdown, disclosed in January 2018, broke the assumption that the user-accessible flag alone was enough, because speculative execution leaked kernel data through cache timing. The fix, kernel page table isolation, gives the kernel a separate set of page tables so kernel memory is not mapped at all while user code runs. It costs 5 to 30 percent on syscall-heavy workloads, which is why PCID, process context identifiers, matter: they let the CPU keep TLB entries across the switch instead of flushing. ### 11.8.6 WORDS - remember these
9. Huge page — one big page instead of many small — a mapping created by terminating the page walk early, 2 MB or 1 GB on x86-64.
10. TLB reach — how much memory the translator covers — TLB entry count multiplied by page size.
11. THP — automatic huge pages — transparent huge pages, kernel-managed promotion of 4 KB mappings to 2 MB.
12. NX bit — this page is data, not code — bit 63 of a page table entry marking the region non-executable.
13. W^X — never writable and executable at once — a policy that no mapping carries both `PROT_WRITE` and `PROT_EXEC`.
14. ASLR — everything moves every run — address space layout randomization, the randomizing of load addresses to defeat fixed-address exploits. ## 11.9 How a program's memory is laid out

■ 11.9.1 PLAIN – in simple words

1. A running program's address space is not one blob. It is a set of regions, each with a job and a permission.
2. **Text** is the machine code. Readable and executable, never writable.
3. **Data** holds variables that start with a value, for example a counter set to 5. It is copied from the file at load time.
4. **BSS** holds variables that start at zero. It takes no space in the file at all, only a note saying "reserve this many zero bytes".
5. **Heap** is memory you ask for while running, and give back when done.
6. **Stack** holds local variables and the trail of function calls. It grows and shrinks automatically as functions are entered and left.
7. **Mapped regions** are everything else: shared libraries, big allocations, files made to look like memory.

■ 11.9.2 PLAIN – a picture in your head

1. Picture a long street. At the low-numbered end is a printing works that cannot be altered: the code.
2. Then a builder's yard that grows outward as you request more land: the heap.
3. At the high-numbered end is a stack of trays. Each function call puts a new tray on top; returning takes it off. It grows toward the yard.
4. If the trays pile up so high they reach the yard, something has gone badly wrong. That is a stack overflow.

Where this comparison breaks: modern systems leave an enormous unused gap between the stack and the heap, and put a deliberately unmapped guard page just below the stack, so the two cannot silently collide. The crash is arranged in advance rather than allowed to happen.

■ 11.9.3 PLAIN – a worked example

1. Here is the real map of a tiny C program on the Linux machine used for this chapter, printed by reading `/proc/self/maps`. Columns are range, permissions and what it is.

```
55a209398000-55a209399000 r--p maps2 read-only data
55a209399000-55a20939a000 r-xp maps2 the code (text)
55a20939a000-55a20939b000 r--p maps2 constants
55a20939c000-55a20939d000 rw-p maps2 data and BSS
55a23da28000-55a23da49000 rw-p [heap] small allocations
7f19b4bff000-7f19b8c00000 rw-p (anon) the 64 MB malloc
7f19b8c00000-7f19b8c28000 r--p libc.so.6 library headers
7f19b8c28000-7f19b8db0000 r-xp libc.so.6 library code
7f19b8e03000-7f19b8e05000 rw-p libc.so.6 library variables
7f19b8ef7000-7f19b8ef9000 r-xp [vdso] kernel fast calls
7ffd3da2b000-7ffd3da4d000 rw-p [stack] the stack
```

2. Read the permissions. Code is r-x. Data is rw-. Nothing is rwx.
3. The heap region is only 132 KB. The 64 MB request did not go there. It got its own mapping at `0x7f19b4bff000`, because glibc sends large requests straight to the kernel instead of extending the heap.
4. The stack is `0x7ffd3da2b000` to `0x7ffd3da4d000`, which is 136 KB right now. It can grow down to the limit, which `ulimit -s` reported as 8192 KB. ### 11.9.4 PLAIN - what is really happening inside
5. `malloc` is a library function, not a system call. Most of the time it never talks to the kernel.
6. It keeps a pool of memory it already owns and hands out pieces from it, remembering which pieces are free in linked lists called **free lists**.
7. `free` does not return memory to the operating system. It puts the piece back on a free list, and often merges it with neighbouring free pieces.
8. That is why a program's memory use rarely goes down after a big job finishes. The memory is free to the program, still owned from the kernel's point of view.
9. Repeated allocate-and-free of different sizes leaves the pool full of holes. Plenty of free bytes, none of them big enough. That is **fragmentation**.
10. A **use-after-free** is keeping a pointer to something you freed. The allocator may have handed those bytes to someone else, so you now read or write their data. It is one of the most common serious security bugs.

- Garbage collection removes the need to call free. The runtime finds objects nothing points at any more and reclaims them automatically.

■ 11.9.5 TECHNICAL – the engineer’s version

- Segment names as the linker uses them. In ELF: `.text`, `.rodata`, `.data`, `.bss`. Inspect with `size ./prog` and `readelf -S ./prog` on Linux, or `size -m` and `otool -l` on macOS.
- `glibc malloc` is `ptmalloc2`, derived from Doug Lea’s `dlmalloc`. Key behaviours, all **implementation details** and all tunable:

Mechanism	Default
mmap threshold	128 KB, grows to 32 MB
Arenas per process	up to 8 x cores
Per-thread cache	tcache, glibc 2.26, 2017
Bin classes	fast, tcache, small, large

- Arenas exist to reduce lock contention: each thread is steered to its own arena so allocations do not serialize. Set `MALLOC_ARENA_MAX` to limit it, which is a common fix for containers showing surprising memory growth.
- Alternative allocators, with origins: `tcmalloc` from Google in 2005, `jemalloc` written by Jason Evans for FreeBSD in 2006 and later used by Firefox and Facebook, and `mimalloc` from Microsoft Research in 2019. Swapping allocator by `LD_PRELOAD` alone often changes throughput by 5 to 30 percent on allocation-heavy servers.
- Slab allocation, published by Jeff Bonwick for SunOS in a 1994 USENIX paper, caches whole pre-constructed objects of one type. Linux uses SLUB today. Inspect with `slabtop` or `cat /proc/slabinfo`. Physical pages themselves are handed out by a buddy allocator, visible in `/proc/buddyinfo`.
- Fragmentation has two forms. **Internal** is the unused tail inside a block rounded up to a size class. **External** is free memory split into pieces too small to use. Size-class allocators trade a little internal fragmentation to nearly eliminate external fragmentation.
- Detecting the classic bugs:

```
valgrind --leak-check=full ./prog          # leaks, invalid access
gcc -fsanitize=address,undefined ./p.c    # fast, needs rebuild
heaptrack ./prog                          # allocation profiles
leaks PID                                # macOS built in
MallocStackLogging=1 ./prog               # macOS allocation traces
```

- The honest version: “free never returns memory to the operating system” is not quite true. `glibc` will trim the top of the heap with `brk` when a large contiguous run at the end becomes free, and blocks that were served by `mmap` are unmapped on `free`. What is true is that blocks freed from the middle of the heap stay owned by the process.
- Garbage collection changes the failure modes rather than removing them. Leaks become “still reachable but never used”, which no collector can fix. Use-after-free largely disappears. In exchange you get pauses, extra memory headroom, and worse locality unless the collector compacts. Java’s G1 became default in JDK 9 in 2017 and ZGC reached production readiness in JDK 15 in 2020, targeting sub-millisecond pauses; Go has run a concurrent collector with sub-millisecond pauses since Go 1.8 in 2017. Rust takes the third path: no collector, ownership checked at compile time.

■ 11.9.6 WORDS – remember these

1. Text segment — the code — the read-only executable region loaded from the binary, mapped r-x.
 2. BSS — the zero-filled variables — uninitialized static storage, reserved by the loader and absent from the file.
 3. Heap — memory you ask for at run time — the region managed by the allocator through `brk` and `mmap`.
 4. Stack — the trail of function calls — a downward-growing region holding frames, guarded by an unmapped page.
 5. Free list — the allocator's list of spare pieces — linked structures of reclaimed blocks, usually bucketed by size class.
 6. Fragmentation — free but unusable — memory split so that no single free region satisfies a request.
- ## 11.10 Shared memory and mapped files

■ 11.10.1 PLAIN – in simple words

1. Two page tables can point at the same physical frame. That is all shared memory is.
 2. A file can also be attached to a range of addresses. Reading those addresses reads the file. That is a **memory-mapped file**.
 3. **Copy-on-write** is the clever middle case. Two processes share a page and both see it, but the page is marked read-only.
 4. The moment either one writes, the hardware faults, the kernel makes a private copy for the writer, and the two go their separate ways.
 5. That is why starting a copy of a process is cheap: nothing is copied until something is changed.
- ### 11.10.2 PLAIN - a picture in your head
6. Think of a reference book in a shared office.
 7. Everyone's desk index says "the dictionary is in cabinet 4". One book, many pointers to it. That is a shared library.
 8. Now a rule: you may read it, but if you want to scribble in it, you must first photocopy the page you want to change and scribble on your copy.
 9. A memory-mapped file is the librarian agreeing to place any page of any book onto your desk the instant you look for it, without you filling in a request slip.

Where this comparison breaks: photocopying is per page, not per book, and the granularity is exactly 4 KB. Changing one byte copies 4096.

■ 11.10.3 PLAIN – a worked example

1. Measured for this chapter. A process filled 1 GB of memory, then forked.

```
parent RSS after filling 1 GB : 1050060 KB
child: fork returned after      15.07 ms
child RSS immediately after fork: 1049404 KB
child: after writing all 1 GB    : copy cost 4202 ms
                                = 16.0 us per 4 KB page
```

2. Forking a 1 GB process took 15 milliseconds, not the seconds a real 1 GB copy would need. Almost nothing was copied.
3. The child's reported resident size was immediately about 1 GB, which looks like a copy but is not. Resident size counts shared pages in full, for both processes. This is the single biggest reason memory numbers confuse people, and section 11.11 deals with it.

- Second measurement: the sharing of library code. `/proc/self/smaps_rollup` for a small program reported:

Counter	Value
Rss	1640 kB
Pss	397 kB
Shared_Clean	1496 kB
Private_Dirty	108 kB

- Resident size 1640 kB, but proportional size only 397 kB, because 1496 kB of it is shared library code counted once per sharer. The program's genuinely private, modified memory is 108 kB.

■ 11.10.4 PLAIN – what is really happening inside

- `mmap` asks the kernel to attach a range of addresses to something: a file, or nothing at all, which is called an anonymous mapping.
- Two flags decide the sharing behaviour. `MAP_SHARED` means writes go to the underlying file and are visible to everyone. `MAP_PRIVATE` means writes become your own copy-on-write copies and are never written back.
- No pages are brought in at `mmap` time. The mapping is a note. Pages arrive on first touch, exactly as in section 11.7.
- On fork the kernel copies the page tables, not the pages, and marks every writable private page read-only in both parent and child.
- Any write now faults. The kernel checks that the page is a copy-on-write page, allocates a fresh frame, copies 4096 bytes, points the writer's entry at the copy and makes it writable again.
- For a mapped file, a write marks the page dirty. A kernel writeback thread sends dirty pages to the file later, or at `msync`, or at `unmap`. ### 11.10.5 TECHNICAL - the engineer's version
- The core call, with the flags that matter:

```
void *p = mmap(NULL, len,
               PROT_READ | PROT_WRITE,
               MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

/* file made to look like memory */
int fd = open("big.dat", O_RDWR);
void *f = mmap(NULL, len, PROT_READ | PROT_WRITE,
               MAP_SHARED, fd, 0);
msync(f, len, MS_SYNC); /* force it to disk */
madvice(f, len, MADV_SEQUENTIAL | MADV_WILLNEED);
```

- `mmap` appeared in the Berkeley and Sun Unix line in the 1980s and is now POSIX. The Windows equivalents are `CreateFileMapping` and `MapViewOfFile`.
- When mapped files win: random access to a large file, many processes reading the same file, and avoiding the copy from page cache into a user buffer. When they lose: sequential streaming, where read with a large buffer is usually faster and has predictable memory behaviour; files that may be truncated underneath you, which turns into `SIGBUS`; and network filesystems, where a page fault can block on the network with no error path.
- Databases disagree publicly about this. LMDB and older MongoDB storage engines were built on `mmap`. The 2022 paper "Are You Sure You Want to Use MMAP in Your Database Management System" from CMU argues against it, citing loss of control over eviction, transactional safety and

page fault costs. PostgreSQL and InnoDB manage their own buffer pools instead. This is a genuine, live disagreement between competent engineers.

5. Copy-on-write is why fork is cheap and why fork in a large process is still not free: page table copying is proportional to the number of mapped pages, and every subsequent write costs a fault. Measured here: 15 ms to fork 1 GB, then 16 microseconds per page written.
6. This has a famous consequence for garbage-collected and reference-counted runtimes. CPython updates a reference count on every object touched, which writes to the object header, which triggers copy-on-write. A forked CPython worker therefore un-shares memory quickly. CPython gained `gc.freeze` in 3.7, released 2018, partly to reduce this. ### 11.10.6 WORDS - remember these
7. `mmap` — attach memory to a thing — the system call mapping a file or anonymous memory into an address space.
8. Anonymous mapping — memory backed by nothing — a mapping with no file, zero filled on first touch, backed by swap if needed.
9. Copy-on-write — share until someone writes — a policy of mapping pages read-only and duplicating them on the first write fault.
10. Memory-mapped file — a file that looks like an array — file contents made accessible through ordinary loads and stores via the page cache.
11. PSS — your fair share of shared memory — proportional set size, private pages plus each shared page divided by its number of sharers.

11.11 Measuring memory for real

■ 11.11.1 PLAIN - in simple words

1. Ask “how much memory is this program using” and there is no single true answer. There are about six, and they all mean different things.
2. **Virtual size** is how much address space the program has claimed. It can be enormous and mean nothing.
3. **Resident size** is how much is actually in RAM right now. Closer to useful, but it counts shared things in full for every sharer.
4. **Private** is the part only this process has. This is the number that would actually be freed if you killed it.
5. Add up resident sizes for all processes and you will get far more than the machine has, because shared pages are counted many times. ### 11.11.2 PLAIN - a picture in your head
6. Five flatmates share one kitchen, and each has a bedroom.
7. Ask each “how much of the flat do you use” and each says “my bedroom plus the kitchen”. Add the answers and you get more than the flat.
8. Virtual size is like counting the whole building because you have a key to the front door.
9. Proportional is bedroom plus one fifth of the kitchen. The five proportional answers do add up to the flat exactly.
10. Private is just the bedroom: what is actually emptied when someone leaves.

Where this comparison breaks: the kitchen can vanish and reappear. Clean file pages are dropped and reloaded silently, so the numbers move even when nothing in the program changes. ### 11.11.3 PLAIN - a worked example

1. Real numbers from this chapter's test program.

Moment	VmSize	VmRSS
At start	2.6 MB	1.4 MB
After malloc 4 GB	4098.6 MB	1.7 MB
After touching 1 GB	4098.6 MB	1025.7 MB
After touching 4 GB	4098.6 MB	4097.7 MB

2. Virtual size jumped by 4096 MB the moment memory was requested. Resident size moved by 0.3 MB. Anyone monitoring virtual size would have raised a false alarm.
3. Now the sharing side, from `/proc/self/smmaps_rollup`: Rss 1640 kB but Pss 397 kB and Private_Dirty 108 kB.
4. Rule of thumb. To answer "will this fit", use PSS or private dirty. To answer "is this leaking", watch private dirty over time. Ignore virtual size unless you are chasing address space exhaustion.

■ 11.11.4 PLAIN – what is really happening inside

1. The kernel maintains, per process, a list of mappings and a count of resident pages.
2. Resident size is the count of pages currently present in physical memory, times the page size, whether shared or not.
3. To compute proportional size the kernel must, for every page, look up how many processes map it and divide. That is expensive, which is why `/proc/PID/smmaps` is much slower to read than `/proc/PID/status`.
4. Dirty pages are the ones modified since being loaded. Private dirty memory is the only kind that must be written to swap before it can be reclaimed.
5. This is why a process can show 2 GB resident and yet freeing it releases only 200 MB.

■ 11.11.5 TECHNICAL – the engineer's version

1. The counters and where they come from:

Counter	Linux source	Meaning
VSZ	VmSize in status	all mappings summed
RSS	VmRSS in status	resident, shared counted
PSS	smmaps, smmaps_rollup	shared divided by sharers
USS	Private_* in smmaps	unique to this process

2. Commands on Linux:

```
ps -o pid,vsz,rss,comm -p PID
cat /proc/PID/status | grep -E 'VmSize|VmRSS|VmSwap'
cat /proc/PID/smmaps_rollup      # Pss, Private_Dirty
pmap -X PID                     # per mapping detail
smem -k -s pss                  # sorted by PSS
systemd-cgtop / cat memory.current # cgroup v2 truth
```

3. On containers and cgroup v2, `memory.current` in the cgroup directory is the number the kernel enforces limits against, and it includes page cache charged to the group. A container killed for exceeding its limit was often killed by cache, not by the application heap. `memory.stat` breaks it down, and `memory.high` throttles before `memory.max` kills.
4. Commands on macOS:

```

vm_stat          # free, active, wired, compressed pages
top -l 1 -s 0 -n 0 # PhysMem, compressor size, swap
footprint -p PID  # Apple's authoritative per-process figure
vmmap PID        # every region, dirty and swapped columns
leaks PID        # unreachable allocations

```

- Why the totals never add up, stated precisely: the sum of RSS over all processes double counts every shared page once per sharer, and omits kernel allocations such as slab, page tables and network buffers, which belong to no process. The sum of PSS plus kernel usage does reconcile with `MemTotal - MemFree`, to within a few percent.
- On the machine used here, `free -m` reported total 8023, used 773, free 6573, buff/cache 910, available 7249. available is an estimate produced by the kernel of how much a new workload could get without swapping. It is the only number in that output worth alerting on.

■ 11.11.6 WORDS – remember these

- VSZ — address space claimed — virtual set size, the sum of all mapping lengths, including untouched ones.
- RSS — pages actually in RAM — resident set size, counting shared pages in full for every process.
- PSS — your share of the shared parts — proportional set size, private plus shared divided by sharer count.
- USS — what freeing it would return — unique set size, the private pages of one process.
- Available memory — what a new program could get — the kernel's estimate of free plus reclaimable, reported by `free`.

11.12 Cache-friendly programming

■ 11.12.1 PLAIN – in simple words

- You cannot change the memory hierarchy. You can change how your data sits in it, and that is often worth more than any algorithmic cleverness.
- Touch memory in the order it is stored. Neighbours, not jumps.
- Keep the things you use together next to each other, and the things you do not use out of the way.
- Work on a chunk small enough to stay in cache, finish with it, then move on.
- Make sure two threads never keep writing to the same 64-byte block. ### 11.12.2 PLAIN - a picture in your head
- Think of packing a suitcase for a trip.
- Array of structs is packing one bag per day, each with a shirt, socks, a book, a towel and a laptop charger.
- Struct of arrays is packing one bag of shirts, one of socks, one of books. Need shirts, carry one bag.
- Padding is leaving deliberate empty space so two people never reach into the same bag at once and knock each other's hands.
- Blocking is unpacking one bag at a time on a small table, rather than spreading all seven across the floor.

Where this comparison breaks: if you genuinely need every field of every item, array of structs is the better layout, because it fetches everything in one go. The right choice depends entirely on your access pattern, and there is no universally faster layout. ### 11.12.3 PLAIN - a worked example

1. Change one, loop order. Summing a 4096 by 4096 array of doubles, measured on the 2.1 GHz Xeon used throughout.

```
for (i...) for (j...) s += a[i][j]; /* 0.019 s */
for (j...) for (i...) s += a[i][j]; /* 0.187 s */
```

Result: 9.0 to 10.4 times faster by swapping two lines.

2. Change two, blocking. Multiplying two 1024 by 1024 matrices, plain triple loop against a version that works on 64 by 64 tiles so each tile stays in cache, both compiled with the vectorizer off so only the memory effect is being measured.

Version	Time
Plain i, j, k loops	4.622 s
Tiled, 64 x 64 blocks	0.563 s

Result: 8.21 times faster, same arithmetic, same result.

3. Change three, struct of arrays. Eight million particles of 40 bytes each, summing only the x field.

Layout	Bytes walked	Time
Array of structs	305 MB	0.0251 s
Struct of arrays	31 MB	0.0069 s

Result: 3.5 to 3.6 times faster, because 10 times less data crossed the memory bus.

4. Bonus change, field order. Two structs with identical fields in different order:

```
struct bad { char a; double b; char c; int d; }; /* 24 bytes */
struct good { double b; int d; char a; char c; }; /* 16 bytes */
```

Sorting fields from largest to smallest cut the size by a third, purely by removing padding. A million of them is 8 MB saved.

■ 11.12.4 PLAIN – what is really happening inside

1. Every read pulls a whole 64-byte line. Your effective bandwidth is (bytes you used) divided by (bytes fetched).
2. Row-order summing uses 64 of 64. Column-order uses 8 of 64. That is the factor of eight, before prefetching and vectorization make it worse.
3. In the particle test, the struct is 40 bytes and you wanted 4 of them. So each 64-byte line delivered about 6 useful bytes. The struct-of-arrays version delivered 64 useful bytes out of 64.
4. Tiling works because a naive matrix multiply walks a whole column of the second matrix for every output element, and that column does not stay in cache. Restricting the work to a 64 by 64 tile makes the three tiles involved fit together in cache, so each loaded value is reused 64 times.

11.12.5 TECHNICAL – the engineer's version

5. False sharing, measured for this chapter with two threads pinned to different cores, each doing 30 million atomic increments.

Layout	Time per increment	Penalty
Both counters in one line	27.8 to 35.2 ns	3.1x to 4.2x
Counters 128 bytes apart	6.6 to 11.5 ns	baseline

- The honest version: with plain non-atomic increments instead of atomic ones, the same test on the same machine showed no measurable penalty, about 0.49 ns per increment either way. A core can hold the line and retire many ordinary stores per ownership transfer, so the ping-pong amortizes. Atomic read-modify-write operations must be globally visible one at a time, so ownership really does move per operation. False sharing is therefore severe for atomics, counters and locks, and often mild for plain stores. Folklore usually omits this distinction.
- The fix, in three languages:


```
| struct counter { _Alignas(64) long value; };
| alignas(std::hardware_destructive_interference_size) long v;
| #[repr(align(64))] struct Padded(AtomicU64);
```
- Padding to 128 bytes rather than 64 is the safer choice on Intel parts, because the adjacent-line prefetcher can pull in the sibling line. Apple silicon uses 128-byte L2 lines. This is why C++17 provides `hardware_destructive_interference_size` rather than a fixed constant.
- Alignment rules: a scalar of size N must sit at an address that is a multiple of N. `sizeof` a struct is rounded up to its strictest member alignment so arrays stay aligned. Order fields from widest to narrowest and the padding usually disappears. Check with `pahole ./prog` on Linux, which prints holes explicitly, or `clang -Xclang -fdump-record-layouts`.
- A checklist that reliably finds wins, roughly in order of payoff:

Change	Typical gain here
Fix traversal order	9x
Block or tile the loop	8x
Array of structs to struct of arrays	3.5x
Pad shared counters	3x to 4x
Enable huge pages	1.3x
Reorder struct fields	size, not speed

- Verify every change with counters, not intuition:

```
| perf stat -e cycles,instructions,cache-misses,\
| LLC-load-misses,dTLB-load-misses ./prog
| perf c2c record ./prog && perf c2c report # false sharing
| valgrind --tool=cachegrind ./prog # simulated, exact
```

■ 11.12.6 WORDS - remember these

- Array of structs — one bag per item — AoS, contiguous records with all fields of one object together.
- Struct of arrays — one bag per field — SoA, parallel arrays with one field of all objects together.
- Padding — deliberate empty bytes — filler inserted by the compiler so each field meets its alignment requirement.
- Blocking — work on one tile at a time — loop tiling, restructuring loops so the active working set fits in a cache level.

5. False sharing — different variables, same line — performance loss when independent data shares one cache line across cores. ## 11.98 Common wrong ideas
6. Wrong: more RAM always makes a machine faster. Right: more RAM removes swapping. If you were not swapping, it changes nothing measurable. Going from 8 GB to 32 GB on a thrashing machine can feel like a new computer; going from 32 GB to 64 GB on a machine that never fills 20 GB changes nothing at all. Faster RAM helps a little; more RAM helps only if you were short.
7. Wrong: swap is bad and should be turned off. Right: swap lets the kernel evict genuinely cold anonymous pages and use that RAM for something useful. Without swap, cold pages are stuck in RAM forever and the only reclaim target left is the page cache, which often makes things worse. Gigabytes sitting in swap with `si` and `so` at zero in `vmstat` is healthy. Constant swap-in and swap-out at the same time is the problem, and that is thrashing, not swap.
8. Wrong: virtual memory is just the page file. Right: virtual memory is address translation. Every process gets its own map from private addresses to real frames. Paging to disk is one optional feature built on top. A machine with swap disabled still uses virtual memory for every instruction it executes.
9. Wrong: “free RAM is wasted RAM” is a myth invented by Linux fans. Right: it is literally how every modern kernel works, including Windows and macOS. Unused RAM is filled with cached file contents which are dropped instantly when a program needs the space. On the machine used here, `free -m` showed 6573 MB free but 7249 MB available, and the difference is exactly cache the kernel will give back for nothing.
10. Wrong: dual channel doubles memory speed. Right: it roughly doubles peak bandwidth, and does not reduce latency at all. A latency-bound program, for example one chasing pointers through a linked list, sees almost no benefit. A bandwidth-bound program, such as an integrated GPU or a large streaming copy, sees a lot.
11. Wrong: lower CAS latency always means faster memory. Right: CAS latency is in clock ticks, so it only means something alongside the clock. DDR4-3200 CL16 and DDR5-6000 CL30 both work out to 10.0 nanoseconds. Convert to nanoseconds with $CL \times 2000 / \text{rate}$ before comparing anything.
12. Wrong: if two processes have 2 GB resident each, they use 4 GB. Right: shared pages, mainly library code and copy-on-write pages after fork, are counted in full for every process. Use PSS, which divides shared pages between sharers and does add up correctly.
13. Wrong: huge pages always help. Right: they help when TLB misses dominate, which mainly means large random working sets. Measured here, transparent huge pages gave 1.3 times on a 512 MB random walk, and major database vendors recommend turning them off because the kernel’s compaction work causes latency spikes.

11.99 Chapter summary in 20 lines

1. Fast memory is small and dear, big memory is slow and cheap, so machines build a ladder from registers down to tape.
2. Measured on one 2.1 GHz machine, one random read cost 1.79 ns from L1 and 179.76 ns from RAM: a hundredfold spread for the same instruction.
3. In August 2026, DDR5 cost about 16.20 USD per gigabyte, SSD about 0.09 USD and hard disk about 0.02 USD, after a shortage that roughly quadrupled DRAM prices from 2024 levels.
4. The ladder works only because of locality: things used recently get used again, and their neighbours get used too.
5. Reading a 4096 by 4096 array along rows rather than down columns was 9.0 to 10.4 times faster, with no change to the arithmetic.

6. Memory arrives on sticks called DIMMs, organized into channels, ranks, bank groups, banks, rows and columns.
7. Extra channels multiply bandwidth and do nothing for latency, which is the most misunderstood fact about RAM.
8. True latency in nanoseconds is CL times 2000 divided by the data rate, so DDR4-3200 CL16 and DDR5-6000 CL30 are both exactly 10.0 ns.
9. Transfer rates rose about sixtyfold in twenty-five years while first-byte latency only halved. That gap is the memory wall.
10. The memory controller now lives on the CPU die, splits addresses into channel, rank, bank, row and column, and reorders requests, so latency is a distribution rather than a number.
11. Programs use virtual addresses; only the memory controller sees physical ones; the MMU translates on every access.
12. Memory is divided into 4 KB pages, mapped by a four-level or five-level page table tree rooted in CR3, cached by the TLB.
13. A real translation measured here turned 0x7f1e1a9ff010 into physical 0x162B8D010 through indices 254, 120, 212 and 511.
14. Neighbouring virtual pages land in scattered physical frames, which is the whole point of the mechanism.
15. Page faults are normal: a minor fault cost about 2 microseconds here, against 20 nanoseconds for an ordinary write, and touching 512 MB produced exactly 131,072 of them.
16. Demand paging means asking for 4 GB moves resident memory by 0.3 MB; memory appears only when written to.
17. Huge pages of 2 MB extend TLB reach from 8 MB to 4 GB and gave a measured 1.3 times on a large random walk.
18. Isolation comes free from the page tables themselves, reinforced by W^X from 2003, ASLR from 2001, and kernel page table isolation after Meltdown in 2018.
19. Copy-on-write makes fork cheap: forking a 1 GB process took 15 ms, and copying happened only when the child wrote, at 16 microseconds per page.
20. Layout beats cleverness: loop order gave 9x, tiling gave 8.2x, struct of arrays gave 3.5x, and padding shared atomic counters gave 3x to 4x, all on the same machine, all without changing a single result.

Storage — Disks, SSDs, Flash and Filesystems

12.0 What this chapter gives you

1. You will be able to say exactly what makes storage different from memory, and why every computer needs both.
2. You will be able to draw a hard disk drive and name every moving part in it, and say how one bit is written and read back.
3. You will be able to compute how long a spinning disk takes to reach a random piece of data, from the drive's rotation speed alone.
4. You will be able to explain a flash memory cell, why it wears out, and why erasing works on big blocks but writing works on small pages.
5. You will be able to explain why an SSD contains its own small computer, and what that computer spends its time doing.
6. You will be able to say why a big file copy to a cheap SSD starts fast and then suddenly slows down.
7. You will be able to trace a byte from a program down through the filesystem, the interface and the drive, and name each standard on the way.
8. You will be able to say what happens on disk when you create, write, rename and delete a file, step by step.
9. You will be able to explain why a 4 GB video will not copy onto a FAT32 USB stick, and what to do about it.
10. You will be able to explain journaling, RAID, backups and secure erase well enough to make real decisions with real money.

12.1 Storage versus memory: the real difference

■ 12.1.1 PLAIN - in simple words

1. Memory is where a computer keeps what it is thinking about right now.
2. Storage is where a computer keeps what it wants to still have tomorrow.
3. The big difference is what happens when the power goes off.
4. Memory forgets. Pull the plug and the contents are gone in under a second.
5. Storage remembers. Pull the plug, come back in a year, the file is still there.
6. A thing that keeps its contents without power is called **non-volatile**.
7. A thing that loses its contents without power is called **volatile**.
8. Memory is fast and expensive and small. Storage is slow and cheap and huge.

9. Both exist because no single technology is fast, cheap, huge and permanent all at once. Nobody has found one. Engineers pick two or three and stack the results in layers.

■ 12.1.2 PLAIN – a picture in your head

1. Think of a person working at a desk in a library.
2. The papers spread on the desk are memory. They are within arm's reach and you can grab any of them instantly.
3. The shelves around the room are storage. Getting a book takes you a walk.
4. The desk is small. You cannot fit the library on it.
5. At the end of the day the cleaner clears the desk completely. That is power loss. Anything not filed on a shelf is gone.
6. So you work fast on the desk, and you file to the shelves anything you want tomorrow.
7. There is also a basement archive, far away, cheap, enormous, and slow to reach. That is tape.

Where this comparison breaks: the desk in a real computer is not merely small, it is thousands of times faster than the shelves, not two or three times. Also, a librarian never has to rewrite a whole shelf to change one book, but a flash chip has almost exactly that problem, which we meet in section 12.5.

■ 12.1.3 PLAIN – a worked example

1. Suppose you edit a photo. The photo file lives on your SSD.
2. You double-click it. The computer copies the file from storage into memory.
3. Every brush stroke changes the copy in memory. The file on disk is untouched.
4. You press Save. Now the memory copy is written back down to storage.
5. If the power fails between step 3 and step 4, your strokes are gone and the old file is still there, unharmed.
6. That single sentence explains why software nags you to save.

■ 12.1.4 PLAIN – what is really happening inside

1. Memory in a normal computer is DRAM. Each bit is a tiny capacitor, a bucket holding a little charge.
2. The bucket leaks. It has to be topped up thousands of times per second. That topping up is called **refresh**, and it needs power.
3. Cut the power and every bucket empties within milliseconds. That is why RAM is volatile: not because of a rule, but because of a leak.
4. Storage stores a bit as something physical that does not need topping up: the direction of a magnet on a disk, or a pocket of trapped electrons in flash.
5. Nothing has to actively hold that state. It just sits there.
6. The price of permanence is speed. Flipping a magnet or pushing electrons through an insulator takes far longer than charging a tiny capacitor.
7. So the whole machine is built as a ladder. Tiny and instant at the top, huge and slow at the bottom, with each level acting as a cache for the one below.

■ 12.1.5 TECHNICAL – the engineer's version

1. The memory hierarchy spans roughly eight orders of magnitude in latency and six in price per byte. Figures below are typical for mid-2026 desktop and server parts.

Level	Typical size	Typical latency
CPU register	under 2 KB	under 1 ns
L1 data cache	32 to 128 KB per core	about 1 ns
L2 cache	0.5 to 4 MB per core	3 to 5 ns
L3 cache	16 to 128 MB shared	10 to 25 ns
DRAM (DDR5)	8 to 512 GB	60 to 100 ns
NVMe SSD	0.5 to 64 TB	20 to 80 us
SATA SSD	0.25 to 8 TB	about 100 us
Hard disk	1 to 44 TB	5 to 15 ms
LTO tape	18 to 40 TB per tape	30 to 90 s

2. Prices move fast and 2025 to 2026 was unusual. AI datacentre demand pulled NAND and DRAM capacity away from consumers and prices roughly doubled.

Medium	Price per TB, Aug 2026	Note
DDR5 DRAM	about 6,000 to 10,000 USD	approximate, volatile market
NVMe SSD, consumer	from about 105 USD	2 TB and 4 TB class
SATA SSD, consumer	from about 96 USD	4 TB class
Hard disk, 12 to 24 TB	about 21 to 32 USD	new and renewed stock
LTO tape media	roughly 6 to 10 USD	drive costs thousands

3. The honest version: “non-volatile” is not “eternal”. NAND flash loses charge over years when unpowered, magnetic domains slowly relax, and tape binders hydrolyse. Non-volatile means “does not need power to hold state for a useful period”, not “permanent”.
4. Byte-addressable persistent memory sat between DRAM and SSD for a few years. Intel Optane, based on 3D XPoint, shipped from 2017 with roughly 300 ns latency. Intel wound the business down in 2022. The idea is not dead, but no mainstream product occupies that rung today.
5. Tools that show the ladder: `lscpu` and `getconf -a` for cache sizes, `free -h` and `vm_stat` for memory, `lsblk` and `diskutil list` for block devices, `smartctl -a` for drive internals, `fio` for measured latency.

■ 12.1.6 WORDS – remember these

1. Volatile — forgets without power — state decays once supply is removed.
2. Non-volatile — remembers without power — retains state with supply removed.
3. DRAM — the fast forgetful memory — dynamic RAM, one capacitor and one transistor per bit, requiring periodic refresh.
4. Latency — the wait before anything arrives — time from request issue to first byte returned.
5. Throughput — how much arrives per second once it starts — sustained transfer rate, usually MB/s.
6. Memory hierarchy — the ladder of speeds — cache levels plus main memory plus backing store, each caching the level below.

12.2 The old ways: cards, tape, drums, cores and floppies

■ 12.2.1 PLAIN – in simple words

1. Before disks, computers stored data as holes in card, as magnetized spots on ribbon, and as rings of metal threaded on wire.

2. A **punched card** is a stiff paper card. A hole means one thing, no hole means the other. A machine feels for the holes with brushes or light.
3. **Paper tape** is the same idea on a long roll instead of separate cards.
4. **Magnetic tape** is a long plastic ribbon coated with rust-like powder. You magnetize small patches of it to record bits.
5. A **magnetic drum** is a spinning metal cylinder with heads along its side.
6. **Core memory** is a mesh of wires with tiny magnetic rings at every crossing. Each ring stores one bit by which way round it is magnetized.
7. A **floppy disk** is a thin flexible plastic disc, coated like tape, spun inside a sleeve.
8. All of these are gone from daily life except tape, which is quietly bigger than ever.

■ 12.2.2 PLAIN – a picture in your head

1. Picture a cassette tape and a vinyl record sitting next to each other.
2. The cassette must be wound forward to reach the middle of a song. To hear track eight you wait for tracks one to seven to go past.
3. The record lets you drop the needle straight onto track eight.
4. That is the whole difference between **sequential access** and **random access**, and it is the difference between tape and disk.
5. Tape is not slow because the ribbon moves slowly. Modern tape moves fast. Tape is slow to start because the right spot may be half a kilometre away along the ribbon.

Where this comparison breaks: a record needle can reach any groove in a second, while a disk head reaches any track in about ten milliseconds, a thousand times quicker. And nobody streams a record at 400 megabytes per second, which a modern tape drive does easily once it gets going.

■ 12.2.3 PLAIN – a worked example

1. The standard IBM punched card, in its final form from 1928, has 80 columns.
2. One column holds one character. So one card holds 80 characters, that is 80 bytes.
3. A standard box holds 2,000 cards. That is 160,000 bytes, about 160 kB.
4. A single modern phone photo of 4 MB would need 25 boxes of cards.
5. Stacked, those 50,000 cards stand about nine metres tall and weigh roughly 130 kilograms.
6. Now the modern end. An LTO-9 cartridge holds 18 TB of raw data in a plastic box the size of a thick paperback.
7. That is 18,000,000 MB, or about 4.5 million phone photos, or 225 million punched cards.

■ 12.2.4 PLAIN – what is really happening inside

1. On tape and on disk the recording trick is identical. A coil of wire carries a current, the current makes a magnetic field, and the field lines up the magnetic particles in the coating underneath.
2. Reverse the current and the particles line up the other way. Two directions, two symbols, one bit.
3. Nothing holds them there. They stay because the coating is chosen to be hard to demagnetize once set.
4. Core memory is the same physics with a different shape. Each ring can be magnetized clockwise or anticlockwise. Wires threaded through the ring set it and sense it.
5. Reading a core is destructive: to find out which way it was, you force it one way and see whether it flipped. If it flipped, it was the other way. The machine must then write the value back.
6. Every core in early machines was threaded by hand, by workers with needles and microscopes. Memory was literally woven.

7. A drum stores data on the outside of a spinning cylinder, with one fixed head per track. No head has to move, so the only wait is for the drum to come round. That made drums fast for their day and small in capacity.

■ 12.2.5 TECHNICAL – the engineer’s version

1. Gustav Tauschek patented magnetic drum memory in Austria in 1932. His drum held about 500,000 bits, roughly 62.5 kB.
2. Jay Forrester at MIT developed practical coincident-current magnetic core memory around 1949 to 1953; the Whirlwind I machine received core in 1953. An Wang and Way-Dong Woo filed related work in 1949.
3. The UNIVAC I UNISERVO of 1951 was the first tape drive on a commercial computer. Its tape was nickel-plated phosphor bronze, 128 characters per inch, about 12,800 characters per second.
4. IBM’s 726 tape unit of 1952 moved the industry to plastic tape at 100 bits per inch and about 7,500 characters per second.
5. Floppy disk lineage: IBM 23FD, 8-inch and read-only, 1971, about 80 kB. IBM 33FD, 8-inch read/write, 1973, 242.9 kB. Shugart SA400, 5.25-inch, 1976, about 110 kB. Sony’s 3.5-inch design from 1981 settled at 720 kB double-density and 1.44 MB high-density from 1987.

Format	Year	Capacity
8-inch, single density	1973	242.9 kB
5.25-inch DD	1978	360 kB
3.5-inch DD	1983	720 kB
3.5-inch HD	1987	1.44 MB
3.5-inch ED	1987	2.88 MB
Iomega Zip	1994	100 MB

6. Linear Tape-Open (LTO) is the surviving tape standard, an open format run by HP, IBM and Quantum. It is a genuine multi-vendor standard, not one company’s product.

Generation	Year	Native capacity
LTO-5	2010	1.5 TB
LTO-6	2012	2.5 TB
LTO-7	2015	6 TB
LTO-8	2017	12 TB
LTO-9	2021	18 TB
LTO-10	2025	30 TB, later 40 TB

7. LTO-9 and LTO-10 both run at 400 MB/s native. Marketing figures of 45 TB and 75 TB assume 2.5 to 1 hardware compression, which real data rarely reaches. Compressed figures are a marketing claim; native figures are the standard.
8. LTO-10 was announced on 13 August 2025 at 30 TB. In November 2025 the program raised it to 40 TB using an aramid-based tape substrate, and simultaneously cut its long-range roadmap: LTO-11 to 70 TB, LTO-12 to 120 TB, LTO-13 to 210 TB, LTO-14 to 365 TB.
9. Tape survives because of cost, air gap and shelf life. Media runs roughly 6 to 10 USD per TB against 21 to 32 USD for disk, a cartridge on a shelf cannot be reached by ransomware, and vendors rate media for about 30 years. The catch is the drive, which costs several thousand US dollars, so tape only pays off above roughly 100 TB of archive.

10. Tools: `mt` and `mt-st` to control a tape device, `tar` and `LTFS` to write it. `LTFS`, standardized as `ISO/IEC 20919`, makes a tape look like a filesystem.

■ 12.2.6 WORDS – remember these

1. Sequential access — you must go past everything in between — media where access time is proportional to distance along the medium.
2. Random access — you can jump straight there — access time roughly independent of the previous position.
3. Core memory — memory woven from magnetic rings — coincident-current ferrite-core store, destructive read with write-back.
4. LTO — the modern tape standard — Linear Tape-Open, open multi-vendor format, generation 10 at 30 to 40 TB native.
5. Air gap — the archive is physically unplugged — offline copy unreachable by any network attacker.
6. Native capacity — the honest number — raw bytes stored with no compression assumed.

12.3 The hard disk drive, in full

■ 12.3.1 PLAIN – in simple words

1. A hard disk is a stack of smooth metal or glass discs that spin very fast.
2. Each disc is coated with a magnetic film. That film is where the bits live.
3. A small arm swings across the discs, like the arm of a record player.
4. On the tip of the arm sits a **head**, which both writes and reads.
5. To write, the head makes a tiny magnetic field that flips a patch of the coating to point one way or the other.
6. To read, the head senses those patches as they rush past underneath.
7. The head never touches the disc. It flies above it on a cushion of air that the spinning disc drags along with it.
8. The gap is a few nanometres. That is smaller than a virus, thousands of times thinner than a hair.
9. If the head touches the disc at full speed, it gouges the coating. That is a head crash, and the data in that path is gone forever.

■ 12.3.2 PLAIN – a picture in your head

1. Imagine a jumbo jet flying at full cruising speed, but only one millimetre above the ground, counting blades of grass as it goes.
2. Now scale that picture down to the size of a coin, and you have a disk head.
3. The head is a small block, roughly the size of a grain of salt cut in four.
4. The disc under it moves at around 100 kilometres per hour at the outer edge.
5. The arm can swing from the middle to the edge and stop on a chosen track in about eight thousandths of a second.
6. The whole thing is sealed, because a single speck of dust is a boulder at that height.

Where this comparison breaks: an aircraft is held up by its own wings and its own engines, while a disk head is a passive glider held up entirely by air the disc drags along. Slow the disc and the head lands. And the head is not merely looking at the ground, it is also repainting it as it goes.

■ 12.3.3 PLAIN – a worked example

1. Take a common 7,200 RPM desktop drive. RPM means revolutions per minute.
2. 7,200 revolutions per minute is 120 revolutions per second.
3. One revolution therefore takes 1 divided by 120, which is 8.33 milliseconds.
4. Suppose the data you want is on the track under the head already. On average you wait half a turn for it to come round: 4.17 milliseconds.
5. Now suppose it is on a different track. The arm must move. That takes about 8.5 milliseconds on average for a desktop drive.
6. Total wait: about 12.7 milliseconds before a single byte arrives.
7. In those 12.7 milliseconds a modern CPU at 4 GHz executes roughly 50 million clock cycles. The processor is waiting an eternity.

■ 12.3.4 PLAIN – what is really happening inside

1. The magnetic film is made of grains. Each grain is a tiny permanent magnet with a preferred direction it likes to point.
2. A group of grains all pointing the same way is a **magnetic domain**. One recorded bit uses tens of grains, not one. Using many grains averages out the noise so the read-back is reliable.
3. Writing: current through a coil in the head makes a field at a narrow gap. The field is strong enough to force the grains under it to flip direction.
4. Reading: here is the part that surprises people. The head does not sense “north” or “south”. It senses **changes**.
5. As the disc moves, wherever the magnetization reverses, there is a small stray field. The read sensor detects those reversals.
6. So the drive does not store 1 as north and 0 as south. It stores a stream of reversals, and an encoding scheme turns reversal patterns into bits.
7. That is why a long run of identical bits is a problem: no reversals means nothing to sense, and the drive loses its place. Encodings deliberately guarantee frequent reversals.
8. The head assembly is moved by a **voice coil**: a coil of wire sitting in a strong permanent magnet, exactly like the motor in a loudspeaker. Push current through it and the arm swings. Reverse it and the arm swings back.
9. The disc stack is turned by a **spindle motor** running on a fluid bearing, which is quieter and more precise than balls.

■ 12.3.5 TECHNICAL – the engineer’s version

1. The IBM 350 Disk Storage Unit, part of the IBM 305 RAMAC, was announced on 14 September 1956. It held 5 million six-bit characters across fifty 24-inch discs, with an average access time of 600 ms. It weighed over a ton and rented for about 3,200 US dollars a month.
2. In 1956 that is roughly 3.75 MB in modern eight-bit bytes, in a cabinet the size of two large refrigerators, delivered by cargo aircraft and moved with forklifts.
3. Flying height fell from about 20 micrometres in 1956 to roughly 1 to 3 nanometres today. Head sliders shrank through “pico” (1.25 by 1.0 by 0.3 mm) to “femto” (0.85 by 0.7 by 0.23 mm) classes.
4. The read sensor changed three times. Inductive coils gave way to anisotropic magnetoresistance, then to **giant magnetoresistance** (GMR), discovered independently in 1988 by Albert Fert in France and Peter Grunberg in Germany. IBM shipped the first GMR-head drive, the Deskstar 16GP, in 1997. Fert and Grunberg shared the 2007 Nobel Prize in Physics for the discovery. Modern heads use tunnelling magnetoresistance, a further refinement.

5. GMR is a genuine case of a physics discovery reaching a shipping product in nine years, and it is why disk capacity grew so fast in the late 1990s.
6. Recording orientation changed too. **Longitudinal** recording laid magnets flat, in the plane of the disc. **Perpendicular magnetic recording** (PMR) stands them on end, packing them tighter and resisting thermal decay. Toshiba shipped the first PMR drive in 2005 and Seagate followed in 2006. All modern drives are perpendicular.
7. **Shingled magnetic recording** (SMR) overlaps write tracks like roof tiles, because a write head is wider than a read head. It buys about 20 percent capacity at a severe cost: rewriting one track means rewriting the whole band. Drive-managed SMR hides this and can stall for seconds under sustained random writes. Western Digital shipping undisclosed SMR in Red NAS drives caused a public dispute in 2020, and the industry now labels it.
8. **Heat-assisted magnetic recording** (HAMR) puts a laser in the head. It heats a spot to roughly 400 to 450 degrees Celsius for about a nanosecond, which briefly makes a very stable, very small grain writable. Seagate's Mozaic 3+ platform shipped from 2024, and Mozaic 4+ began shipping 44 TB drives with ten platters at over 4 TB each in March 2026. Western Digital pursued microwave-assisted recording (MAMR) instead.

Parameter	1956 IBM 350	2026 HAMR drive
Capacity	about 3.75 MB	44 TB
Platters	50	10
Platter diameter	24 inch	3.5 inch
Access time	600 ms	about 8 ms

9. Drives above about 8 TB are usually sealed and filled with helium, first shipped by HGST as the Ultrastar He6 in 2013. Helium is one seventh the density of air, so there is less turbulence and less drag, which allows more platters in the same height and cuts power.
10. Areal density is the headline metric, in bits per square inch. Production drives are now in the region of 1.5 to 2 terabits per square inch, and HAMR is the path beyond.
11. Tools: `smartctl -a /dev/sda` reads SMART attributes including reallocated sector count and head flying-height sensors on some models.

■ 12.3.6 WORDS – remember these

1. Platter — one of the spinning discs — rigid aluminium or glass substrate with a sputtered magnetic film.
2. Head — the part that reads and writes — the slider carrying a write coil and a magnetoresistive read sensor.
3. Air bearing — the cushion the head flies on — the thin viscous film of gas dragged by the rotating platter, a few nanometres thick.
4. Magnetic domain — a patch of magnets all pointing the same way — a region of aligned grains representing part of one recorded symbol.
5. Voice coil actuator — the loudspeaker motor that swings the arm — rotary VCM under closed-loop servo control.
6. GMR — the effect that made big disks possible — giant magnetoresistance, Fert and Grunberg 1988, Nobel Prize 2007.
7. HAMR — heat the spot before writing it — heat-assisted magnetic recording, using a near-field laser to lower coercivity momentarily.

8. SMR — tracks overlapped like roof tiles — shingled magnetic recording, requiring band rewrites, sold as drive-managed or host-managed.

12.4 HDD geometry and performance

■ 12.4.1 PLAIN – in simple words

1. The surface of a disc is divided into rings called **tracks**.
2. Each track is chopped into short arcs called **sectors**. A sector is the smallest chunk the drive will read or write.
3. A sector used to hold 512 bytes. Modern drives use 4,096 bytes.
4. There are several platters stacked up, each with a surface on top and one underneath, each with its own head.
5. All the heads move together on one arm. So track 500 on every surface is reachable without moving the arm. That set of tracks is a **cylinder**.
6. To reach data the drive does three things: move the arm to the right track, wait for the right sector to come round, then transfer the data.
7. Moving the arm is **seek time**. Waiting for rotation is **rotational latency**. Only the third part actually moves data.
8. If your next request is right next to the last one, you skip the first two steps. That is why reading a file in order is fast.
9. If your next request is somewhere random, you pay both waits every time. That is why random access on a hard disk is brutally slow.

■ 12.4.2 PLAIN – a picture in your head

1. Picture a very large car park laid out in concentric rings, with numbered bays around each ring.
2. You are on a moped in the middle. Someone radios you a bay number.
3. First you ride out to the right ring. That is the seek.
4. But the car park is on a turntable, spinning. You must wait for your bay to come round to you. That is the rotational latency.
5. Then you read the number plate as it passes. That is the transfer.
6. If the next twenty bays you need are the next twenty in the same ring, you just sit still and read them all as they go by. Wonderful.
7. If each next bay is in a random ring, you spend all day riding and waiting and almost no time reading.

Where this comparison breaks: a real drive also reads ahead, caches, and reorders queued requests to reduce total travel, so a clever drive is better than a moped rider taking orders one at a time. And the drive's own controller hides a lot of this, so the outside world never sees the real geometry.

■ 12.4.3 PLAIN – a worked example

1. Rotational latency is pure arithmetic. Average latency is the time for half a revolution.
2. Formula: average rotational latency in milliseconds equals 30,000 divided by the RPM.
3. At 5,400 RPM: 30,000 divided by 5,400 equals 5.56 ms.
4. At 7,200 RPM: 30,000 divided by 7,200 equals 4.17 ms.
5. At 10,000 RPM: 3.00 ms. At 15,000 RPM: 2.00 ms.
6. Now add a typical average seek. Take a 7,200 RPM drive with an 8.5 ms seek.
7. Average access time is 8.5 plus 4.17, about 12.7 ms.
8. Operations per second is 1 divided by 0.0127, about 79.

9. So a 7,200 RPM drive does roughly 80 random operations per second. That number has barely changed since 1995.

```

One random read on a 7200 RPM drive
seek 8.5 ms      rotate 4.17 ms      transfer 0.04 ms
|=====| |=====| |
0          8.5       12.7       12.74 ms
                        ^ 4 KB actually moved here
99.7 percent of the time is waiting, not reading.

```

■ 12.4.4 PLAIN – what is really happening inside

1. Old drives were addressed by **cylinder, head, sector**: tell the drive which ring, which surface, which arc. This is CHS addressing.
2. That worked while software could know the real geometry. It stopped working the moment drives started lying about their shape.
3. Drives had to lie because of a physical fact: outer tracks are longer than inner tracks. A fixed number of sectors per track wastes the outside.
4. So drives use **zoned recording**: more sectors per track near the edge, fewer near the middle. Geometry is no longer uniform, so CHS is fiction.
5. The fix is **logical block addressing**, LBA. The drive presents a simple numbered list of sectors, 0, 1, 2, and upward. The drive alone knows where block 4,000,000 physically is.
6. Zoned recording has a visible side effect: the outer tracks pass under the head faster, so they transfer more bytes per second. A drive is genuinely faster at the start of its address range than at the end, often by half.
7. Sectors are not just data. Each carries a preamble to synchronize timing, a servo pattern telling the head where it is, the data itself, and an error correcting code.
8. Moving from 512-byte to 4,096-byte sectors saves all that overhead per byte and allows a stronger error code. That is **Advanced Format**.

■ 12.4.5 TECHNICAL – the engineer's version

1. CHS addressing in the PC BIOS interrupt 13h interface allowed 1,024 cylinders, 256 heads and 63 sectors, capping capacity at 8.4 GB with 512-byte sectors. Various translation hacks pushed it further before LBA replaced it outright.
2. LBA-28 in ATA capped out at 2^{28} sectors of 512 bytes, that is 128 GiB, usually quoted as 137 GB. LBA-48, introduced in ATA-6 in 2003, raises the cap to 2^{48} sectors, about 128 PiB.
3. Advanced Format was defined by IDEMA and the industry transitioned from around January 2011. Two variants exist: 512e presents 512-byte logical sectors over 4,096-byte physical ones, and 4Kn presents 4,096 bytes both ways.
4. On a 512e drive, a misaligned 4 KB write triggers a read-modify-write of the physical sector, roughly halving small random write performance. Aligning partitions to a 1 MiB boundary, now the default in every modern partitioner, avoids this.
5. Sustained transfer rate depends on linear density and rotation. A modern 7,200 RPM 20 TB drive sustains roughly 260 to 290 MB/s on outer tracks and about 120 to 140 MB/s on inner tracks. Seagate quotes about 300 MB/s for the 44 TB Mozaic 4+ drives.

Drive class	Rot. latency	Typical random IOPS
5,400 RPM laptop / NAS	5.56 ms	50 to 75
7,200 RPM desktop	4.17 ms	75 to 100
10,000 RPM enterprise	3.00 ms	125 to 150

Drive class	Rot. latency	Typical random IOPS
15,000 RPM enterprise	2.00 ms	175 to 210

- Native Command Queuing, part of SATA, lets a drive hold up to 32 outstanding commands and service them in an efficient order. It can raise random IOPS by 30 to 50 percent at high queue depth by reducing total head travel.
- The gap between sequential and random is not subtle. A 20 TB drive doing 1 MB sequential reads delivers roughly 260 MB/s. The same drive doing 4 KB random reads at queue depth 1 delivers about 80 times 4 KB, that is roughly 0.32 MB/s. The ratio is about 800 to 1.
- Measure it yourself:

```
# Sequential read throughput, 1 MB blocks
fio --name=seq --rw=read --bs=1M --size=4G \
    --filename=/dev/sdb --direct=1

# Random read IOPS, 4 KB blocks, one at a time
fio --name=rand --rw=randread --bs=4k --size=4G \
    --iodepth=1 --filename=/dev/sdb --direct=1
```

- Inspect logical and physical sector size with `lsblk -o NAME,PHY-SEC,LOG-SEC` on Linux, or `fsutil fsinfo sectorinfo C:` on Windows.

■ 12.4.6 WORDS – remember these

- Track — one ring on the disc surface — a single concentric data path.
- Sector — the smallest readable chunk — a physical block, 512 or 4,096 bytes plus servo, sync and ECC overhead.
- Cylinder — the same track on every surface — the set of tracks reachable without moving the actuator.
- Seek time — the wait while the arm moves — actuator settling time to a target track, quoted as an average over random seeks.
- Rotational latency — the wait for the sector to come round — 30,000 divided by RPM in milliseconds, on average.
- LBA — one long numbered list of blocks — logical block addressing, hiding real geometry behind a linear address space.
- Advanced Format — 4K sectors instead of 512 — 512e or 4Kn physical sector formats defined by IDEMA.
- IOPS — operations per second — input/output operations per second, always quoted with block size and queue depth or it is meaningless.

12.5 Flash memory, in full

■ 12.5.1 PLAIN – in simple words

- Flash memory stores a bit by trapping electrons inside an insulator, where they cannot get out on their own.
- A cell is a transistor with an extra hidden pocket built into it.
- Put electrons in the pocket and the transistor becomes harder to switch on.
- Leave the pocket empty and it switches on easily.
- To read a cell you apply a test voltage and see whether it switches on. That tells you whether the pocket is full.

6. To write, you shove electrons through the insulator into the pocket using a high voltage.
7. To erase, you shove them all back out again, also with a high voltage.
8. The insulator is what makes it non-volatile. Electrons stay put with no power. The insulator is also what wears out, because forcing charge through it damages it a little every single time.
9. That is the whole of flash memory. Everything else is engineering around those two sentences.

■ 12.5.2 PLAIN – a picture in your head

1. Think of a water tank with no tap and no drain, sealed on all sides.
2. Normally water cannot get in or out. That is your data sitting safe.
3. To fill it you use enormous pressure to force water straight through the wall of the tank. To empty it you do the same in reverse.
4. It works, but every time you do it the wall gets a little more damaged.
5. After enough fills and empties the wall starts to seep. Now the tank slowly loses water on its own, and a tank you filled last year no longer reads full.
6. That is exactly why flash memory has a limited number of writes, and why an old worn drive also forgets faster when left unplugged.
7. Now the reading part. You do not open the tank to check it. You judge the level from outside by how much it resists a push.
8. If the tank holds only “empty” or “full”, that judgement is easy. If you try to distinguish sixteen different water levels, it gets very hard, and small leaks matter enormously.

Where this comparison breaks: electrons are not water and do not slosh. The “pressure” is a quantum effect called tunnelling, where an electron simply appears on the far side of a thin barrier. And the damage is not a hole in a wall, it is charge getting stuck inside the insulating oxide itself.

■ 12.5.3 PLAIN – a worked example

1. Cells are cheaper if you store more than one bit each, by using more charge levels. Here is what that costs you.

Type	Bits per cell	Voltage levels
SLC	1	2
MLC	2	4
TLC	3	8
QLC	4	16
PLC	5	32

2. Going from SLC to QLC gives you four times the capacity from the same silicon. That is why every consumer drive is TLC or QLC.
3. But the voltage window is fixed. With 16 levels in the same window, each level is a narrow band, and a small charge leak pushes a cell into the neighbouring band and corrupts the value.
4. So endurance and speed collapse as bits per cell rise.

Type	P/E cycles	Relative write speed
SLC	50,000 to 100,000	fastest
MLC	3,000 to 10,000	about half SLC
TLC	1,000 to 3,000	about a third
QLC	100 to 1,000	about a fifth

Type	P/E cycles	Relative write speed
PLC	under 100, projected	not yet in products

5. PLC is active research, not a shipping product as of 2026. Treat any PLC capacity claim as a roadmap, not a specification.

■ 12.5.4 PLAIN - what is really happening inside

1. Cells are wired into a grid. Rows are called **word lines**, columns are called **bit lines**.
2. In **NOR flash** every cell hangs directly off a bit line, in parallel. You can read any single byte instantly, but each cell needs its own contact, so cells are large and the chip is expensive per byte.
3. In **NAND flash** the cells are wired in long series strings, like old Christmas lights. Far fewer contacts are needed, so cells pack much tighter.
4. The price of that is that you cannot touch one cell alone. To read one cell you must switch every other cell in its string fully on so current can pass through them.
5. So NAND works on whole rows at a time. A row of cells read together is a **page**. Pages are grouped into a **block**. Blocks are grouped into **planes**, and planes into a **die**.
6. Now the crucial asymmetry, the fact that shapes every SSD ever built:
 1. You can read one page.
 2. You can write one page, but only if it is currently erased.
 3. You cannot erase one page. You can only erase a whole block.
7. Writing pushes cells one direction only. Erasing is the only way back, and erasing is coarse.
8. So changing one byte in a page is impossible in place. The drive must write the new version of that page somewhere else entirely, and remember where.
9. That single restriction is why an SSD needs the complicated software we meet in section 12.6.
10. **3D NAND** is the other big idea. Instead of shrinking cells sideways, which stopped working around 2013, manufacturers turned the strings on end and stacked layers vertically, then drilled holes through the stack.
11. This let them go back to a relaxed, reliable cell size and gain density by height instead. It is the single reason flash kept getting cheaper.

■ 12.5.5 TECHNICAL - the engineer's version

1. The floating-gate MOSFET was invented by Dawon Kahng and Simon Min Sze at Bell Labs in 1967. It is an ordinary MOS transistor with an electrically isolated polysilicon gate buried in the oxide between control gate and channel.
2. Fujio Masuoka at Toshiba invented flash memory around 1980. NOR flash was presented in 1984 and Intel shipped the first commercial NOR part in 1988. NAND flash was presented at the IEEE International Electron Devices Meeting in San Francisco in 1987. The name came from a colleague who thought the block erase resembled a camera flash.
3. Charge is moved by Fowler-Nordheim tunnelling for erase and either tunnelling or channel hot-electron injection for program, at internally generated voltages of roughly 15 to 20 V produced by on-chip charge pumps.
4. Trapped charge shifts the transistor threshold voltage. Reading is a sequence of sense operations at different word-line reference voltages; a TLC read needs seven reference points to separate eight states.
5. **Charge trap flash** replaced the conducting floating gate with an insulating silicon nitride layer that traps charge in discrete defect sites. Because the storage layer is not conductive, a single defect does not drain the whole cell, which is what makes tall 3D stacks practical. Samsung V-

NAND has used charge trap from the start; Intel and Micron used a floating-gate 3D design for several generations before moving across.

6. Modern 3D NAND geometry, typical of 2024 to 2026 TLC parts:

Unit	Typical size	Operation time
Page	16 KiB	read 50 to 90 us
Page program	16 KiB	500 to 2,500 us
Block	1,000 to 2,000 pages	erase 2 to 10 ms
Die (LUN)	512 Gbit to 2 Tbit	4 to 6 planes

7. Note the ratio: erasing a block costs roughly one hundred times a page read, and it destroys everything in that block. This is the reason garbage collection exists.
8. Layer counts as of 2026: SK hynix is in mainstream production at 321 layers, Samsung's V9 generation is in the high 200s, Micron's G9 is around 276, and Kioxia and SanDisk's BiCS8 is 218. Samsung announced a 400-layer part for 2026 and demonstrated a 900-layer research die in May 2026 by bonding two 450-layer stacks. Stacks above roughly 128 layers are almost always built as two or more bonded decks, a technique called string stacking.
9. NOR flash did not die. It survives wherever code must be executed directly from the chip with no controller in the way: BIOS and UEFI firmware, boot ROMs, microcontrollers and automotive parts. NOR gives byte-level random read and execute-in-place; NAND gives density and cost per bit. NAND won storage, NOR kept firmware.
10. Inspect real flash geometry on Linux with `nvme id-ctrl /dev/nvme0` and `nvme id-ns /dev/nvme0n1`, which report namespace sizes and, on drives that support it, optimal write size and NPWG or NPWA alignment hints.

■ 12.5.6 WORDS – remember these

1. Floating gate — the hidden pocket that holds electrons — an isolated polysilicon gate whose stored charge shifts the transistor threshold.
2. Charge trap — the same job done by an insulator — a silicon nitride layer trapping charge in defect sites, standard in 3D NAND.
3. Tunnelling — electrons crossing a barrier they should not — Fowler-Nordheim tunnelling through the tunnel oxide under a high field.
4. Page — the smallest thing you can read or write — typically 16 KiB, plus spare area for error correction.
5. Block — the smallest thing you can erase — one to two thousand pages, several megabytes.
6. SLC, MLC, TLC, QLC — how many bits crammed into one cell — 1, 2, 3 and 4 bits, using 2, 4, 8 and 16 threshold levels.
7. P/E cycle — one wear event — a complete program and erase of a block, the unit in which flash lifetime is counted.
8. 3D NAND — stacking cells upward instead of shrinking them — vertical strings through a multi-layer stack, now over 300 layers.

12.6 The SSD controller and the flash translation layer

■ 12.6.1 PLAIN – in simple words

1. An SSD is not just flash chips in a box. It contains a small computer whose only job is to manage those chips.

2. That computer has its own processor cores, its own firmware and often its own memory.
3. It exists because of the asymmetry in the last section: the outside world wants to overwrite any block at any time, and flash cannot do that.
4. So the controller lies, politely and consistently. It tells your operating system “here is a simple numbered list of blocks you can overwrite freely”.
5. Underneath, it never overwrites anything. It writes each new version to a fresh place and updates a private map.
6. That map is the **flash translation layer**, or FTL. It converts the address your computer asked for into the physical page where the data really is.
7. The controller must also spread wear evenly, clean up rubbish in the background, and hide all of it from you.
8. Almost everything odd about SSD behaviour comes from that hidden work.

■ 12.6.2 PLAIN – a picture in your head

1. Picture a large notebook where you are not allowed to use an eraser on one line. You can only tear out and rewrite a whole page at a time.
2. So when you want to change one line, you write the new version on the next blank line instead, and you keep an index at the front saying which line is the current one.
3. The old line is now rubbish, but it is still sitting there taking space.
4. Eventually the notebook fills with a mixture of current lines and rubbish lines scattered across every page.
5. To get space back you pick a page that is mostly rubbish, copy its few surviving lines onto a fresh page, update the index, and tear out the old page.
6. That tidying is **garbage collection**. It costs real work that nobody asked for, and it is why an SSD writes more than you told it to.
7. And you deliberately spread your writing across all pages rather than always using the front ones, so no page wears out first. That is **wear levelling**.

Where this comparison breaks: a real controller does this tidying while you are also writing, competing for the same chips, which is why performance can dip under sustained load. And the index itself must be kept safely, because losing it loses the whole drive, not one page.

■ 12.6.3 PLAIN – a worked example

1. Here is **write amplification** with real arithmetic.
2. Take a block of 256 pages. 192 pages hold live data, 64 hold stale data.
3. You want to reclaim that block. First you must copy the 192 live pages elsewhere, then erase the block.
4. So to free 64 pages of space, the drive performed 192 page writes.
5. Now suppose your application writes 64 pages of new data into that freed space. Total flash writes: 192 copied plus 64 new equals 256.
6. Host writes: 64 pages. Flash writes: 256 pages.
7. Write amplification factor equals 256 divided by 64, which is 4.
8. Your drive is wearing out four times faster than your data volume suggests.
9. Now repeat the calculation on a drive that is only half full, where a typical block has 64 live pages and 192 stale.
10. Copy 64 live pages, free 192, then write 192 new. Flash writes 64 plus 192 equals 256, host writes 192, amplification is 1.33.

11. That is the whole explanation of why a full SSD is slow and short-lived and an empty one is fast. Free space is not idle. It is working capital.

■ 12.6.4 PLAIN – what is really happening inside

1. The FTL keeps a table. Logical block number in, physical page location out.
2. Every write allocates a fresh page, writes there, and updates one table entry. Nothing is ever changed in place.
3. **Wear levelling** comes in two kinds. Dynamic levelling spreads new writes over blocks that are already free. Static levelling also moves data that has sat untouched for a long time, so that cold blocks take their share of wear. Without static levelling, a drive holding a large read-only archive would wear out the small remaining area very fast.
4. **Over-provisioning** is hidden spare capacity the controller keeps for itself. It never appears in your capacity figure. More spare space means garbage collection always has somewhere easy to work, which lowers write amplification.
5. **TRIM** solves a problem you would not guess at. When you delete a file, the filesystem just marks its blocks free in its own records. It does not tell the drive.
6. So the drive still believes every one of those pages holds live data, and dutifully copies them during garbage collection forever.
7. TRIM is a command that says “these logical blocks no longer contain anything you need to preserve”. Now the controller can drop them, and the pages become free for nothing.
8. Without TRIM, an SSD gradually behaves as if it were completely full, even when the filesystem shows it as empty.
9. **SLC caching** is why big copies slow down. TLC and QLC cells can be operated in one-bit mode, which is much faster. Controllers keep part of the flash in that mode as a fast landing zone.
10. Incoming writes land in the fast zone at full speed. Later, when the drive is idle, they are folded down into dense multi-bit storage.
11. If you write more than the cache holds, the cache runs out mid-copy and writes must go straight to slow storage, while folding also competes for the same chips. Speed drops sharply and visibly.
12. The FTL table is far too large to search through, so the controller keeps it in fast memory. Either a dedicated DRAM chip on the drive, or a slice of your computer’s own memory borrowed over the interface.

■ 12.6.5 TECHNICAL – the engineer’s version

1. Typical controllers are multi-core embedded designs. Phison E26 and E28, Silicon Motion SM2508, and Samsung’s in-house controllers use two to five ARM Cortex-R class cores plus hardware engines for ECC, encryption and DMA.
2. Error correction is not optional. Modern TLC and QLC require LDPC (low-density parity-check) decoding with soft-decision reads. Raw bit error rates in the 10^{-3} range are corrected down to below 10^{-15} .
3. Mapping granularity drives the DRAM requirement. A page-mapped FTL with 4 KiB granularity and a 4-byte entry needs about 1 GB of DRAM per 1 TB of NAND. That is the origin of the industry rule of thumb.
4. **Host Memory Buffer** (HMB) is an NVMe feature that lets a DRAM-less drive borrow host RAM, typically 32 to 64 MB. It holds only a hot fraction of the map, so DRAM-less drives are competitive on light desktop work and clearly worse on sustained random writes.
5. Over-provisioning has two sources. First, the decimal versus binary gap: a drive sold as 1 TB (10^{12} bytes) built from 1 TiB (1.0995 times 10^{12} bytes) of NAND has 7.37 percent spare for

free. Enterprise parts add more deliberately, which is why capacities read 960 GB, 1,920 GB or 3,840 GB instead of round powers.

Class	Usable / raw	Over-provision
Consumer 1 TB	1,000 / 1,099 GB	about 7 percent
Enterprise 960 GB	960 / 1,099 GB	about 13 percent
Enterprise 800 GB	800 / 1,099 GB	about 27 percent

6. TRIM is the ATA DATA SET MANAGEMENT command with the deallocate bit; the SCSI equivalent is UNMAP, and NVMe uses Dataset Management with the deallocate attribute. Windows 7 (2009) was the first mainstream OS to issue it, Linux gained support in kernel 2.6.33 (2010), and macOS enabled it for Apple SSDs from 10.6.8 (2011).
7. Two ways to issue it on Linux: continuous with the discard mount option, or batched on a timer, which is now the preferred default.

```
# See whether the device reports discard support
lsblk -D          # DISC-GRAN and DISC-MAX columns

# Run a batched trim now, and report bytes released
sudo fstrim -av

# Check the systemd timer that normally does this weekly
systemctl status fstrim.timer
```

8. SLC cache sizing is an implementation detail that varies per model. A typical 2 TB consumer QLC drive might sustain 5,000 MB/s for the first 150 to 250 GB and then fall to 100 to 450 MB/s. TLC drives usually fall to 900 to 1,700 MB/s instead, which is far less painful. Reviewers call this the post-cache or steady-state write speed, and it is the number that matters for video work and large backups.
9. **Power-loss protection** in enterprise drives is a bank of tantalum or ceramic capacitors sized to flush the volatile write buffer and the mapping table to NAND after supply is cut. Consumer drives almost never have it. Without PLP, a drive may honestly acknowledge a write that is still in volatile buffer, which is why databases insist on explicit cache flushes.
10. Observe the drive's own accounting with `nvme smart-log /dev/nvme0`, which reports `data_units_written`, `percentage_used` and `unsafe_shutdowns`. On drives exposing vendor logs, `smartctl -a` may also show host writes versus NAND writes, from which you can compute a real write amplification factor.

■ 12.6.6 WORDS – remember these

1. FTL — the private map from your addresses to real chips — flash translation layer, maintaining logical-to-physical mapping and block state.
2. Wear levelling — spread the damage evenly — dynamic and static algorithms equalizing program/erase counts across blocks.
3. Garbage collection — tidying up to reclaim erased blocks — relocating valid pages out of victim blocks before erasing them.
4. Write amplification — the drive writes more than you asked — WAF, the ratio of NAND writes to host writes.
5. Over-provisioning — hidden spare room — reserved capacity invisible to the host, lowering WAF and raising sustained performance.

6. TRIM — telling the drive a block is now rubbish — the deallocate hint, ATA DATA SET MANAGEMENT, SCSI UNMAP, NVMe Dataset Management.
7. SLC cache — a fast landing strip made of slow flash — a region operated in pseudo-SLC mode, later folded into TLC or QLC.
8. HMB — the drive borrows your computer's memory — Host Memory Buffer, an NVMe feature used by DRAM-less controllers.
9. Power-loss protection — capacitors that finish the job — onboard energy reserve flushing volatile buffers on supply loss.

12.7 SSD endurance and failure

■ 12.7.1 PLAIN – in simple words

1. Every flash block can only be erased and rewritten a limited number of times.
2. When a block's insulator is too damaged to hold charge reliably, the controller retires it and uses a spare.
3. When the spares run out, the drive stops accepting writes.
4. Manufacturers publish this as **terabytes written**, or TBW: how much data you may write over the warranty period.
5. Reading is nearly free, but not entirely. Reading a page slightly disturbs its neighbours, so after a great many reads a block must be refreshed.
6. A flash cell also leaks charge slowly. An unpowered SSD gradually forgets, over months to years, and forgets faster if it is worn or hot.
7. Hard drives usually die gradually and noisily. SSDs usually die suddenly and silently.
8. That difference matters more for your backup plan than any speed figure in this chapter.

■ 12.7.2 PLAIN – a picture in your head

1. Think of a hard drive as an old car with 300,000 kilometres on it. It squeaks, it leaks, warning lights come on, one cylinder misfires. You get weeks of notice that something is wrong.
2. Think of an SSD as a light bulb. It works perfectly, perfectly, perfectly, and then it does not work at all, with no warning of any kind.
3. Most hard drive failures leave the platters intact and let a recovery lab read the data off with a new head assembly.
4. Most SSD failures are controller failures. The flash chips are fine and completely unreadable, because only that controller knew the map.

Where this comparison breaks: plenty of hard drives also die instantly, from a seized motor or a snapped head, and plenty of SSDs fail gracefully by dropping into a read-only mode that lets you copy everything off. These are tendencies, not laws.

■ 12.7.3 PLAIN – a worked example

1. Is a consumer SSD's endurance rating actually a problem for you? Let us find out with arithmetic.
2. A 2 TB Samsung 990 Pro is rated at 1,200 TBW over a five-year warranty.
3. 1,200 TB divided by 5 years is 240 TB per year.
4. 240 TB divided by 365 days is about 658 GB per day, every single day, for five years.
5. Typical desktop use is 10 to 40 GB per day. At 30 GB per day you would need 1,200,000 divided by 30, that is 40,000 days, about 109 years.
6. So for ordinary use, endurance is not the thing that will kill your drive. Controller failure, firmware bugs and power events are far more likely.

7. Now the other case. A database server writing 2 TB per day reaches 1,200 TBW in 600 days, under two years. This is exactly why enterprise drives are rated in drive writes per day instead.
8. Sanity check the rating from first principles. TBW is roughly capacity times P/E cycles divided by write amplification. 2 TB times 1,500 cycles divided by 2.5 gives 1,200 TB. The published figure is consistent with TLC flash of around 1,500 cycles at a write amplification of about 2.5.

■ 12.7.4 PLAIN – what is really happening inside

1. Wear is not uniform damage across a chip. It is charge getting trapped inside the tunnel oxide, which shifts and blurs the voltage levels.
2. As levels blur, the error correction has to work harder. Reads take longer because the controller must retry at different reference voltages.
3. A worn drive therefore gets slower before it fails, especially on reads of old data.
4. **Read disturb:** reading one page requires putting a raised voltage on every other word line in the string. That nudges charge into neighbouring cells. After roughly one hundred thousand to a million reads of one block, the controller must copy the block elsewhere and erase it.
5. **Retention:** a powered drive can refresh weak blocks in the background. An unpowered drive cannot. The clock is running on every cell.
6. Retention halves roughly for every 10 degrees Celsius of extra storage temperature, and falls sharply once the drive is near its rated cycle count.
7. So a brand new SSD in a cool drawer will hold data for many years. A heavily worn SSD in a hot loft may not survive one summer.
8. If you use SSDs for cold archives, power them on and read them through once or twice a year. Or use disks or tape, which do not have this failure mode.

■ 12.7.5 TECHNICAL – the engineer’s version

1. JEDEC standard JESD218 defines the endurance and retention test method. Client SSDs must retain data for 1 year at 30 degrees Celsius after reaching rated endurance; enterprise SSDs must retain for 3 months at 40 degrees. Both are end-of-life figures, not new-drive figures.
2. Client drives are rated in TBW; enterprise drives in DWPD, drive writes per day, over a stated warranty term. Conversion: TBW equals DWPD times capacity in TB times 365 times years.
3. A 1 DWPD read-intensive 3.84 TB drive over 5 years is 1 times 3.84 times 365 times 5, that is 7,008 TBW. A 3 DWPD mixed-use part of the same size is 21,024 TBW.
4. SMART attributes worth watching. On SATA SSDs: 5 Reallocated Sector Count, 177 Wear Leveling Count, 231 SSD Life Left, 241 Total LBAs Written. On NVMe: `percentage_used`, `available_spare`, `available_spare_threshold`, `media_errors`, `unsafe_shutdowns`. `percentage_used` can legitimately exceed 100, which means the drive has passed its rated endurance, not that it failed.
5. Backblaze’s published quarterly drive statistics report hard drive annualized failure rates generally in the range of about 1.3 to 1.6 percent across their fleet, with SSD boot drives comparable at similar ages. The often-repeated claim that SSDs are simply more reliable than HDDs is not well supported by that data; the failure modes differ more than the rates do.
6. Google and Facebook field studies published in 2016 found that SSD uncorrectable error rates correlate more strongly with age and with the number of blocks already retired than with raw bytes written, and that early life has a distinct higher-error period. Experts disagree on how far those findings, drawn from older MLC fleets, transfer to modern TLC and QLC parts.

Metric	HDD 7,200 RPM	SATA SSD	NVMe Gen5 SSD
Read latency	5 to 15 ms	about 100 us	20 to 80 us
4K random read IOPS	75 to 100	90k to 100k	1.0M to 2.5M
Sequential read	150 to 300 MB/s	500 to 560 MB/s	12 to 15 GB/s
Price per TB, 2026	21 to 32 USD	from 96 USD	from 105 USD

7. Failure mode summary. HDD: gradual, dominated by media defects, reallocated sectors and head or motor wear, usually preceded by SMART warnings and often partially recoverable. SSD: bi-modal, either slow wear-out ending in a read-only state, or abrupt controller or firmware failure with total loss and essentially no consumer recovery path.
8. Firmware bugs are a real and repeated cause of sudden SSD loss. Several vendors have shipped defects that brick drives at a specific power-on-hours count, notably a family of HPE SAS SSDs in 2019 and 2020 that failed at 32,768 and again at 40,000 hours. Check for firmware updates on drives you depend on.

■ 12.7.6 WORDS – remember these

1. TBW — how much you may write before the warranty ends — terabytes written, the client endurance rating.
2. DDPD — how many times you may rewrite the whole drive daily — drive writes per day over a stated warranty term.
3. Read disturb — reading damages the neighbours a little — cumulative disturbance requiring periodic block refresh.
4. Retention — how long it remembers unplugged — data retention time, specified by JESD218 at end of rated life and temperature dependent.
5. Available spare — how many replacement blocks are left — the NVMe SMART field that actually predicts end of life.
6. Bimodal failure — either slow decline or instant death — the two distinct SSD failure populations, wear-out and controller fault.

12.8 How storage connects: interfaces and buses

■ 12.8.1 PLAIN – in simple words

1. A drive is useless until it is wired to the computer. That wiring is an **interface**: a cable, a connector and a language.
2. The language matters as much as the cable. It sets how many requests can be in flight at once.
3. Old drives were slow, so the language only allowed one request at a time. Nobody minded, because the disk was the bottleneck anyway.
4. Then SSDs arrived and could serve thousands of requests at once, and the polite one-at-a-time language became the thing holding everything back.
5. So the industry wrote a new language, **NVMe**, designed for a device with no moving parts, and connected drives straight to the processor's own fast bus.
6. That bus is **PCI Express**, usually shortened to PCIe. It is the same bus a graphics card uses.
7. Everything else on this page is a variation on those two ideas: how many wires, and how good the language is.

■ 12.8.2 PLAIN – a picture in your head

1. Picture a warehouse counter with one clerk who takes one order, fetches it, comes back, and only then takes the next order. That is the old ATA and SATA model with a spinning disk behind it.
2. Now put a thousand robots in the warehouse. The clerk is still taking one order at a time, so 999 robots stand idle.
3. NVMe replaces the clerk with thousands of order slots that customers fill in themselves, and the robots pick work out of the slots as they become free.
4. Each processor core gets its own order slot, so cores never queue behind each other.

Where this comparison breaks: real NVMe queues live in the host's memory and the drive fetches from them by direct memory access, so there is no clerk at all. And the drive can complete work out of order, which the picture of a queue does not capture.

■ 12.8.3 PLAIN – a worked example

1. A PCIe **lane** is one pair of wires each way. Drives usually use four lanes, written x4.
2. Each generation roughly doubles the speed per lane.

Generation	Year	Per lane	x4 total
PCIe 3.0	2010	0.985 GB/s	3.9 GB/s
PCIe 4.0	2017	1.97 GB/s	7.9 GB/s
PCIe 5.0	2019	3.94 GB/s	15.8 GB/s
PCIe 6.0	2022	7.56 GB/s	30.2 GB/s

3. Compare that with SATA 3.0 at 6 gigabits per second. After encoding overhead that is about 550 to 560 MB/s in practice.
4. So a PCIe 5.0 x4 slot carries roughly 28 times what a SATA cable carries.
5. Yet a fast NVMe drive reaching 14 GB/s is only about 90 percent of what its Gen5 x4 link allows. The drive, not the bus, is now the limit again.

■ 12.8.4 PLAIN – what is really happening inside

1. The oldest common interface was **IDE**, later renamed **PATA**: a wide 40-pin ribbon cable carrying 16 bits side by side, with two drives per cable configured as master and slave by jumper.
2. Parallel wiring failed as speeds rose, because 16 signals never arrive at exactly the same instant. That mismatch is called skew.
3. **SATA** solved it by sending one bit stream very fast down a single differential pair each way, with one drive per cable and no jumpers.
4. **SAS** is the enterprise cousin: same cabling idea, richer command set from SCSI, two ports per drive so two servers can reach it, and expanders so one controller can address thousands of drives.
5. NVMe threw away the disk-shaped assumptions entirely. There is no cable protocol to translate, no single command queue, and no controller chip sitting between drive and CPU.
6. Physically, an NVMe drive is usually an **M.2** stick: 22 mm wide, most often 80 mm long, hence the name 2280.
7. The notch cut into the M.2 edge connector is the **key**, and it prevents you fitting a card the socket cannot drive. An M-key socket carries PCIe x4; a B-key socket carries at most PCIe x2 or SATA.
8. In phones and cheap tablets the flash is soldered down and uses **eMMC** or **UFS** instead, which are simpler and lower powered.

■ 12.8.5 TECHNICAL – the engineer’s version

1. IDE was created by Western Digital with Compaq in 1986 and standardized as ATA-1 in 1994. Its fastest mode, Ultra DMA 133, reached 133 MB/s and required an 80-conductor cable with ground wires interleaved to control crosstalk.
2. SATA revision 1.0 arrived in 2003 at 1.5 Gbit/s, 2.0 in 2004 at 3 Gbit/s, and 3.0 in 2009 at 6 Gbit/s. All three use 8b/10b encoding, so 6 Gbit/s yields 600 MB/s of raw payload and about 550 MB/s after protocol overhead. SATA has had no speed increase since 2009; it is a finished standard.
3. AHCI, the standard SATA host controller interface, provides one command queue of at most 32 commands, and each command costs several uncached register accesses. NVMe provides up to 65,535 I/O queues of up to 65,536 commands each, with doorbell registers written at most twice per command.
4. NVMe 1.0 was published on 1 March 2011; the current revision is NVMe 2.3, published in August 2025. The specification family split into base, command set and transport documents at NVMe 2.0 in 2021.
5. PCIe encoding changed twice. Generations 1 and 2 used 8b/10b, giving a 20 percent tax. Generations 3 to 5 use 128b/130b, about 1.5 percent. PCIe 6.0 moved to PAM4 signalling with fixed-size FLIT framing and forward error correction. PCI-SIG released the PCIe 7.0 specification in 2025 at 128 GT/s per lane, roughly 15.1 GB/s per lane; products are not yet shipping.
6. M.2 lengths in use are 2230, 2242, 2260, 2280 and 22110, the last two digits or three being the length in millimetres. Handheld consoles and thin laptops often take only 2230.
7. Larger form factors: U.2, defined by SFF-8639, puts NVMe in a 2.5-inch hot-swappable drive. U.3, defined by SFF-TA-1001, accepts SAS, SATA and NVMe in one bay. Datacentres increasingly use EDSFF rulers such as E1.S and E3.S, which cool better and pack more capacity per rack unit.

Interface	Raw rate	Practical payload
SATA 3.0	6 Gbit/s	about 550 MB/s
SAS-4	22.5 Gbit/s	about 2.2 GB/s
USB 3.2 Gen 2	10 Gbit/s	about 1.0 GB/s
USB4 / Thunderbolt 4	40 Gbit/s	3.0 to 3.8 GB/s

8. Thunderbolt 5 and USB4 version 2 raise the link to 80 Gbit/s symmetric, with Thunderbolt 5 able to run 120 Gbit/s in one direction for displays. External NVMe enclosures should use the UASP protocol, not the older BOT mass-storage protocol, or they lose command queuing and most of their speed.
9. Mobile storage: eMMC 5.1 uses an 8-bit parallel bus and reaches roughly 250 to 400 MB/s. UFS uses a serial MIPI M-PHY link with full command queuing. UFS 4.0, published in 2022, reaches about 4,200 MB/s sequential read, and UFS 4.1 followed in 2025. eMMC is effectively obsolete in flagship phones and survives in low-cost devices and embedded boards.
10. Useful commands: `lspci -vv` to see link speed and width negotiated, `nvme list` and `nvme id-ctrl`, `smartctl -a`, and on macOS `system_profiler SPNVMeDataType`.

```
# What PCIe generation and width did the SSD actually get?
sudo lspci -vv -s $(lspci | grep -i nvme | cut -d' ' -f1) \
  | grep -E 'LnkCap|LnkSta'
# LnkCap says what it can do, LnkSta says what it negotiated.
```

■ 12.8.6 WORDS – remember these

1. PATA / IDE — the old wide ribbon cable — parallel ATA, 16-bit, two devices per channel, up to 133 MB/s.
2. SATA — one fast serial pair per direction — serial ATA, 6 Gbit/s since 2009, driven through AHCI with a 32-command queue.
3. SAS — the enterprise serial interface — Serial Attached SCSI, dual ported, expander addressable, 22.5 Gbit/s at SAS-4.
4. NVMe — the language written for flash — NVM Express over PCIe, up to 65,535 queues of 65,536 commands.
5. PCIe lane — one wire pair each way — a differential lane, doubling in rate roughly every generation.
6. M.2 — the small stick shaped drive — 22 mm wide card, keyed to advertise which signals its socket carries.
7. UFS — the phone's fast storage — Universal Flash Storage, serial and queued, about 4,200 MB/s at UFS 4.0.

12.9 What a filesystem is

■ 12.9.1 PLAIN – in simple words

1. A drive does not know what a file is. It only offers a very long numbered list of fixed-size blocks.
2. Humans want named things, in folders, that can grow and shrink.
3. A **filesystem** is the software that turns one into the other.
4. It has to answer four questions at all times: which blocks are free, which blocks belong to which file, what each file is called, and what else we know about it.
5. The information about a file, as opposed to the file's contents, is called **metadata**: size, owner, permissions, dates.
6. A folder is not a special kind of object. It is just a file whose contents are a list of names and pointers.
7. In the Unix family, the record holding a file's metadata and the addresses of its data blocks is called an **inode**.
8. The surprise for most people: the file's name is not in the inode. The name lives in the folder. The inode does not know what it is called.

■ 12.9.2 PLAIN – a picture in your head

1. Think of a hotel. The rooms are numbered blocks on the disk.
2. The inode is the guest's registration card. It records who they are, when they arrived, and which rooms they occupy.
3. The directory is the list at reception mapping the name "Patel" to registration card number 4,192.
4. Two names can point at the same card. That is a **hard link**: one guest, two names at reception, one bill.
5. Deleting a name only crosses it off the reception list. The card is only destroyed when the last name pointing to it is gone.
6. That is exactly how Unix deletion works, and why the system call is called `unlink` rather than `delete`.

Where this comparison breaks: a hotel guest knows their own name, but an inode genuinely does not contain its filename. And renaming a real guest requires nothing but a pen at reception, which is precisely why renaming a file is fast no matter how large the file is.

■ 12.9.3 PLAIN – a worked example

1. Here is what happens on an ext4 filesystem when you run four commands.

```
touch notes.txt
1. allocate a free inode, say number 4192
2. write inode: mode, owner, size 0, timestamps
3. add entry ("notes.txt" -> 4192) to the directory file
4. mark inode 4192 used in the inode bitmap

write 10 KB into it
1. allocate 3 data blocks of 4 KB from the block bitmap
2. write the bytes into those blocks
3. record the extent (start block, length 3) in the inode
4. set inode size = 10240, update mtime

mv notes.txt diary.txt
1. add ("diary.txt" -> 4192) to the directory
2. remove ("notes.txt" -> 4192)
3. no data block is touched at all

rm diary.txt
1. remove the directory entry
2. decrement the inode link count, now 0
3. free the 3 data blocks in the bitmap
4. free inode 4192 in the inode bitmap
5. the 10 KB of bytes are still physically there
```

2. Notice step 5 of the delete. Nothing overwrote your data. Only the bookkeeping changed. That is the whole basis of file recovery.
3. Notice also that renaming a 50 GB video is instant, because a rename edits a directory entry and never touches the data.

■ 12.9.4 PLAIN – what is really happening inside

1. Blocks on disk are 512 or 4,096 bytes, but filesystems usually group them into a larger unit, called a **block** in Unix and a **cluster** in the Windows world. 4 KB is the common default.
2. A file always occupies a whole number of clusters. A 1-byte file consumes a full 4 KB. The wasted remainder is **internal fragmentation**, or slack.
3. Free space must be tracked. Three classic methods:
 1. A **bitmap**, one bit per block. Simple and compact. Used by ext4 and NTFS.
 2. A **linked chain**, where each entry names the next. Used by FAT.
 3. **Extents**, recording start and length rather than every block, which is far more compact for big files. Used by ext4, NTFS, XFS, APFS.
4. Metadata typically includes file type, permission bits, owner and group IDs, size, link count, and three or four timestamps.
5. A directory is a file whose contents are (name, inode number) pairs. Modern filesystems index those pairs with a hash or a B-tree so that a directory with a million files is still fast to search.
6. Path lookup is therefore recursive. To open /home/ana/a.txt, the kernel reads the root directory to find home, reads that to find ana, reads that to find a.txt, then reads that inode. Each step is a real disk lookup, cached aggressively in memory.
7. Mounting attaches the root of one filesystem onto a directory of another, so the user sees one seamless tree.

■ 12.9.5 TECHNICAL – the engineer’s version

1. A classic Unix inode is 128 or 256 bytes and holds mode, uid, gid, size, link count, timestamps and either direct block pointers or an extent tree root. ext4 uses 256-byte inodes by default, which leaves room for nanosecond timestamps and inline extended attributes.
2. The original Unix File System used 12 direct pointers plus single, double and triple indirect blocks. With 4 KB blocks that reaches about 4 TB per file but costs up to four extra reads for a distant block. Extents replaced it because one extent record can describe up to 128 MiB in ext4.
3. Inodes are allocated at format time in ext2, ext3 and ext4. mke2fs defaults to roughly one inode per 16 KB of space, so a filesystem can run out of inodes while showing free space. `df -i` shows this, and it is a classic incident on mail and cache servers full of tiny files.
4. Filesystem block size is fixed at format time in most designs. XFS and ext4 are limited to the CPU page size, so 4 KiB on x86-64; APFS uses 4 KiB blocks; ZFS uses a variable record size up to 1 MiB.

```
stat notes.txt          # inode number, links, size, blocks, times
df -i /                 # inodes used and free, not just bytes
du -sh --apparent-size . # real bytes, versus du -sh block usage
ls -di /home            # inode number of a directory
debugfs -R "stat <4192>" /dev/sda1 # raw inode dump, ext4
```

5. POSIX guarantees that `rename()` within one filesystem is atomic: an observer sees either the old name or the new one, never neither and never both. This is the foundation of the write-to-temporary-then-rename idiom used by nearly every well-written program that updates a file safely.
6. Hard links share an inode and cannot cross filesystems or, in most systems, point at directories. Symbolic links are separate inodes containing a path string, resolved at every use, and may dangle.

■ 12.9.6 WORDS – remember these

1. Filesystem — the software that turns blocks into files — the on-disk format plus the driver implementing it.
2. Block or cluster — the allocation unit — the smallest number of sectors the filesystem will hand to a file, usually 4 KiB.
3. Metadata — everything about a file except its contents — mode, ownership, size, timestamps, link count, block map.
4. Inode — the file’s record card — a fixed-size structure holding metadata and the map to data blocks, identified by number, not by name.
5. Directory — a file listing names — a mapping from names to inode numbers, usually indexed by hash or B-tree.
6. Extent — one run of consecutive blocks — a (start, length) record replacing per-block pointers.
7. Hard link — a second name for the same file — an additional directory entry referencing the same inode, raising its link count.
8. Slack — the wasted tail of the last cluster — internal fragmentation from rounding file size up to a whole allocation unit.

12.10 Real filesystems compared

■ 12.10.1 PLAIN – in simple words

1. There is no single filesystem. Every operating system grew its own, and they all still exist because removable drives have to be readable everywhere.
2. **FAT** is the oldest survivor. Simple, small, understood by every device from cameras to car stereos, and very limited.
3. **exFAT** is FAT modernized to remove the size limits, and is now the normal choice for a big memory card or a shared USB stick.
4. **NTFS** is the Windows filesystem: journaled, permissioned, and full of extra features.
5. **ext4** is the standard Linux filesystem: reliable, fast, unexciting, and used on most servers on the internet.
6. **APFS** is Apple's, built for flash, and very good at making instant copies.
7. **Btrfs** and **ZFS** are the careful ones. They check every block against a stored fingerprint and can repair damage by themselves if you gave them a spare copy.
8. The rule of thumb: use your operating system's native filesystem for internal drives, and exFAT only for drives that must travel between systems.

■ 12.10.2 PLAIN – a picture in your head

1. Think of a photocopier that is clever enough to be lazy.
2. You ask for a copy of a 500-page report. Instead of copying anything, it writes a note saying "this copy is the same as the original".
3. The copy appears instantly and takes no shelf space.
4. Later you edit page 12 of the copy. Only then does it actually duplicate page 12, and only page 12. The other 499 pages stay shared.
5. That is **copy-on-write**, and it is what APFS, ZFS and Btrfs do. It is why duplicating a 50 GB folder on a Mac can finish instantly.
6. A **snapshot** is the same trick applied to a whole filesystem: freeze the current state, and only start copying blocks when something changes them.

Where this comparison breaks: the photocopier note is a single record, while a real copy-on-write filesystem must keep a reference count on every shared block and free a block only when the last reference goes. And free space reporting becomes confusing: deleting a file may free nothing, because a snapshot still references it.

■ 12.10.3 PLAIN – a worked example

1. Why will a 4 GB video not copy to a FAT32 memory stick, even with 200 GB free?
2. In a FAT32 directory entry, the file size is stored in a 32-bit unsigned field. That is a hard part of the on-disk format, not a bug.
3. The largest number that fits in 32 bits is 2 to the power 32, minus 1, which is 4,294,967,295.
4. So the largest possible FAT32 file is 4,294,967,295 bytes, one byte short of 4 GiB.
5. There is no workaround inside FAT32. The field is not big enough to write the number down.
6. The fixes are: reformat as exFAT or NTFS, or split the file into parts.
7. The related limit people meet is that the Windows graphical format tool refuses to make a FAT32 volume larger than 32 GB. That is a policy in the tool, not a limit of the format, which reaches 2 TiB with 512-byte sectors.

■ 12.10.4 PLAIN – what is really happening inside

1. FAT keeps one table, the File Allocation Table, with one entry per cluster. Each entry either says “free”, “bad”, “end of file”, or gives the number of the next cluster in this file. A file is a chain you follow.
2. That is why FAT is easy to implement and slow for large files: to reach the middle of a file you must walk the chain from the start.
3. NTFS keeps everything, including its own bookkeeping, in one enormous file called the **master file table**, or MFT. Each file gets a record, normally 1 KB.
4. A small file’s contents live inside its own MFT record, so no data block is allocated at all. This is why NTFS handles thousands of tiny files well.
5. ext4 splits the volume into block groups, each with its own inode table and bitmaps, so related metadata and data stay physically close.
6. APFS, ZFS and Btrfs never overwrite a live block. A change writes new blocks, then new parent blocks pointing at them, up to the root, and finally one atomic pointer swap at the very top.
7. That is why they can offer instant snapshots and why they are always consistent after a crash. The old tree is still intact until the moment the root changes.
8. ZFS and Btrfs additionally store a checksum of every block in that block’s parent. On read, the checksum is verified. If it fails and a redundant copy exists, the good copy is returned and the bad one is rewritten silently. That is **self-healing**, and it catches errors a drive never reports.

■ 12.10.5 TECHNICAL – the engineer’s version

1. Dates and origins. FAT12 came with Microsoft’s 86-DOS work around 1980; FAT16 with PC DOS 3.0 in 1984; FAT32 with Windows 95 OSR2 in August 1996. exFAT appeared in Windows CE 6.0 in late 2006; Microsoft published the specification on 28 August 2019 and Linux gained a native driver in kernel 5.4 in November 2019.
2. NTFS shipped with Windows NT 3.1 in 1993. ext2 was written by Remy Card in 1993, ext3 added journaling in 2001 through work by Stephen Tweedie, and ext4 was marked stable in Linux 2.6.28 in December 2008. ZFS came from Sun Microsystems, principally Jeff Bonwick and Matt Ahrens, and shipped in OpenSolaris in 2005. Btrfs was started by Chris Mason at Oracle in 2007, with the on-disk format declared stable in Linux 3.10 in 2013. APFS was announced at Apple’s WWDC in June 2016, shipped in iOS 10.3 in March 2017 and in macOS High Sierra in September 2017.

Filesystem	Max file	Max volume
FAT16	2 GiB	2 GiB at 32 KiB clusters
FAT32	4 GiB minus 1 byte	2 TiB at 512 B sectors
exFAT	16 EiB (spec)	128 PiB (spec)
NTFS	8 PiB minus 2 MiB	8 PiB at 2 MiB clusters
ext4	16 TiB at 4 KiB blocks	1 EiB
APFS	8 EiB	8 EiB
Btrfs	16 EiB	16 EiB
ZFS	16 EiB	256 ZiB per pool

Filesystem	Crash strategy	Main platform
FAT32 / exFAT	none (exFAT opt. TexFAT)	removable media
NTFS	metadata journal	Windows
ext4	journal, ordered mode	Linux

Filesystem	Crash strategy	Main platform
APFS	copy-on-write	macOS, iOS
Btrfs / ZFS	copy-on-write plus checksums	Linux, BSD, Solaris

3. The NTFS 8 PiB figure applies from Windows 10 version 1709 and Windows Server 2019; older releases capped volumes at 256 TiB with 64 KiB clusters. The theoretical on-disk limit is 16 EiB. Version and edition matter here, so quote the Windows release with any NTFS limit.
4. ext4's 16 TiB per-file limit follows from 32-bit extent block counts with 4 KiB blocks. The 1 EiB volume limit follows from 48-bit block addressing.
5. APFS space sharing means several volumes in one container draw from a shared free pool, so `df` shows the same free figure several times. A **clone** is a file-level copy-on-write duplicate created by the `clonefile` system call and used by `cp -c` and the Finder. A **snapshot** is the container-level equivalent and is what Time Machine local backups use.
6. macOS still cannot write to NTFS without third-party software, and Windows cannot read APFS or ext4 without third-party software. exFAT is the only format that all three major systems read and write out of the box, which is why cameras and large memory cards use it. SD cards above 32 GB, the SDXC class, specify exFAT in the SD standard.
7. Btrfs RAID 5 and RAID 6 profiles are still documented by the project as having unresolved issues and are not recommended for data you care about, as of 2026. Its RAID 0, 1 and 10 profiles are considered sound. Experts disagree on Btrfs stability generally; it is the default in some enterprise Linux distributions and avoided in others.

■ 12.10.6 WORDS – remember these

1. FAT — the simple old chain-based format — File Allocation Table, one entry per cluster forming per-file linked lists.
2. exFAT — FAT with the limits removed — Microsoft's format for SDXC and large removable media, specification published 2019.
3. MFT — the giant index inside NTFS — Master File Table, one 1 KB record per file, small files stored resident.
4. Copy-on-write — never change a live block — write new blocks and swap the root pointer atomically.
5. Snapshot — a frozen view of the whole filesystem — a retained root pointer with reference counts preventing block reuse.
6. Clone — an instant duplicate that shares blocks — a file-level copy-on-write copy, `clonefile` on APFS, `relink` on Btrfs and XFS.
7. Self-healing — the filesystem repairs bad blocks itself — checksum verification on read plus automatic repair from a redundant copy.

12.11 Journaling, consistency and crashes

■ 12.11.1 PLAIN – in simple words

1. Almost nothing a filesystem does is a single write. Most operations are three or four writes that must all happen or none.
2. Appending to a file means: mark blocks used, write the data, update the file's record. A crash between any two leaves a half-finished mess.
3. If the blocks are marked used but the file record never learns about them, the space is lost forever.

4. If the file record claims blocks that were never marked used, the same blocks may later be handed to a second file. Now two files share bytes and both are ruined.
5. A **journal** fixes this by writing down what you are about to do, before you do it.
6. After a crash the system reads the journal. Anything fully written down is redone. Anything incomplete is thrown away.
7. Either way the filesystem is consistent afterwards, and it takes seconds rather than hours to check.
8. Consistent does not mean nothing was lost. It means nothing is contradictory. The last few seconds of your work can still be gone.

■ 12.11.2 PLAIN – a picture in your head

1. Think of a bank teller moving money between two accounts.
2. Take 100 out of account A. Put 100 into account B. If the power fails in between, 100 has vanished from the world.
3. So before touching either account, the teller writes in a bound ledger: “about to move 100 from A to B”, and only then starts.
4. If the lights go out, the next morning someone reads the ledger, sees an entry, and either finishes the move or undoes it.
5. The ledger is the journal. It is small, written in order, and never has to be searched.
6. Copy-on-write filesystems solve the same problem differently. They prepare a complete new version of everything off to one side, then change one single pointer to make it live.

Where this comparison breaks: a real journal does not have to hold the file data, only the bookkeeping, so a crash can leave a consistent filesystem containing a file full of rubbish. And the ledger only helps if the pen truly touched the paper before the lights went out, which is exactly the problem covered in the technical block below.

■ 12.11.3 PLAIN – a worked example

1. Here is how a journaled metadata update runs, in order.

```

1. write journal: "intend: alloc blk 5000-5002, inode 4192,
                  size 10240"
2. FLUSH -> force the drive to commit that to media
3. write journal commit record
4. FLUSH -> the intent is now durable
5. write the real bitmap, inode and directory blocks
6. mark the journal entry complete

Crash after step 4 -> on mount, replay steps 5 and 6.
Crash before step 3 -> on mount, discard, nothing happened.
Crash between 5 and 6 -> replay is harmless, it is idempotent.
```

2. Notice steps 2 and 4. Without them the drive's own write cache may reorder everything and the whole scheme collapses.
3. Notice also that this costs two extra flushes and one extra write per operation. Journaling is not free; it buys recovery time with throughput.

■ 12.11.4 PLAIN – what is really happening inside

1. Drives lie about writes for speed. A drive with a volatile write cache says “done” when the bytes reach its internal memory, not the platter or the flash.
2. That is safe for performance and lethal for ordering. A **flush** command forces the drive to actually commit everything it has acknowledged.

3. The kernel also has a cache, the page cache. A normal `write()` only copies bytes into memory and returns. The bytes may reach the drive seconds later.
4. `fsync()` is the system call that says “do not return until this file’s data and metadata are truly on stable storage”, and it triggers the drive flush.
5. Databases live or die on this. Every committed transaction must be provably on disk before the commit is reported to the client, or the database has lied about durability.
6. That is why database benchmarks care about flush latency more than about throughput, and why enterprise SSDs with capacitors are worth their price: they can honour a flush instantly because their cache is protected.
7. Journals normally protect metadata only. Your file’s bookkeeping survives; the file’s contents may be a mixture of old and new.
8. Copy-on-write designs need no journal for consistency, because the old tree remains valid until the atomic root swap. They still use a small log for speed when handling many tiny synchronous writes.

■ 12.11.5 TECHNICAL – the engineer’s version

1. ext4 offers three journal modes, set with the `data=` mount option: `journal` writes file data through the journal too, which is safest and slowest; `ordered` is the default and forces data blocks out before the metadata commit that references them; `writeback` orders nothing and can expose stale block contents after a crash.
2. NTFS journals metadata in `$LogFile` using write-ahead logging with undo and redo records. It never journals file data. XFS and JFS are also metadata-only journals.
3. Barriers are implemented as `FLUSH CACHE` in ATA, `SYNCHRONIZE CACHE` in SCSI and `Flush` in NVMe, or per-command with Force Unit Access. Mounting with `nobarrier` on a drive without power-loss protection is a known way to corrupt a filesystem, and should never be done outside a battery-backed array.
4. On macOS, `fsync()` deliberately does not force the drive to flush its cache. Programs that need real durability must call `fcntl(fd, F_FULLFSYNC, 0)`. This is a documented Apple-specific behaviour and a classic portability trap.
5. `fdatasync()` skips metadata that does not affect retrieval, such as `mtime`, which is measurably faster for database write-ahead logs.
6. Filesystem check tools. On a journaled filesystem a normal mount only replays the journal, taking under a second. A full structural check is a different operation: `fsck.ext4 -f` walks every inode, rebuilds the block and inode bitmaps, verifies link counts, and moves unreferenced-but-allocated inodes into `lost+found`. On a 20 TB ext4 filesystem with many files this can take hours and requires substantial memory. `xfs_repair` and `chkdsk /f` are the equivalents. ZFS deliberately has no `fsck`; it has `zpool scrub`, which verifies checksums on a live pool.

```
sudo tune2fs -l /dev/sda1 | grep -i 'features\|state'
sudo dumpe2fs -h /dev/sda1          # journal size and location
sudo zpool status ; sudo zpool scrub tank
sudo btrfs scrub start /mnt/data
```

7. The honest version: journaling protects filesystem structure, not your data. A power cut during a database write on a journaled filesystem can still lose committed-looking transactions if the application did not call `fsync` and the drive had no power-loss protection. Three separate parties must each do their job: the application, the kernel, and the drive.

■ 12.11.6 WORDS – remember these

1. Journal — write down the plan before doing it — a write-ahead log of metadata changes, replayed or discarded at mount.
2. Atomicity — all of it or none of it — the guarantee that a multi-step update is never observed half-applied.
3. Write barrier — force the order — a cache flush or FUA operation preventing the device from reordering across a point.
4. fsync — really put it on the disk now — the system call forcing page cache and device cache to stable storage.
5. Ordered mode — data first, then the pointer to it — ext4's default, preventing a new inode from exposing stale block contents.
6. fsck — the full structural rebuild — offline consistency check reconstructing bitmaps and link counts, slow on large volumes.
7. Scrub — read everything and verify it — background checksum verification and repair in ZFS and Btrfs.

12.12 Partitions, formatting and booting

■ 12.12.1 PLAIN – in simple words

1. One physical drive is usually cut into several independent regions called **partitions**.
2. The list of partitions is written in a small table at the very start of the drive.
3. There are two such table formats in use. The old one is **MBR**, the modern one is **GPT**.
4. MBR fits in one 512-byte sector, allows only four partitions, and cannot describe anything past 2 TB.
5. GPT allows a hundred and twenty-eight partitions, describes drives far larger than any that exist, and keeps a spare copy at the end of the drive in case the first is damaged.
6. **Formatting** a partition means writing an empty filesystem onto it: fresh bookkeeping structures saying “nothing is here”.
7. A quick format writes only those structures. It never touches your old data, which is simply no longer referenced.
8. That is why a quick format is not a way to destroy anything, and why recovery software often gets almost everything back from a quick-formatted drive.
9. **Mounting** is attaching a filesystem so you can see it. Windows gives it a letter; macOS and Linux graft it onto a folder in one big tree.

■ 12.12.2 PLAIN – a picture in your head

1. Think of a drive as a large empty warehouse and partitions as internal walls.
2. The partition table is the sign by the front door listing each room and where it starts and ends.
3. Formatting a room means putting up empty shelves and a fresh index card at the door. It does not mean sweeping the floor.
4. Quick formatting replaces the index card only. Everything on the shelves is still there, just unlisted.
5. Mounting is opening a door from the corridor into that room, so people can walk in.

Where this comparison breaks: the drive does not know or care about the walls. Partitions are purely a convention agreed between software, written in a table the drive itself never reads. You can wipe the table and every byte of the data is still physically present.

■ 12.12.3 PLAIN – a worked example

1. Why exactly does MBR stop at 2 TB?
2. An MBR partition entry records the starting sector and the length as 32-bit numbers of sectors.
3. The largest count is 2 to the power 32, minus 1, that is 4,294,967,295 sectors.
4. With traditional 512-byte sectors: 4,294,967,295 times 512 equals about 2.199 times 10 to the 12 bytes, which is 2 TiB.
5. On a 4Kn drive with 4,096-byte sectors the same arithmetic gives 16 TiB, so the limit is a property of sector size, not of storage in general.
6. GPT stores those figures as 64-bit sector numbers. 2 to the power 64 sectors of 512 bytes is about 8 zebibytes, which no drive will reach this century.

```
An MBR: one 512-byte sector at LBA 0
+-----+
| bootstrap code                446 bytes|
| partition entry 1             16 bytes|
| partition entry 2             16 bytes|
| partition entry 3             16 bytes|
| partition entry 4             16 bytes|
| signature 0x55 0xAA           2 bytes|
+-----+
GPT: protective MBR at LBA 0, header at LBA 1,
128 entries of 128 bytes at LBA 2..33, backup at the end.
```

■ 12.12.4 PLAIN – what is really happening inside

1. When an old BIOS machine starts, it loads the first sector of the boot drive into memory and jumps into it. That is the 446 bytes of bootstrap code in the MBR. Those 446 bytes then load a bigger loader from somewhere else.
2. A UEFI machine does something far more sensible. It reads the partition table, finds a partition marked as the **EFI System Partition**, which is formatted FAT32, and runs a normal file from it, such as `bootx64.efi`.
3. That is why the EFI partition is FAT32 on a modern Mac and PC: the firmware must be able to read it before any operating system exists.
4. Secure Boot adds a signature check on that file against keys stored in firmware.
5. A **logical volume manager** adds a layer between partitions and filesystems. It gathers whole drives into a pool and hands out volumes from that pool, which can be grown, shrunk, moved between drives and snapshotted while in use.
6. Without it, growing a filesystem means the free space must happen to sit directly after it on the same drive. With it, you just add another drive.
7. Mount points differ by system, but the idea is identical: the filesystem's root is grafted somewhere the user can reach.

■ 12.12.5 TECHNICAL – the engineer's version

1. The MBR layout dates from PC DOS 2.0 in 1983. Four primary partition entries only; the extended partition trick chains further logical partitions through a linked list of extended boot records, which is fragile and now obsolete.
2. GPT is defined by the UEFI specification, descended from Intel's EFI work in the late 1990s for Itanium. Its header and entry array are each protected by a CRC32 checksum, and a full backup copy sits in the last sectors of the disk. LBA 0 holds a protective MBR describing one partition of type 0xEE covering the whole disk, so that MBR-only tools do not think the disk is blank and offer to initialize it.

- Partition types are GUIDs in GPT. Common ones: EFI System Partition, Microsoft Basic Data, Linux filesystem, Linux LVM, Apple APFS Container.
- Alignment matters. Partitions should start on a 1 MiB boundary so that filesystem 4 KiB blocks align with 4 KiB physical sectors and with SSD page and erase-block boundaries. Every current partitioner does this by default; tools from before about 2010 started at sector 63 and did not.
- Formatting levels: a **low-level format**, meaning writing servo and sector structures, is a factory operation on modern drives and cannot be performed by a user. A **high-level format** writes filesystem metadata. Windows full format since Vista also writes zeros across the volume and runs a surface scan; Windows quick format does not.
- Volume managers: LVM2 on Linux with physical volumes, volume groups and logical volumes; Storage Spaces on Windows; APFS containers on macOS; and ZFS, which merges volume management and filesystem into one layer deliberately.

System	Where drives appear	Table tool
Linux	/mnt, /media/user/LABEL	lsblk, fdisk, parted
macOS	/Volumes/NAME	diskutil list
Windows	drive letters, or folders	diskpart, Disk Management

```
lsblk -f                # devices, filesystems, mountpoints
sudo parted /dev/sda print # partition table type and entries
sudo blkid              # UUIDs used by /etc/fstab
diskutil list           # macOS, shows APFS containers
```

- Mount by UUID or label, never by device name. Device names such as /dev/sdb are assigned in discovery order and can change between boots, which is a common cause of a machine failing to boot after adding a drive.

■ 12.12.6 WORDS – remember these

- Partition — one region of a drive — a contiguous LBA range described in a partition table.
- MBR — the old 512-byte table — Master Boot Record, four primary entries, 32-bit sector counts, 2 TiB ceiling at 512-byte sectors.
- GPT — the modern table — GUID Partition Table, part of UEFI, CRC-protected, with a backup copy at the end of the disk.
- EFI System Partition — the small FAT32 partition the firmware boots from — holds signed .efi loader files.
- Quick format — new bookkeeping, old data untouched — writing fresh filesystem metadata without erasing block contents.
- Mounting — attaching a filesystem into the visible tree — associating a filesystem with a mount point or drive letter.
- LVM — a layer that makes volumes flexible — logical volume manager pooling physical extents and serving resizable logical volumes.

12.13 RAID and redundancy

■ 12.13.1 PLAIN – in simple words

1. **RAID** means using several drives together so that they behave as one, for speed, or for safety, or both.
2. **RAID 0** splits your data across drives in stripes. Twice the drives, twice the speed, all the capacity, and no safety at all. Lose one drive and you lose everything.
3. **RAID 1** writes the same data to two drives. Half the capacity, and either drive alone can carry on.
4. **RAID 5** spreads data across at least three drives and adds one drive's worth of **parity**, a checksum that lets any single missing drive be rebuilt. You lose one drive of capacity.
5. **RAID 6** does the same with two independent parities, so any two drives may die. You lose two drives of capacity.
6. **RAID 10** mirrors pairs and stripes across the pairs. Half the capacity, fast, and rebuilds quickly.
7. RAID protects against exactly one thing: a drive failing. That is all.
8. It does not protect against deletion, ransomware, corrupted files, a power surge, theft, fire or flood. Every one of those hits all drives at once because they are all in one box.
9. So RAID is about staying online, not about keeping data. It is not a backup.

■ 12.13.2 PLAIN – a picture in your head

1. Imagine four people each holding one page of a four-page document, plus a fifth person holding a page of “checksums”: for every character position, the sum of the four letters treated as numbers.
2. If one of the four is missing, you can work out their page exactly by subtracting the other three from the checksum page.
3. That is RAID 5 parity. It is not a copy of anything; it is the difference that lets you reconstruct whichever one you lost.
4. Now notice the danger. To reconstruct one page you must read every other page completely and correctly. If a second person makes even one mistake while you are reconstructing, the answer is wrong.

Where this comparison breaks: real parity uses exclusive-or rather than addition, and RAID 6 uses a second parity computed over a finite field so the two equations are independent. And the real risk is not a person making a mistake but a drive returning an unreadable sector during the rebuild.

■ 12.13.3 PLAIN – a worked example

1. Eight 20 TB drives, and here is what you get from each layout.

Level	Usable	Survives
RAID 0	160 TB	no drive loss
RAID 1 (4 pairs)	80 TB	1 per pair
RAID 5	140 TB	any 1 drive
RAID 6	120 TB	any 2 drives
RAID 10	80 TB	1 per mirror pair

2. Now rebuild time, which is where RAID 5 loses its reputation.
3. Replacing a 20 TB drive means writing 20 TB onto it. At a generous sustained 150 MB/s that is 20,000,000 MB divided by 150, which is 133,333 seconds.
4. That is 37 hours, and only if the array is otherwise completely idle. On a busy array, two to five days is realistic.
5. During all that time you have zero redundancy and every surviving drive is being read from end to end, which is the hardest work they ever do.

6. Consumer drives are specified for one unrecoverable read error per 10 to the 14 bits, which is one per 12.5 TB read. Reading seven 20 TB drives in full is 140 TB. On the specified figure, hitting at least one error is likely.
7. That arithmetic is why RAID 6 or mirroring is the normal advice above about 4 TB per drive. The honest version: real drives usually beat their specified error rate by a wide margin, so the “RAID 5 is mathematically certain to fail” claim is too strong. Experts disagree on the size of the margin. Both camps agree RAID 6 is the safer choice at these capacities.

■ 12.13.4 PLAIN – what is really happening inside

1. Parity is computed with exclusive-or, written XOR. XOR of a set of bits is 1 if an odd number of them are 1.
2. Its useful property: if you XOR all the data blocks together to make parity, then XOR any subset including parity but missing one block, you get the missing block back.
3. RAID 5 rotates which drive holds parity for each stripe, so no single drive becomes the write bottleneck.
4. Small writes to RAID 5 are expensive. To change one block, the controller must read the old block and old parity, compute the new parity, then write both. One logical write becomes two reads and two writes. This is the RAID 5 write penalty.
5. RAID 6's second parity uses Reed-Solomon arithmetic so that the two equations are independent and any two failures can be solved.
6. The **write hole** is a real hazard on parity RAID without a journal or battery: if power is lost between writing data and writing parity, the stripe is silently inconsistent, and a later rebuild will produce wrong data.
7. Hardware RAID controllers use a battery or capacitor-backed cache to close the write hole. Linux md uses a write-intent bitmap and an optional journal. ZFS avoids it entirely because RAID-Z stripes are copy-on-write and variable width.
8. RAID controllers rebuild from parity but usually cannot tell you which copy is right if the data is merely wrong rather than missing. Filesystems with checksums can, which is the argument for ZFS over classic RAID.

■ 12.13.5 TECHNICAL – the engineer's version

1. The taxonomy comes from the 1988 SIGMOD paper “A Case for Redundant Arrays of Inexpensive Disks (RAID)” by David Patterson, Garth Gibson and Randy Katz at the University of California, Berkeley. The industry later renamed the I to “Independent”.
2. RAID 2, 3 and 4 exist in the paper but are effectively unused: RAID 2 used Hamming codes and bit striping, RAID 3 byte striping with a dedicated parity drive, RAID 4 block striping with a dedicated parity drive. RAID 5's rotated parity removed RAID 4's bottleneck.
3. Nested levels are written as two digits: RAID 10 is a stripe over mirrors, RAID 01 is a mirror over stripes, and RAID 10 tolerates more failure patterns and rebuilds far faster because only one drive's worth of data must be copied.
4. ZFS RAID-Z1, Z2 and Z3 correspond roughly to single, double and triple parity, but with variable-width stripes that eliminate the write hole. A `zpool scrub` verifies every checksum and repairs from redundancy. Rebuilds in ZFS, called *resilvering*, copy only allocated blocks, so a half-empty pool resilvers in roughly half the time. Traditional RAID rebuilds the whole drive regardless.
5. Unrecoverable read error rate is specified as 1 in 10^{14} bits for most consumer SATA drives and 1 in 10^{15} for enterprise SATA, SAS and NL-SAS. 10^{14} bits is 12.5 TB; 10^{15} is 125 TB.
6. Practical guidance widely used in 2026: mirrors or RAID 10 for databases and virtual machines where write latency matters; RAID 6 or RAID-Z2 for bulk archives; RAID 5 only on small fast

drives with a short rebuild window; RAID 0 only for scratch data you can regenerate.

7. The **3-2-1 rule**, popularized by the photographer Peter Krogh in his book “The DAM Book” in 2005: keep at least 3 copies of the data, on at least 2 different kinds of media, with at least 1 copy off site. A common modern extension is 3-2-1-1-0: one copy offline or immutable, and zero errors on a verified restore test.
8. The last part is the part people skip. A backup that has never been restored is a hypothesis, not a backup. Test restores on a schedule.

```
cat /proc/mdstat                # Linux md array state
sudo mdadm --detail /dev/md0
sudo zpool status -v tank        # errors per device, scrub state
sudo smartctl -a /dev/sda | grep -i reallocated
```

■ 12.13.6 WORDS – remember these

1. RAID — several drives behaving as one — Redundant Array of Independent Disks, from the 1988 Berkeley paper.
2. Striping — data spread across drives — RAID 0, dividing data into fixed-size chunks written in rotation.
3. Mirroring — the same data on two drives — RAID 1, full duplication.
4. Parity — the difference that rebuilds a lost drive — XOR for single parity, Reed-Solomon for the second.
5. Rebuild / resilver — reconstructing a replaced drive — a full-array read at maximum stress with no redundancy while it runs.
6. Write hole — data and parity out of step after a power cut — silent stripe inconsistency on parity RAID without a journal or backed cache.
7. URE — an unreadable sector — unrecoverable read error, specified at 1 in 10^{14} bits for consumer drives.
8. 3-2-1 — the backup rule — three copies, two media types, one off site.

12.14 Data recovery, deletion and secure erase

■ 12.14.1 PLAIN – in simple words

1. Deleting a file does not remove the file. It removes the entry that points to the file.
2. The bytes stay exactly where they were until something else happens to be given that space.
3. That is why recovery software works, and why an empty-looking second-hand drive can be full of somebody’s private life.
4. On a hard disk, writing new data over the old data really does destroy it. One pass of zeros is enough on any drive made this century.
5. On an SSD, overwriting does not work, because the drive refuses to write in the same place. Your “overwrite” lands on a fresh page and the original is still sitting in a page you cannot address.
6. So SSDs need a different approach: a command that tells the drive itself to wipe everything, including the parts you cannot see.
7. The best approach of all is to encrypt the drive from the day you buy it. Then wiping it means destroying one key, which takes milliseconds.
8. Encrypted from the start, an unerased drive is already unreadable rubbish to anyone without the key.

■ 12.14.2 PLAIN – a picture in your head

1. Think of a library index card system. Deleting a book means throwing away its card. The book is still on the shelf.
2. Recovery software walks the shelves ignoring the index, looking for anything that looks like a book. That is **file carving**.
3. Now the SSD problem. Imagine a librarian who, whenever you ask to replace a book, refuses to touch the original and instead shelves your replacement in a new spot, then quietly updates the index.
4. You can hand over replacement after replacement and the originals stay on the shelves, in a wing of the building you are not allowed to enter.
5. The only way to clear that wing is to ask the librarian, who alone has the keys, to burn everything.

Where this comparison breaks: file carving only finds file contents, not names, folders or dates, and cannot reassemble a file whose pieces were scattered. It works best on photos and videos, which have clear start and end markers and are usually written in one go.

■ 12.14.3 PLAIN – a worked example

1. A JPEG image always starts with the bytes FF D8 FF and ends with FF D9.
2. Carving software reads the whole raw drive looking for those markers and writes out everything in between as a candidate file.
3. It needs no filesystem at all, which is why it survives a quick format.
4. Here is the difference between the erase options, on a 2 TB NVMe SSD.

Method	Time	Actually destroys data
Delete file	instant	no
Quick format	seconds	no
Overwrite with zeros	~3 min	not reliably, on flash
NVMe format, crypto erase	under 1 s	yes
NVMe sanitize, block erase	minutes	yes

5. The crypto erase is instant because it does not touch the data at all. It destroys the internal key the data was encrypted with, and the remaining bytes become noise.

■ 12.14.4 PLAIN – what is really happening inside

1. Modern SSDs, and many hard drives, already encrypt everything they store, whether you asked for it or not. The key lives inside the drive.
2. That is called a self-encrypting drive. Normally the key is used automatically and you never notice.
3. A cryptographic erase simply replaces that key with a new random one. Every byte on the media instantly becomes undecryptable.
4. **ATA Secure Erase** is a command in the ATA standard telling the drive to erase its entire user area itself, including reallocated sectors that normal writes cannot reach. **Enhanced Secure Erase** additionally covers areas the basic command may skip.
5. NVMe has two: **Format NVM**, which can do a user-data erase or a cryptographic erase, and **Sanitize**, which is stronger and covers over-provisioned areas, caches and any unmapped data.
6. Full-disk encryption puts the key under your control rather than the drive maker's. BitLocker on Windows, FileVault on macOS, LUKS with dm-crypt on Linux, all normally using AES in XTS mode with a 256-bit key.

7. With full-disk encryption in place, disposal is trivial. Forget the passphrase, destroy the key slot, and the drive is a brick to everyone, including you.
8. Degaussing, a strong magnetic pulse, destroys hard disks and tapes permanently, and does absolutely nothing to flash, because flash stores charge and not magnetism.

■ 12.14.5 TECHNICAL – the engineer’s version

1. NIST Special Publication 800-88 Revision 1, “Guidelines for Media Sanitization”, published in 2014, defines three levels: Clear, Purge and Destroy. It states explicitly that for modern magnetic media a single overwrite pass is adequate for Clear, and that cryptographic erase is an acceptable Purge method for self-encrypting media.
2. The 35-pass Gutmann method comes from Peter Gutmann’s 1996 Usenix paper and targeted the specific MFM and RLL encodings of drives of that era. Gutmann himself has written that applying all 35 passes to a modern drive is pointless. Repeating it as current advice is a myth.
3. Commands and tools:

```
# ATA drives: check support, then issue a secure erase
sudo hdparm -I /dev/sda | grep -A6 Security
sudo hdparm --user-master u --security-set-pass p /dev/sda
sudo hdparm --user-master u --security-erase p /dev/sda

# NVMe: crypto erase (ses=2) or user-data erase (ses=1)
sudo nvme format /dev/nvme0n1 --ses=2
sudo nvme sanitize /dev/nvme0n1 --sanact=4    # crypto erase
sudo nvme sanitize-log /dev/nvme0n1          # progress
```

4. Drives are often left in a frozen security state by the firmware, and `hdparm` will report frozen. The usual workaround is a suspend and resume cycle, or hot-plugging the SATA data cable, both of which carry risk. Prefer `nvme sanitize` or vendor tools where available.
5. Recovery tooling: `photorec` and `scalpel` for carving, `testdisk` for rebuilding partition tables, `ddrescue` for imaging a failing drive with retry and error mapping. Always image first with `ddrescue` and work on the copy, never on the original.
6. On SSDs, TRIM interacts with recovery in a way that surprises people. On a TRIM-enabled drive with deterministic read-after-trim, reading a trimmed block returns zeros almost immediately after deletion. So on a modern SSD, deleting a file often does destroy it within seconds, and recovery frequently fails where it would have succeeded on a hard disk. This is a helpful property for privacy and a painful one for accidents.
7. Physical destruction remains the standard for the highest classifications: shredding to a specified particle size, disintegration, or incineration. For flash, particle size matters, because a surviving chip fragment can still hold readable pages.
8. The practical rule: encrypt at first use. Every erase question becomes easy afterwards, and the answer to “what if it fails before I can wipe it” becomes “it does not matter”, which no amount of erasing software can give you.

■ 12.14.6 WORDS – remember these

1. File carving — finding files by their shape, ignoring the index — signature based recovery scanning raw sectors.
2. ATA Secure Erase — the drive wipes itself — an ATA command covering the whole user area including reallocated sectors.
3. Sanitize — the strongest built-in erase — NVMe or ATA operation covering over-provisioned, cached and unmapped regions.

4. Cryptographic erase — throw away the key, not the data — destroying the media encryption key of a self-encrypting drive.
5. Self-encrypting drive — it was always encrypted — media encryption performed in the controller with an internally held key.
6. Full-disk encryption — the key is yours — BitLocker, FileVault, LUKS, normally AES-XTS with a 256-bit key.
7. Degaussing — a magnetic pulse that destroys magnetic media — effective on HDD and tape, useless on flash.

12.98 Common wrong ideas

1. Wrong: deleting a file erases it. Right: deleting removes a directory entry and frees the blocks. The bytes stay until something reuses the space, which is why recovery tools work on hard disks.
2. Wrong: an SSD has no moving parts, so it cannot fail. Right: flash wears out with every erase, leaks charge when unpowered, and SSDs commonly die suddenly from controller or firmware faults with no warning at all.
3. Wrong: defragmenting an SSD helps. Right: an SSD has no seek penalty, so defragmentation only burns write cycles. Windows deliberately runs TRIM instead when the drive is flash.
4. Wrong: RAID means my data is backed up. Right: RAID protects against a drive dying. Deletion, ransomware, corruption, fire and theft all reach every drive in the array at once.
5. Wrong: my 1 TB drive is faulty because the computer shows 931 GB. Right: the maker counted 10^{12} bytes and the operating system is counting 2^{40} bytes. Both numbers are correct, the units differ.
6. Wrong: a quick format wipes the drive. Right: it writes fresh empty bookkeeping and leaves every byte of old data physically in place.
7. Wrong: you must overwrite a drive seven or 35 times to be safe. Right: NIST SP 800-88 says one pass is enough on modern magnetic media, and on flash no number of overwrite passes is reliable. Use sanitize or cryptographic erase.
8. Wrong: more bits per flash cell is simply better value. Right: QLC gives more capacity per chip but roughly a tenth of the endurance of MLC and much slower sustained writes once the cache is exhausted.
9. Wrong: a filesystem journal protects my file contents. Right: most journals protect metadata only. After a crash the structure is consistent but a file may hold a mixture of old and new bytes.
10. Wrong: a 4 GB file will not fit because the USB stick is too small. Right: on FAT32 the file size field is 32 bits, so no file above 4 GiB minus one byte can be described, however much free space there is.

12.99 Chapter summary in 20 lines

1. Memory forgets without power and storage does not; every machine needs both because no technology is fast, cheap, huge and permanent at once.
2. The storage ladder runs from registers at under a nanosecond to tape at tens of seconds, spanning about eight orders of magnitude in latency.
3. Tape is not dead. LTO-10 reached 30 TB native in August 2025 and 40 TB in November 2025, at roughly a third of disk's cost per terabyte.
4. A hard disk writes bits by flipping magnetic domains and reads them by sensing reversals, with the head flying a few nanometres up on an air bearing.

5. The IBM 350 of 1956 held about 3.75 MB on fifty 24-inch discs and weighed over a ton; a 2026 HAMR drive holds 44 TB on ten 3.5-inch platters.
6. Giant magnetoresistance, discovered in 1988 and awarded the 2007 Nobel Prize in Physics, is what made high-capacity drives possible.
7. Disk access time is seek plus rotational latency; average rotational latency is 30,000 divided by RPM in milliseconds.
8. That arithmetic caps a 7,200 RPM drive at roughly 80 random operations per second, a figure that has barely improved in thirty years.
9. Flash stores charge behind an insulator. Reading is cheap, writing damages the insulator, and more bits per cell means far less endurance.
10. NAND's defining rule: you read a page, you write a page, but you can only erase a whole block, so nothing is ever changed in place.
11. That rule forces an SSD to contain a small computer running a flash translation layer, doing wear levelling and garbage collection.
12. Write amplification measures the extra work; it rises sharply as the drive fills, which is why free space is working capital and not waste.
13. TRIM tells the drive which blocks are rubbish; without it an SSD slowly behaves as if permanently full.
14. NVMe over PCIe replaced one 32-command queue with up to 65,535 queues, which is why an SSD on PCIe 5.0 x4 reaches 14 GB/s and SATA stops at 550 MB/s.
15. A filesystem turns a numbered list of blocks into named files; the inode holds the metadata and the name lives in the directory, not in the inode.
16. Creating, writing, renaming and deleting are all metadata edits; renaming a 50 GB file is instant and deleting it overwrites nothing.
17. FAT32 cannot hold a file of 4 GiB or more because its size field is 32 bits wide; exFAT and NTFS remove that limit.
18. Journals make a crash leave a consistent structure, not intact contents; copy-on-write filesystems get the same guarantee by swapping one root pointer.
19. RAID keeps a service running when a drive dies, and protects against nothing else; the 3-2-1 rule, plus a tested restore, is the actual backup.
20. Encrypt every drive from first use, because then secure disposal is one discarded key rather than a hopeful overwrite that flash will quietly ignore.

Buses, Ports, Drivers and Firmware

13.0 What this chapter gives you

1. You will be able to point at any part of a motherboard and say what it does.
2. You will be able to explain what a bus is, and why fast links stopped being wide and became narrow instead.
3. You will be able to read “PCIe 4.0 x4” and work out the bandwidth yourself.
4. You will be able to describe every way a processor talks to a device: memory-mapped registers, ports, polling, interrupts and direct memory access.
5. You will be able to follow an interrupt from a wire to a running piece of code, and read the interrupt counters on a real machine.
6. You will be able to explain exactly what happens in the moment you plug in a USB device, step by step.
7. You will be able to answer the question “what is a driver” with a full chain of layers, not a slogan.
8. You will be able to say how Windows, Linux and macOS each load drivers, and how each one picks the right driver for a chip.
9. You will be able to separate firmware from driver from application, and say where each one lives.
10. You will be able to describe what happens in the first second after you press the power button, up to the point where the bootloader takes over.

Chapter 18 continues the story from where the bootloader hands off to the operating system kernel. This chapter stops at that handover.

13.1 The motherboard as a city

■ 13.1.1 PLAIN – in simple words

1. The motherboard is a flat green board with everything plugged into it.
2. It is not one thing. It is a set of sockets, slots, wires and small chips.
3. The wires are not loose. They are thin copper lines baked into the board itself, called traces.
4. A modern board is a sandwich of six to sixteen thin layers of copper and glue, so traces can cross without touching.
5. Every part on the board does one job. Once you know the jobs, the board stops looking like a maze.
6. The big square socket in the middle holds the processor.
7. The long thin slots beside it hold the memory sticks.
8. The long slots below hold cards, such as a graphics card.
9. The small flat slots hold storage sticks.

10. The metal cluster at the back edge holds the connectors you can see from outside the case.

■ 13.1.2 PLAIN – a picture in your head

1. Think of the board as a small city seen from above.
2. The processor socket is the city centre, where all the work is done.
3. The power circuits next to it are the power station, feeding the centre.
4. The memory slots are the warehouses, right next to the centre because the trips there are constant.
5. The chipset is the suburban traffic hub, handling everything that is not urgent enough to reach the centre directly.
6. The card slots are goods yards where big vehicles dock.
7. The back panel is the city gates, where things enter from outside.
8. The little battery is the town clock's backup, keeping time when the power is off.

Where this comparison breaks: a real city has traffic that goes anywhere. On a board almost every route is fixed at the factory and cannot change. A trace between two points is a permanent street, not a road that can be re-planned.

■ 13.1.3 PLAIN – a worked example

1. Take a common desktop board layout and walk it top to bottom.
2. Here is what you would find, in order, on a typical mid-range board sold since about 2023.

```
[ rear I/O panel ] [ CPU power socket ]
+-----+
| VRM | CPU socket | VRM | D | D | D | D |
|-----+-----+-----| I | I | I | I |
| PCIe x16 slot (from CPU) | M | M | M | M |
| [ M.2 slot, under heatsink ] M | M | M |
|-----+-----+-----|
| chipset chip under heatsink | SATA ports |
| PCIe x1 | PCIe x4 | M.2 | M.2 | BIOS chip |
| front panel + USB headers | CMOS battery |
+-----+-----+-----+
```

3. The board is read like a page. Power at the top, processor in the middle, slow devices at the bottom.
4. Fast things sit near the processor because the wires must be short.
5. Slow things sit far away because a longer wire does not hurt them.

■ 13.1.4 PLAIN – what is really happening inside

1. **CPU socket:** a bed of spring pins that press against the processor. It carries power in and every high-speed signal out.
2. **VRM**, short for voltage regulator module: the small square chips, coils and cylinders next to the socket. They cut 12 volts down to under 1.5 volts for the processor.
3. **DIMM slots:** the long clipped slots holding memory sticks. Their wires go straight to the processor on any modern board.
4. **Chipset:** one large chip under a low heatsink. It is a fan-out switch, turning a few fast lanes from the processor into many slower ports.
5. **PCIe slots:** the long open-ended slots for cards. They carry high-speed serial links plus power.
6. **M.2 slots:** short flat slots for storage sticks. They usually carry the same kind of high-speed link as a card slot, in a smaller shape.
7. **SATA ports:** small flat right-angle sockets for older drives.

8. **Headers:** bare pin blocks with no plastic shell. The case's power button, front USB ports, fans and front audio plug into these.
9. **CMOS battery:** a coin cell that keeps the clock and settings alive when the machine is unplugged.
10. **BIOS flash chip:** a tiny eight-legged chip holding the startup code. Without it the board cannot even find the processor's first instruction.
11. **Super IO chip:** a small chip that runs fans, reads temperatures and voltages, and provides the last old-style ports.
12. **Rear IO panel:** the metal strip of outward-facing connectors.

■ 13.1.5 TECHNICAL – the engineer's version

1. Sockets are LGA (land grid array, pins in the socket) or PGA (pin grid array, pins on the chip). Intel LGA1700 has 1700 contacts; AMD AM5 is LGA with 1718 contacts and replaced the pinned AM4 in 2022.
2. A large share of socket contacts are power and ground, not signals. On LGA1700 only a minority carry data.
3. VRMs are multiphase synchronous buck converters. A phase is one inductor plus a high-side and low-side MOSFET, driven by a PWM controller. Phases are switched out of step to smooth the current ripple.
4. A desktop processor core voltage is roughly 0.7 V to 1.4 V, and sustained current can exceed 200 A on high-end parts. That is why the VRM needs heatsinks.
5. DDR4 and DDR5 unbuffered DIMMs both have 288 pins, but the key notch sits in a different place, so they are not interchangeable. DDR5 moved the power management IC onto the DIMM itself.
6. On Intel platforms the chipset is the PCH (platform controller hub), attached over DMI (Direct Media Interface), which is electrically PCI Express. Recent desktop chipsets use a DMI 4.0 x8 link.
7. On AMD AM5 the chipset is called a Promontory device and attaches over a PCIe link; high-end boards chain two chipset chips together.
8. The firmware chip is SPI NOR flash in an 8-pin SOIC or WSON package, typically 16 MB or 32 MB (128 or 256 megabit) on boards since about 2020.
9. Super IO chips are made by Nuvoton and ITE, for example the ITE IT8689E. They hang off the LPC or eSPI bus and expose fan tachometers, PWM fan control, voltage and temperature sensing, and legacy serial ports.
10. The coin cell is a CR2032, 3 V lithium, about 225 mAh, and typically lasts three to seven years.
11. Board layer counts: four layers on budget boards, eight to twelve on mainstream boards, and more on server boards, because DDR5 and PCIe 5.0 signalling needs clean reference planes.

Part	Job in one line	Typical figure
VRM phase	Step 12 V down	50 to 110 A each
CMOS cell	Keep clock and setup	CR2032, 3 V
BIOS flash	Hold startup code	16 to 32 MB SPI
DMI 4.0 x8	CPU to chipset link	About 16 GB/s

12. Tools that show this without opening the case: `dmidecode -t 2` on Linux prints the board maker and model, and `lspci` lists the chipset devices.

■ 13.1.6 WORDS – remember these

1. Trace — a printed wire on the board — a copper track etched in a PCB layer.
2. Socket — the seat the processor sits in — LGA or PGA contact array.

3. VRM — the power step-down circuit — multiphase synchronous buck converter.
4. Chipset — the traffic hub chip — PCH or southbridge fan-out switch.
5. Header — a bare block of pins — an unshrouded board-level connector.
6. Super IO — the fans and sensors chip — LPC or eSPI attached legacy IO controller.
7. Flash chip — where startup code lives — SPI NOR device holding the firmware image.

13.2 What a bus actually is

■ 13.2.1 PLAIN – in simple words

1. A bus is a set of wires shared by several parts, so they can pass numbers to each other.
2. It carries three kinds of information: where, what, and when.
3. The “where” wires carry an address, meaning which device or which memory slot is being spoken to.
4. The “what” wires carry the data itself.
5. The “when” wires carry control signals: read or write, start now, I am ready, I am finished.
6. Only one part may talk at a time on a shared bus, or the messages would collide.
7. There are two ways to send many bits: side by side on many wires, or one after another on a few wires.
8. Side by side is called parallel. One after another is called serial.
9. Parallel sounds faster, and for a long time it was.
10. Above a certain speed, parallel stops working and serial wins. This chapter explains why.

■ 13.2.2 PLAIN – a picture in your head

1. Imagine eight runners set off together carrying one letter each. The message is only complete when all eight arrive.
2. That is a parallel bus. Eight wires, eight bits, all at once.
3. If one runner is slightly slower, everyone waits for them. The whole message is only as fast as the worst runner.
4. Now make them run much faster. Tiny differences in path length become large differences in arrival time.
5. Worse, running close together, they knock into each other and drop letters.
6. Now imagine one runner, on one clean track, carrying letters one after another, but sprinting far faster than the group ever could.
7. That is a serial link. Fewer wires, but each wire far quicker.

Where this comparison breaks: real serial links are not truly one runner. They have a separate track in each direction, and often several tracks in parallel called lanes. The difference is that each lane carries its own timing, so the lanes do not have to arrive in perfect step.

■ 13.2.3 PLAIN – a worked example

1. The bandwidth of a simple parallel bus is width times clock rate.
2. Formula: bytes per second = (bus width in bits / 8) x transfers per second.
3. Old PCI: 32 bits wide, 33.33 million transfers per second.
4. $32 / 8 = 4$ bytes. $4 \times 33,333,333 = 133,333,332$ bytes per second.
5. That is about 133 MB/s, and it is the number the PCI specification quotes.
6. Now a 64-bit PCI slot at 66 MHz: $8 \text{ bytes} \times 66,666,666 = \text{about } 533 \text{ MB/s}$.
7. Now compare one PCI Express 3.0 lane. It is two wires each way, not 32.
8. It runs at 8 gigatransfers per second and delivers about 985 MB/s each way at the same time.

9. One tiny serial lane beats the whole 32-wire parallel bus, and does it in both directions at once.

Link	Wires for data	Bandwidth
PCI 32-bit 33 MHz	32 shared	133 MB/s shared
PCI 64-bit 66 MHz	64 shared	533 MB/s shared
PCIe 3.0 x1	4 (2 per way)	985 MB/s each

■ 13.2.4 PLAIN – what is really happening inside

1. On a parallel bus every bit of one number leaves at the same instant, on its own wire.
2. Traces on a board are never exactly the same length, because they must bend around other parts.
3. A signal moves along a board trace at roughly half the speed of light, about 15 centimetres per nanosecond.
4. So a 1.5 centimetre difference in trace length is a 0.1 nanosecond difference in arrival. That gap is called skew.
5. At 33 million transfers per second each bit lasts 30 nanoseconds, so 0.1 nanosecond of skew does not matter at all.
6. At 8 billion transfers per second each bit lasts 0.125 nanoseconds. Now 0.1 nanoseconds of skew almost swallows the whole bit.
7. Board makers fix this by adding wiggles to short traces to match lengths, which is why you see squiggly patterns near memory slots.
8. Length matching gets harder as speed rises, and eventually impossible.
9. Second problem: a changing signal on one wire pushes a small copy of itself onto the wire beside it. That is crosstalk. More wires close together means more of it.
10. Third problem: on a shared bus, every device hanging off the wires adds electrical load, which rounds off the sharp edges of the signal.
11. Serial links dodge all three. Fewer wires means less crosstalk. Each lane carries its own timing, so skew between lanes can be corrected in the receiver. Point-to-point means only two ends, so the load is fixed and known.
12. Modern serial links do not send a separate clock. The clock is recovered from the data itself, which is why the data has to be encoded so it never stays flat for long.

■ 13.2.5 TECHNICAL – the engineer's version

1. A classic bus has three groups: an address bus, a data bus and a control bus. The 8088 in the 1981 IBM PC had a 20-bit address bus and an 8-bit external data bus, reaching 1 MiB of address space.
2. Shared multi-drop buses are limited by settling time, reflections from unterminated stubs, and the capacitive load of every attached device.
3. Modern serial links are differential: each lane is a pair of wires driven to opposite voltages. The receiver reads the difference, so noise that hits both wires equally cancels out.
4. Signalling levels are low. PCI Express uses roughly 0.8 to 1.2 V differential swing, versus 5 V or 3.3 V single-ended on old PCI.
5. Clocking is embedded. Transmitters scramble and encode data so transitions occur often enough for a clock and data recovery (CDR) circuit to lock on.
6. Line codes cost bandwidth. 8b/10b sends 10 symbols for 8 data bits, a 20 percent overhead. 128b/130b sends 130 for 128, about 1.5 percent.
7. Raw rate is quoted in GT/s, gigatransfers per second, not GHz, because modern links may carry more than one bit per transfer.

8. Bandwidth formula for a serial link: $\text{bytes/s} = (\text{GT/s} \times \text{lanes} \times \text{coding efficiency}) / 8$.
9. Worked check for PCIe 3.0 x1: $8 \text{ GT/s} \times 1 \times (128/130) / 8 = 0.985 \text{ GB/s}$.
10. Worked check for PCIe 5.0 x4: $32 \times 4 \times (128/130) / 8 = 15.75 \text{ GB/s}$ each direction.

Parallel, shared: CPU ===== dev dev dev dev (all share one set of wires)	Serial, point-to-point: CPU <-TX--> device (one pair each way, own clock recovery)

11. Skew budget: at 32 GT/s a unit interval is 31.25 picoseconds. Board designers work to intra-pair skew targets in the low single-digit picoseconds. This is why parallel buses stopped scaling.
12. The honest version: serial did not beat parallel on raw physics alone. It won because it moved the hard problem into silicon, where equalization, scrambling and error correction are cheap, and away from the board, where length matching is expensive.
13. Parallel is not dead. Inside a chip, and between a processor and DRAM, wide parallel interfaces still win because the distances are short and fixed. DDR5 still uses a 64-bit data path per channel pair.

■ 13.2.6 WORDS - remember these

1. Bus — shared wires between parts — a multi-drop interconnect with address, data and control groups.
2. Parallel — many bits at once — simultaneous transmission across a wide data path.
3. Serial — bits one after another — sequential transmission on a lane.
4. Skew — bits arriving at different times — timing difference between parallel signals at the receiver.
5. Crosstalk — one wire disturbing its neighbour — unwanted coupling between adjacent conductors.
6. Differential pair — two wires carrying opposites — balanced signalling with common-mode noise rejection.
7. GT/s — how many transfers per second — gigatransfers per second, before coding overhead is removed.
8. Line code — the packaging of bits on the wire — the mapping such as 8b/10b or 128b/130b.

13.3 The history of expansion buses

■ 13.3.1 PLAIN - in simple words

1. From the beginning, computers needed a way to add hardware after buying the machine.
2. The answer was a slot: a row of contacts on the board, into which a card slides.
3. The first widely copied slot on personal computers came with the IBM PC in 1981.
4. It was slow, but it was documented, so anyone could make cards for it.
5. Graphics grew hungry, and each new slot design existed mainly because pictures needed more room.
6. Every generation was faster and had fewer wires than you would expect.
7. Today one design has replaced them all: PCI Express, usually written PCIe.
8. PCIe comes in different physical sizes and different speed generations, and the two are separate ideas.
9. A slot's size tells you how many lanes it can hold. A generation tells you how fast a lane runs.

■ 13.3.2 PLAIN – a picture in your head

1. Think of a loading bay with a number of doors.
2. The size of the bay is how many doors it has: one, four, eight or sixteen.
3. The generation is how fast one door can move goods.
4. A sixteen-door bay of an old generation and a four-door bay of a new generation can move about the same amount.
5. That is why an older graphics card in a big slot can be matched by a newer card in a smaller one.
6. Also, a bay may have sixteen doors built, but only four of them connected to a road. The bay is full-size but only four doors are usable.

Where this comparison breaks: doors move goods one way. A PCIe lane moves data both ways at once, so the total for a link is double the one-way figure if both directions are busy.

■ 13.3.3 PLAIN – a worked example

1. A graphics card that says “PCIe 4.0 x16” wants sixteen lanes at generation four speed.
2. Generation four gives about 1.969 GB/s each way per lane.
3. Sixteen lanes: $16 \times 1.969 =$ about 31.5 GB/s each way.
4. Put the same card in a slot wired at x4. Now $4 \times 1.969 =$ about 7.9 GB/s.
5. Put it in a PCIe 3.0 x16 slot instead. $16 \times 0.985 =$ about 15.75 GB/s.
6. Notice that a generation four x8 link and a generation three x16 link give the same figure. That is not a coincidence: each generation roughly doubles the per-lane speed.

■ 13.3.4 PLAIN – what is really happening inside

1. **ISA, 1981:** 62 contacts, 8 data bits, running at the processor clock of 4.77 MHz. In 1984 the IBM PC/AT added a second connector section for 16 bits, and the speed settled at 8 MHz.
2. Cards on ISA had switches and jumpers on them, because nothing was automatic. You set the card’s address and interrupt number by hand.
3. **VESA Local Bus, 1992:** a hack. It bolted an extra connector onto the end of an ISA slot and wired it straight to the 486 processor’s own bus.
4. It was fast for the time, up to about 200 MB/s, but the processor could only drive two or three such cards, and it died when the Pentium changed the processor’s bus.
5. **PCI, 1992:** designed at Intel and then handed to a group called the PCI Special Interest Group. 32 bits at 33 MHz, about 133 MB/s, shared by every card in the machine.
6. PCI’s real gift was not speed. It was that each card could describe itself, so the machine could set addresses and interrupts automatically. No more jumpers.
7. **AGP, 1997:** a private lane just for graphics, so the video card did not have to queue behind everything else on PCI.
8. **PCI Express, 2003:** the change of shape. It threw away the shared parallel bus and used point-to-point serial lanes with a switch in the middle.
9. Each PCIe device gets its own link. Nothing is shared, so one slow device no longer holds up the rest.

■ 13.3.5 TECHNICAL – the engineer’s version

1. Dates and figures for the main expansion buses:

Bus	Year	Peak bandwidth
ISA 8-bit	1981	About 4 to 8 MB/s

Bus	Year	Peak bandwidth
ISA 16-bit	1984	About 8 to 16 MB/s
VESA Local Bus	1992	Up to about 200 MB/s
PCI 32/33	1992	133 MB/s shared
AGP 8x	2002	2133 MB/s
PCIe 1.0 x16	2003	4 GB/s each way

- AGP was introduced by Intel with the 440LX chipset on 26 August 1997. Its base clock was 66 MHz: 1x gives 266 MB/s, 2x gives 533 MB/s, 4x gives 1066 MB/s at 1.5 V, and 8x gives 2133 MB/s at 0.8 V.
- Compaq coined the retronym Industry Standard Architecture for the IBM AT bus in the late 1980s, because IBM never gave it a public name.
- PCI Express per-lane figures, one direction, after coding overhead:

Gen	Year	GT/s	Per lane each way
1.0	2003	2.5	250 MB/s
2.0	2007	5.0	500 MB/s
3.0	2010	8.0	985 MB/s
4.0	2017	16.0	1.969 GB/s
5.0	2019	32.0	3.938 GB/s
6.0	2022	64.0	7.563 GB/s
7.0	2025	128	15.125 GB/s

- Generations 1.0 and 2.0 use 8b/10b, so the usable fraction is 80 percent. Generations 3.0 through 5.0 use 128b/130b, about 98.5 percent.
- Generation 6.0, released in January 2022, changed the physics: PAM4 signalling carries two bits per symbol, with forward error correction and fixed 256-byte FLIT packets carrying 242 bytes of payload. Generation 7.0 was finalized in June 2025 and keeps PAM4.
- First shipping PCIe 6.0 hardware appeared in 2025, three years after the specification. Specification date and product date are always different. Say which one you mean.
- Physical versus electrical width: a slot with a x16 connector may only have four lanes wired. This is written as “x16 slot, x4 electrical”. It happens because the processor has a fixed lane budget.
- A typical desktop processor since 2021 exposes 20 to 28 usable lanes. A graphics card takes 16 and an NVMe drive takes 4, so the rest of the slots are wired through the chipset and share the chipset’s uplink.
- Bifurcation** is splitting one wide link into several narrower ones: an x16 root port configured as x4/x4/x4/x4 to drive four NVMe drives on one adapter card. It must be supported by the processor’s root complex and exposed in firmware setup. Cheap adapter cards that do not include a PCIe switch require bifurcation; cards with a switch chip do not.
- Whether bifurcation is available at all is an implementation detail of one processor and one board’s firmware, not something PCI Express guarantees. Two boards with the same chipset often differ.
- Link training negotiates width and speed at power-on. If a lane fails, the link retrains narrower, so a card can silently fall back to x8 or x1.
- Observe it on Linux with `lspci -vv` and read the `LnkCap` and `LnkSta` lines:

```
LnkCap: Speed 16GT/s, Width x16
LnkSta: Speed 2.5GT/s (downgraded), Width x16
```

14. A speed of 2.5 GT/s at idle is usually normal power saving, not a fault. Check under load before concluding anything.

■ 13.3.6 WORDS – remember these

1. Slot — the place a card plugs in — a card edge connector on a defined pinout.
2. Lane — one serial path — a transmit pair plus a receive pair in PCIe.
3. Generation — how fast a lane runs — the PCIe specification revision setting the transfer rate.
4. Electrical width — how many lanes are actually wired — the negotiated link width, which may be less than the connector size.
5. Bifurcation — splitting one wide slot into several — configuring a root port as multiple narrower links.
6. Link training — the handshake at power-on — the LTSSM process that agrees width, speed and equalization.

13.4 How the processor talks to a device

■ 13.4.1 PLAIN – in simple words

1. A device is not magic. It is a chip with a set of numbered boxes inside it, called registers.
2. Writing a number into one box makes the device do something.
3. Reading a number out of another box tells you what the device is doing.
4. So “controlling a device” means writing and reading small numbers at fixed places.
5. There are two ways for the processor to reach those boxes.
6. The first way pretends the device’s boxes are ordinary memory addresses. Write to address such-and-such and the device hears it. This is called memory-mapped input and output.
7. The second way uses a separate, smaller set of addresses reached only by two special instructions. This is called port-mapped input and output.
8. Once you can reach the boxes, you still need to know when the device has finished a job.
9. You can keep asking, over and over. That is called polling, and it wastes the processor’s time.
10. Or the device can tap the processor on the shoulder when it is ready. That is an interrupt.
11. Moving a lot of data through the processor one piece at a time is slow. So devices are allowed to move data to and from memory by themselves. That is direct memory access.

■ 13.4.2 PLAIN – a picture in your head

1. Imagine a hotel reception desk with a row of pigeonholes on the wall behind it.
2. Each pigeonhole is a device register. Putting a card in box 3 means “start the lift”. Reading box 4 tells you which floor the lift is on.
3. Memory-mapped input and output means those pigeonholes are simply part of the same wall as all the other room boxes. You use the same ladder.
4. Port-mapped input and output means the pigeonholes are in a locked side room with its own small ladder and its own numbering.
5. Polling is standing at the desk asking “is the lift here yet?” every two seconds, doing nothing else.
6. An interrupt is going back to work and letting a bell ring when the lift arrives.
7. Direct memory access is telling the porter to move the luggage between rooms without carrying each bag past the desk yourself.

Where this comparison breaks: a hotel porter can be trusted to only touch the right rooms. A device doing direct memory access can, unless stopped, write anywhere in memory. That risk is exactly why the IOMMU was invented.

■ 13.4.3 PLAIN – a worked example

1. Here is a classic serial port, in the style of the original IBM PC, using port-mapped input and output.
2. The first serial port lives at port address 0x3F8.
3. Port 0x3F8 is the data box. Port 0x3FD is the status box.
4. Bit 5 of the status box means “the transmit box is empty, you may send”.
5. Sending one character by polling looks like this.

```
/* wait until bit 5 of the status port is set */
while ((inb(0x3FD) & 0x20) == 0) {
    /* spin: the CPU does nothing useful here */
}
/* Linux order is outb(value, port). Some other
   headers use outb(port, value). Always check. */
outb('A', 0x3F8);    /* write the character */
```

6. At 9600 bits per second, one character takes about 1 millisecond.
7. A 3 GHz processor executes roughly 3 million cycles in that millisecond.
8. So sending one character by polling burns about 3 million cycles of doing nothing.
9. With an interrupt, the processor writes the character, goes off and runs other programs, and is called back when the port is free again.
10. With direct memory access, the processor hands over a pointer to 4 KB of text and is called back once, at the end, instead of 4096 times.

■ 13.4.4 PLAIN – what is really happening inside

1. When the processor executes a write to a memory address, that address goes out on the interconnect.
2. Something has to decide who owns that address. That job belongs to address decoding logic in the processor’s root complex and in bridges.
3. Ranges of the address space are handed out at start-up to devices. Each device says how big a window it needs, and firmware assigns it a base address.
4. So writing to address 0xF7C0_0010 might not touch memory at all. It might land in register 0x10 of a network card.
5. These device windows must not be cached, because reading the same address twice can legitimately give different answers, and a write must reach the device now and not sit in a cache.
6. Port-mapped input and output is different. On x86 the instructions IN and OUT carry a 16-bit port number, giving 65,536 ports in a space entirely separate from memory.
7. For direct memory access, the processor writes into the device’s registers the address of a buffer in main memory and a length, then sets a “go” bit.
8. The device then becomes a bus master: it issues its own read and write requests to memory, without the processor being involved.
9. Real buffers are rarely one solid block of physical memory. A 1 MB buffer seen by a program as continuous may be scattered across 256 separate 4 KB physical pages.
10. So the driver builds a list of address and length pairs, and hands the list to the device. The device walks the list. This is scatter-gather.
11. Since the device writes straight into memory, the processor’s cached copy of that memory can become stale. Either the hardware keeps caches coherent, or the driver must explicitly invalidate the range.
12. A device that can write anywhere in memory can overwrite the operating system. The IOMMU

sits between devices and memory and translates and checks every device address, exactly as the MMU does for programs.

■ 13.4.5 TECHNICAL – the engineer’s version

1. MMIO regions are advertised through PCI Base Address Registers (BARs). A device has up to six 32-bit BARs; a 64-bit BAR consumes two.
2. At power-on, firmware writes all ones to a BAR and reads it back. The number of low bits that stay zero tells you the size of the window the device wants. Firmware then programs a base address.
3. On x86 the page attributes for MMIO are set to UC (uncacheable) or WC (write-combining) via the MTRRs and PAT, so accesses are not reordered or merged in ways the device would misread.
4. Port IO on x86 uses IN, OUT, INS and OUTS. Notable legacy addresses: 0x20 and 0xA0 for the two 8259A interrupt controllers, 0x60 and 0x64 for the keyboard controller, 0x70 and 0x71 for CMOS and the real-time clock, 0x3F8 for COM1, 0x378 for LPT1.
5. Port IO survives only for legacy compatibility. ARM and RISC-V have no separate IO space at all; everything is memory-mapped.
6. The original PC used an Intel 8237A DMA controller with four channels, and the PC/AT added a second for 16-bit channels 5, 6 and 7, with channel 4 used to cascade. This is third-party DMA, where a central controller moves the data.
7. Third-party DMA is obsolete on PCI Express. Every PCIe endpoint that moves bulk data is a bus master doing first-party DMA.
8. Bus mastering must be explicitly enabled by setting the Bus Master Enable bit in the device’s PCI command register. A driver that forgets this sees a device that appears to do nothing.
9. Scatter-gather lists are called SGLs, or descriptor rings in networking. An NVMe drive supports both PRP (physical region page) lists and SGLs.
10. IOMMU implementations: Intel VT-d, AMD-Vi, and Arm SMMU. They provide an IO virtual address space per device, using the PCIe requester ID (bus:device.function) to select a page table.
11. The IOMMU defends against DMA attacks. The FireWire DMA attacks of the 2000s and the Thunderclap research of 2019 both exploited devices with unrestricted memory access. It also makes device passthrough to virtual machines safe.
12. IOMMU costs. Address translation adds latency, and IOTLB misses hurt. High-throughput setups often use large pages or, in some deployments, passthrough mode, trading protection for speed. Engineers genuinely disagree about where that line should be.

Mechanism	Who moves the data	Cost per byte
Polled PIO	The CPU, in a loop	Very high
IRQ + PIO	The CPU, on demand	High
DMA	The device itself	Near zero for CPU

13. Observe on Linux: `lspci -vv` shows BAR regions as “Memory at ...” and “I/O ports at ...”, and `cat /proc/iomem` and `/proc/ioports` list the whole assignment.

■ 13.4.6 WORDS – remember these

1. Register — a numbered box inside a chip — an addressable control or status location in a device.
2. MMIO — device boxes look like memory — memory-mapped IO through a BAR window, marked uncacheable.
3. Port IO — a separate small address space — x86 IN and OUT over a 16-bit port space.
4. Polling — asking again and again — busy-wait on a status register.

5. DMA — the device moves data itself — direct memory access by a bus master.
6. Scatter-gather — a list of memory pieces — a descriptor list of address and length pairs.
7. IOMMU — the guard between device and memory — an IO memory management unit translating and checking device addresses.

13.5 Interrupts, end to end

■ 13.5.1 PLAIN – in simple words

1. An interrupt is a device saying “stop what you are doing and deal with me”.
2. Without interrupts, the processor would have to check every device constantly, which would waste almost all of its time.
3. The processor finishes the instruction it is on, saves where it was, and jumps to a small piece of code written for that device.
4. That piece of code is called the interrupt handler, or interrupt service routine.
5. When the handler finishes, the processor restores what it saved and carries on exactly where it left off. The interrupted program never knows.
6. Many devices may want attention at once, so a helper chip collects and ranks the requests. That is the interrupt controller.
7. Each source has a number, so the processor knows which handler to run.
8. A table in memory maps each number to the address of its handler.
9. Interrupt handlers must be extremely short, because everything else is frozen while one runs.
10. So the work is split: a tiny urgent part now, and the slow part scheduled for a moment later.

■ 13.5.2 PLAIN – a picture in your head

1. Imagine a clerk working through a pile of forms, with a bell on the desk.
2. Anyone needing attention rings the bell rather than standing in the queue.
3. The clerk puts a bookmark in the form they are on, deals with the bell, then goes back to the exact same line of the same form.
4. There are many bells, one per department, all wired into a small box on the wall that ranks them and passes the most important one through.
5. Beside the desk is a printed list: bell number 1 means the front door, bell number 4 means the fire door.
6. If a bell means “the delivery arrived”, the clerk does not unload the van right then. They write “van outside” on a note, and go back to the forms. The unloading happens between forms.
7. That split is the top half and the bottom half.

Where this comparison breaks: a clerk can ignore a bell. A processor with interrupts enabled cannot, which is why a device stuck ringing its bell can freeze a machine completely. That failure is called an interrupt storm.

■ 13.5.3 PLAIN – a worked example

1. You press the letter A on an old-style keyboard attached to a PC.
2. The keyboard controller sees a key change and raises interrupt request 1.
3. The interrupt controller checks that IRQ 1 is not masked and that nothing more important is being served, then signals the processor.
4. The processor finishes its current instruction and reads the interrupt number, 0x21 in the classic mapping.

5. It looks up entry 0x21 in the interrupt descriptor table and jumps to the handler address stored there.
6. The handler reads one byte from port 0x60. That byte is a scan code, not a letter. For A it is 0x1E on press and 0x9E on release.
7. The handler puts the scan code in a queue and returns immediately. Total time: a few microseconds.
8. Later, ordinary kernel code takes the scan code out of the queue, maps it through the current keyboard layout, and delivers the character A to whichever window has focus.
9. Note what did not happen inside the handler: no layout lookup, no window search, no drawing.

■ 13.5.4 PLAIN – what is really happening inside

1. In the oldest design, an interrupt is literally a wire. The device pulls the wire and holds it until it is serviced. This is level-triggered.
2. A wire per device does not scale, so several devices shared one wire and the handler had to ask each of them “was it you?”.
3. Message-signalled interrupts replaced the wire completely. The device does a normal memory write to a special address, and that write is the interrupt. No extra wire at all.
4. The interrupt controller’s jobs are: collect requests, apply masks, rank by priority, and hand the processor a number.
5. The number is an index into a table of handler addresses.
6. Before jumping, the processor pushes enough state to return: the instruction pointer, the flags, and on x86 the code segment.
7. The handler runs with further interrupts of the same or lower priority blocked, so it must not wait for anything.
8. The handler acknowledges the interrupt, so the controller knows it may deliver the next one. Forgetting this leaves the machine wedged.
9. Then it does the smallest possible urgent job, and queues the rest.
10. The queued rest runs later with interrupts enabled again, so it may take locks, allocate memory, and wake up sleeping processes.
11. At very high rates, even a short handler costs too much. A network card receiving 1 million packets per second cannot afford 1 million interrupts.
12. So the device is told to wait: fire one interrupt per 32 packets, or one every 50 microseconds, whichever comes first. That is coalescing. It trades a little latency for a lot of processor time.

■ 13.5.5 TECHNICAL – the engineer’s version

1. The original IBM PC used one Intel 8259A PIC with 8 inputs. The 1984 PC/AT cascaded a second 8259A into IRQ 2, giving 15 usable lines.
2. Classic IRQ assignments, still visible in firmware and in old manuals. These are a convention set by the IBM PC and PC/AT, not a written standard, but every clone copied them and software depended on them:

IRQ	Assigned to	Note
0	System timer	8253/8254 PIT
1	Keyboard	Port 0x60
2	Cascade to slave	Not usable directly
6	Floppy controller	Legacy
12	PS/2 mouse	Port 0x60 also

IRQ	Assigned to	Note
14	Primary ATA	Legacy disk

- The APIC (Advanced Programmable Interrupt Controller) arrived with the Pentium era. Each core has a local APIC; an IO APIC, such as the Intel 82093AA, routes external lines. This allows routing an interrupt to a chosen core and supports 24 inputs per IO APIC.
- MSI (message-signalled interrupts) was added in the PCI 2.2 specification. A device performs a posted memory write to an address in the range 0xFEE00000 on x86, carrying a vector in the data payload.
- MSI allows up to 32 vectors per device in the PCI specification, though Windows uses at most 16. MSI-X, added in PCI 3.0, allows up to 2048 vectors, each with its own address, data and mask bit in a table in device memory.
- MSI-X is what makes multi-queue devices work: one vector per queue, each pinned to a different core, so a 32-core machine can service a NIC on all 32 cores without lock contention.
- x86 has 256 interrupt vectors, 0 to 255. Vectors 0 to 31 are reserved for processor exceptions such as divide error (0), page fault (14) and general protection fault (13). Devices use 32 upward.
- In real mode the table is the IVT at physical address 0, 256 entries of 4 bytes. In protected and long mode it is the IDT, 256 entries of 8 or 16 bytes, located by the IDTR register and loaded with LIDT.
- Linux names the split top half and bottom half. The top half is the handler registered with `request_irq`. Bottom halves are `softirqs`, tasklets, threaded IRQ handlers, or workqueues, in increasing order of how much they are allowed to do.
- Windows names it differently: the ISR runs at device `IRQL`, and defers work to a DPC (deferred procedure call) at `DISPATCH_LEVEL`.
- Coalescing on Linux NICs is controlled with `ethtool -c eth0` to read and `ethtool -C eth0 rx-usecs 50` to set. NVMe uses a similar aggregation threshold and time window.
- Read live counts on Linux:


```
cat /proc/interrupts          # per-CPU counts, per source
watch -n1 'grep nvme /proc/interrupts'
```
- On macOS there is no `/proc`; use `ioreg -l` for device trees and the Instruments tool for interrupt activity. On Windows, Performance Monitor exposes “Interrupts/sec” and “DPCs Queued/sec”.
- An interrupt storm shows as one line in `/proc/interrupts` climbing by hundreds of thousands per second while the machine is idle. The usual cause is a shared level-triggered line whose owner never deasserts.

■ 13.5.6 WORDS - remember these

- Interrupt — a device asking for attention — an asynchronous transfer of control to a handler.
- IRQ — the request number — an interrupt request line or its assigned index.
- Vector — the table entry number — an index into the IDT selecting a handler.
- ISR — the code that runs on an interrupt — the interrupt service routine, running with interrupts partly masked.
- Top half and bottom half — urgent bit now, slow bit later — hard IRQ context versus `softirq`, tasklet, workqueue or DPC.
- MSI-X — an interrupt sent as a memory write — message-signalled interrupts with up to 2048 per-vector table entries.
- Coalescing — batching interrupts — delaying notification by a packet count or time threshold to cut CPU overhead.

13.6 USB, properly

■ 13.6.1 PLAIN – in simple words

1. Before 1996, every kind of device had its own kind of plug.
2. A mouse had one plug, a keyboard another, a printer a third, a modem a fourth, and a joystick a fifth.
3. Each of them needed its own settings, and each machine had only one or two of each socket.
4. USB, the Universal Serial Bus, was made to replace all of them with one plug that anyone could use.
5. Its three promises were: one connector for everything, plug it in while the machine is running, and no manual settings.
6. It also carries power, so small devices need no separate supply.
7. USB is not a shared party line. There is one boss, called the host, and everything else answers only when asked.
8. Devices never speak first. Even a mouse only replies when the host asks it for an update, which the host does about once every millisecond.
9. You can add more sockets by plugging in a hub, which is a splitter that passes the host's questions along.
10. When you plug something in, the machine goes through a fixed set of steps to find out what it is. That process is called enumeration.

■ 13.6.2 PLAIN – a picture in your head

1. Think of a teacher in a classroom who alone decides who speaks.
2. The teacher goes around asking each pupil in turn: "anything for me?"
3. Most pupils say "nothing". A mouse says "nothing" a thousand times a second, then once says "moved left by three".
4. No pupil ever shouts out. If two spoke at once, neither would be heard.
5. Group leaders sit between the teacher and the pupils, passing questions down and answers back. Those are hubs.
6. A new pupil arriving at the door is noticed by the nearest group leader, who tells the teacher. The teacher then asks the newcomer their name, gives them a seat number, and asks what subject they study.

Where this comparison breaks: USB 3 and later added a second set of wires that lets a device signal readiness rather than being asked, which is closer to raising a hand. And USB On-The-Go lets a phone act as the teacher instead of the pupil. The strict one-boss model is true of USB 2 and earlier.

■ 13.6.3 PLAIN – a worked example

1. Plug a keyboard into a running computer. Here is the exact sequence.

1 attach	device pulls one data line high through a resistor; the hub sees the change
2 report	hub tells host: something on port 3
3 reset	host tells hub to reset that port
4 default	device now answers at address 0, endpoint 0
5 read 8	host reads first 8 bytes of the device descriptor to learn the max packet size
6 address	host sends SET_ADDRESS with a free number from 1 to 127; device switches to it
7 descript	host reads full device, configuration, interface and endpoint descriptors

8 match	host looks up class, vendor and product IDs and picks a driver
9 configure	host sends SET_CONFIGURATION; device powers up its interfaces and starts work

2. In step 7 the host learns that this device reports interface class 3, which means human interface device, subclass 1 and protocol 1, which mean boot keyboard.
3. That is enough. No vendor driver is needed. The operating system's own HID keyboard driver handles it.
4. Total time from plug to working: typically well under a second.
5. Now the steady state. The host polls the keyboard's interrupt endpoint every 8 milliseconds by default for a low-speed keyboard.
6. Each poll returns 8 bytes: one byte of modifier keys, one reserved byte, and up to six key codes held down at once.
7. That six-key limit is why cheap keyboards cannot report many simultaneous key presses in boot protocol.

■ 13.6.4 PLAIN – what is really happening inside

1. The shape is a tiered star. The host controller is the root. Hubs form the branches. Devices are the leaves.
2. The specification allows 7 tiers counting the root, which means at most 5 hubs between the host and a device, and up to 127 addresses in total.
3. Inside a device, the things the host talks to are called endpoints. An endpoint is a buffer with a number and a direction.
4. Endpoint 0 always exists, works both ways, and is used for setup and control. Every device has it.
5. A logical connection between the host's driver and one endpoint is called a pipe.
6. There are four kinds of transfer, and a device picks one per endpoint.
7. **Control**: short, guaranteed, used for setup and commands. Endpoint 0 only.
8. **Bulk**: large amounts, correctness guaranteed, timing not guaranteed. Used by drives and printers. It gets whatever bandwidth is left over.
9. **Interrupt**: small and regular. The host promises to ask at least this often. Used by mice, keyboards and touchpads. The name is misleading: nothing interrupts anything, the host simply polls on a schedule.
10. **Isochronous**: a guaranteed slice of bandwidth every frame, with no retries. Used by webcams, microphones and audio interfaces, where a late packet is worse than a lost one.
11. Time is divided into frames of 1 millisecond at full speed, and microframes of 125 microseconds at high speed. Bandwidth is reserved inside each frame.
12. Interrupt and isochronous endpoints get their reservation first. If not enough is left, the device is refused, which is the real reason a webcam sometimes fails to start when other devices share a controller.

■ 13.6.5 TECHNICAL – the engineer's version

1. USB 1.0 was released in January 1996 by a promoter group including Compaq, DEC, IBM, Intel, Microsoft, NEC and Nortel. Ajay Bhatt led the effort at Intel. USB 1.1, in September 1998, was the version that actually worked in the field.
2. Speeds and dates:

Version	Year	Raw rate	Old marketing
1.0 Low	1996	1.5 Mbit/s	Low Speed
1.1 Full	1998	12 Mbit/s	Full Speed
2.0	2000	480 Mbit/s	High Speed
3.0	2008	5 Gbit/s	SuperSpeed
3.1	2013	10 Gbit/s	SuperSpeed+
3.2	2017	20 Gbit/s	SuperSpeed 20Gbps
USB4	2019	20 or 40 Gb/s	USB4 Gen 2 / 3
USB4 v2.0	2022	80 Gbit/s	USB 80Gbps

- The naming is a genuine mess and the USB-IF made it worse twice. When USB 3.1 arrived, plain USB 3.0 was renamed USB 3.1 Gen 1. When USB 3.2 arrived, both were renamed again to USB 3.2 Gen 1 and USB 3.2 Gen 2. So a 5 Gbit/s port has had three official names for the same silicon.
- As of 2026 the USB-IF's guidance is to drop the version numbers in consumer labelling and use speed names only: USB 5Gbps, USB 10Gbps, USB 20Gbps, USB 40Gbps and USB 80Gbps. Product pages still use the old names, so expect both.
- Mapping: USB 3.2 Gen 1x1 is 5 Gbit/s, Gen 2x1 is 10 Gbit/s, Gen 2x2 is 20 Gbit/s using two lanes. USB4 Gen 3x2 is 40 Gbit/s and USB4 Gen 4x2 is 80 Gbit/s. USB4 Version 2.0 also defines an asymmetric mode of 120 Gbit/s one way and 40 Gbit/s the other, for high-resolution displays.
- Encoding overhead is real. USB 3.0 at 5 Gbit/s uses 8b/10b, so the usable ceiling is 500 MB/s before protocol overhead, and real drives reach around 400 to 450 MB/s.
- Power. USB 2.0 defines a unit load of 100 mA at 5 V, with up to 5 units, so 500 mA or 2.5 W. USB 3.x raises the unit load to 150 mA with up to 6 units, so 900 mA or 4.5 W.
- USB Power Delivery negotiates far more. PD revision 1.0 reached 100 W. PD revision 3.1, published in 2021, added Extended Power Range with 28 V, 36 V and 48 V fixed levels, reaching 240 W at 48 V and 5 A. A cable must be electronically marked to carry 5 A.
- USB Type-C is a **connector**, defined in a specification finalized in August 2014, with 24 pins and reversible insertion. It is not a speed and not a protocol.
- This is the single most common confusion in the whole subject. A USB-C socket may run USB 2.0 at 480 Mbit/s and nothing else, or it may carry USB 80Gbps plus DisplayPort plus PCIe plus 240 W. The shape tells you nothing. Only the logo and the specification sheet tell you.
- Device classes are defined by the USB-IF so that one driver serves many products. Class 1 audio, class 3 HID, class 6 still image, class 7 printer, class 8 mass storage, class 9 hub, class 14 video, class 255 vendor-specific.
- HID, defined in 1996 and revised as HID 1.11 in 2001, is why a keyboard from any maker works with no download. It uses a self-describing report descriptor, so the device tells the host the layout of its own data.
- Boot protocol is the cut-down fixed 8-byte HID keyboard format that firmware can drive before any operating system loads. That is why your keyboard works in the BIOS setup screen.
- Vendor and product IDs are 16-bit each and assigned by the USB-IF. For example 0x046D is Logitech.
- Observe on Linux with `lsusb -t` for the topology tree and `lsusb -v` for descriptors. On macOS, `system_profiler SPUSBDataType`. On Windows, Device Manager with "Devices by connection" selected.

```
/: Bus 01.Port 1: Dev 1, Class=root_hub, 480M
   |-- Port 2: Dev 3, If 0, Class=HID, Driver=usbhid, 1.5M
   |-- Port 4: Dev 5, If 0, Class=Mass Storage, 480M
```

■ 13.6.6 WORDS – remember these

1. Host controller — the boss chip on the machine — the xHCI controller issuing all transactions.
2. Hub — a splitter — a device that repeats host traffic to more ports and reports attachments.
3. Endpoint — a numbered buffer in a device — a source or sink with a direction and transfer type.
4. Pipe — a connection to one endpoint — the logical channel between host driver and endpoint.
5. Enumeration — finding out what was plugged in — reset, address assignment, descriptor read and configuration.
6. Descriptor — the device's self-description — the byte structures giving IDs, classes, endpoints and strings.
7. Device class — a family with a standard driver — the interface class code that selects a generic driver.
8. USB-C — the oval reversible plug — a 24-pin connector specification, not a speed or a protocol.

13.7 The other ports on the machine

■ 13.7.1 PLAIN – in simple words

1. Not every socket on a machine is USB. Each remaining one exists because it does something USB did not do well enough at the time.
2. Thunderbolt looks exactly like USB-C but can carry far more, including a graphics card's connection and a screen's picture at the same time.
3. HDMI and DisplayPort carry picture and sound to a screen.
4. The square-ish clicky socket is Ethernet, for a network cable.
5. The small round holes are audio jacks, for headphones and microphones.
6. The thin slot is for SD cards, used mostly by cameras.
7. The old wide sockets, serial and parallel, have almost gone from home machines but are alive in factories and laboratories.

■ 13.7.2 PLAIN – a picture in your head

1. Think of USB as a general delivery van that handles most parcels.
2. Thunderbolt is a motorway tunnel. Other roads drive through it unchanged and come out the other end still themselves.
3. That is the key idea: Thunderbolt does not convert the traffic. It wraps it, carries it, and unwraps it.
4. So a graphics card in a box on your desk really is talking to the processor over its normal card connection. The cable just carries it.

Where this comparison breaks: a tunnel has fixed lanes. Thunderbolt shares one budget between video and data, so a very high resolution screen leaves less room for the drive plugged into the same dock.

■ 13.7.3 PLAIN – a worked example

1. You plug a laptop into one dock with a single USB-C cable, and get a 4K screen, a wired network, three USB ports and charging.
2. Inside the cable there is one Thunderbolt link.
3. The screen's picture rides inside it as DisplayPort.
4. The network chip and the USB ports on the dock sit behind a PCI Express switch, and that PCI Express traffic also rides inside it.
5. Power travels on separate power pins, negotiated by USB Power Delivery.

6. Take the same cable and plug in a second 4K screen at 120 Hz. Something will drop in quality or refresh rate, because the video share of the budget is now full.

■ 13.7.4 PLAIN – what is really happening inside

1. Thunderbolt builds a small switched network in the cable. Each end has a controller chip that packs, routes and unpacks.
2. Because it tunnels PCI Express, a Thunderbolt device is, to the operating system, a card plugged into the machine. This is why external graphics boxes work and also why Thunderbolt security matters.
3. HDMI and DisplayPort differ in origin. HDMI grew from television and carries a clocked stream of pixels. DisplayPort grew from computing and sends packets, which is why it can drive several screens down one cable.
4. Ethernet's eight-pin socket uses two or four twisted pairs. Twisting the pairs is what cancels interference over 100 metres of cable.
5. A headphone jack with three black bands has four conductors: left, right, ground and microphone. That is what TRRS means: tip, ring, ring, sleeve.
6. An SD card is a small controller chip plus flash memory. The card, not the camera, decides where data physically goes.

■ 13.7.5 TECHNICAL – the engineer's version

1. Thunderbolt began as Intel's Light Peak research and shipped in February 2011 on the MacBook Pro with a Mini DisplayPort connector.

Version	Year	Bandwidth	Connector
TB 1	2011	10 Gbit/s	Mini DisplayPort
TB 2	2013	20 Gbit/s	Mini DisplayPort
TB 3	2015	40 Gbit/s	USB-C
TB 4	2020	40 Gbit/s	USB-C
TB 5	2023	80 Gbit/s	USB-C

2. Thunderbolt 4 did not raise the headline speed over Thunderbolt 3. It raised the minimum requirements: 32 Gbit/s of PCIe tunnelling, two 4K displays, and mandatory DMA protection through the IOMMU.
3. Thunderbolt 5, announced by Intel in September 2023 with the controller codenamed Barlow Ridge, provides 80 Gbit/s in both directions and an asymmetric Bandwidth Boost mode of 120 Gbit/s out and 40 Gbit/s back. It tunnels PCIe 4.0 x4, about 64 Gbit/s. It is built on USB4 Version 2.0.
4. DMA protection is not optional history. The Thunderspy work published in 2020 showed pre-2019 Thunderbolt machines could be compromised through direct memory access from a malicious device.
5. Display standards: HDMI 1.0 in 2002; HDMI 2.1 in 2017 at 48 Gbit/s; HDMI 2.2, announced at CES in January 2025 and published in mid-2025, at 96 Gbit/s with "Ultra96" certified cables. DisplayPort 1.0 came from VESA in 2006; DisplayPort 2.0 in 2019 defines UHBR20 at 80 Gbit/s raw; DisplayPort 2.1b was announced alongside HDMI 2.2 with DP80LL cables.
6. HDMI is licensed by HDMI Licensing Administrator with per-unit royalties. DisplayPort is royalty-free from VESA. That commercial difference, not a technical one, is why televisions use HDMI and monitors often use both.

7. Ethernet on copper: 10BASE-T from 1990, 100BASE-TX from 1995, 1000BASE-T from IEEE 802.3ab in 1999, 10GBASE-T from 802.3an in 2006, and 2.5GBASE-T and 5GBASE-T from 802.3bz in 2016. The connector is an 8P8C modular plug, almost always called RJ45 by convention rather than by standard.
8. Audio jacks are 3.5 mm. TRS is three conductors and stereo out. TRRS is four and adds a microphone. Two incompatible TRRS wirings exist: CTIA, used by nearly everything today, and OMTP, used by some older phones, with microphone and ground swapped. The colour code, green for line out and pink for microphone, comes from the Microsoft and Intel PC 99 guide.
9. SD cards were introduced in August 1999 by SanDisk, Panasonic and Toshiba, with the SD Association formed in January 2000. Capacity tiers: SDSC to 2 GB, SDHC to 32 GB from 2006, SDXC to 2 TB from 2009, SDUC to 128 TB from 2018.
2018. Bus modes run from 12.5 MB/s default speed through UHS-I at 104 MB/s and UHS-II at 312 MB/s to SD Express, which puts PCIe and NVMe on the card and reaches into the gigabytes per second.
10. Serial: the EIA RS-232 standard dates from 1960. The PC/AT gave it a DE-9 plug in 1984. COM1 sits at port 0x3F8 on IRQ 4. It survives on network switch consoles, laboratory instruments, industrial controllers and embedded debug headers, usually now through a USB-to-serial adapter using an FTDI or Silicon Labs chip.
11. Parallel: the Centronics printer interface, standardized as IEEE 1284 in 1994, at port 0x378 on IRQ 7. It survives on old industrial machines and, for hobbyists, as a source of simple bit-banged input and output.

■ 13.7.6 WORDS – remember these

1. Tunnelling — carrying one protocol inside another — encapsulating PCIe and DisplayPort inside Thunderbolt or USB4 transport.
2. Thunderbolt — the high-capacity USB-C link — an Intel interconnect tunnelling PCIe and DisplayPort, now aligned with USB4.
3. TRRS — the four-part headphone plug — tip, ring, ring, sleeve, adding a microphone conductor, in CTIA or OMTP wiring.
4. 8P8C — the network socket — the eight-position eight-contact modular connector commonly called RJ45.
5. UHS — the faster SD card mode — Ultra High Speed bus modes I, II and III.

13.8 What a driver actually is

■ 13.8.1 PLAIN – in simple words

1. A driver is a piece of code that translates.
2. On one side it accepts a general request that says nothing about hardware, such as “write these bytes”.
3. On the other side it produces the exact numbers that one particular chip needs, written into that chip’s exact registers, in that chip’s exact order.
4. That is the whole job. A translator between the general and the specific.
5. It exists because there are thousands of different chips that do the same job in different ways.
6. Without drivers, every program would need code for every chip ever made.
7. With drivers, a program says “write these bytes” and never learns what kind of drive it is.
8. A driver is not a program you run. It is code the operating system loads and calls when needed.

9. Most drivers live inside the operating system's core, so a fault in one can bring down the whole machine.
10. The operating system chooses which driver to load by matching numbers the device reports about itself.

■ 13.8.2 PLAIN – a picture in your head

1. Imagine an international company where head office issues one instruction: “ship this box to the customer”.
2. Each country's office knows its own local rules, forms, languages and couriers.
3. Head office never learns those rules. It only knows the one instruction.
4. Each local office is a driver. It turns one general instruction into the specific local procedure.
5. Replace a courier in one country and only that one office changes. Head office is untouched.
6. Now imagine a local office that files a form wrongly and locks the whole company's systems. That is what a bad kernel driver does.

Where this comparison breaks: a local office can refuse an instruction it does not understand. A driver is usually trusted, running with the same power as the operating system core, so it is believed even when it is wrong.

■ 13.8.3 PLAIN – a worked example

1. Follow one key press from finger to screen. This is the exact chain.

```
finger presses key
-> keyboard matrix scan in keyboard's own chip
-> USB HID report, 8 bytes, on an interrupt endpoint
-> host controller (xHCI) driver takes the transfer
-> USB core hands the report to the matching driver
-> HID class driver decodes the report descriptor
-> input subsystem turns it into a key event
-> keymap turns key code into a character
-> window system sends the character to the focused app
-> the app draws the letter
```

2. Now follow writing a file. The chain runs the other way.

```
app calls write(fd, buf, 4096)
-> system call: user mode traps into the kernel
-> VFS picks the filesystem for that mount
-> ext4 or APFS or NTFS turns it into block writes
-> block layer queues and merges the requests
-> NVMe class driver builds a command in a queue
-> PCIe driver has already mapped the device's BARs
-> doorbell register write tells the SSD to look
-> SSD firmware does wear levelling and writes flash
-> completion queue entry plus MSI-X interrupt
-> the kernel wakes the waiting application
```

3. Note how many layers are generic and how few are device-specific.
4. Only two boxes in each chain know what chip this actually is.

■ 13.8.4 PLAIN – what is really happening inside

1. The layers, from top to bottom, always look like this.
2. **Application:** a normal program. Knows nothing about hardware.
3. **System call:** the doorway into the kernel. The processor switches from user mode to kernel mode.

4. **Kernel subsystem:** the generic manager for a category, such as the file layer, the network stack, or the input layer. Defines the general request.
5. **Class driver:** code for a whole family of devices that follow one standard, such as USB mass storage or NVMe. It knows the standard, not the brand.
6. **Bus driver:** code that knows how to find and reach devices on a particular bus, such as PCIe or USB. It enumerates devices, reads their IDs, and hands out address windows and interrupts.
7. **Device driver:** code for one particular chip or chip family. It knows the register map, the quirks, and the initialization sequence.
8. **Hardware:** the chip itself.
9. Not every chain has every layer. A well-standardized device may need no vendor-specific driver at all, because the class driver is enough.
10. That is why a USB stick works instantly and a graphics card does not: the stick follows a class standard, the graphics card is a custom machine with a proprietary register map.
11. Requests travel down and completions travel up. Down is usually a call. Up is usually an interrupt followed by a callback.

■ 13.8.5 TECHNICAL – the engineer’s version

1. A driver is defined by the interface it implements, not by what it contains. On Linux it registers a struct `file_operations` or a bus-specific struct `pci_driver` with probe and remove callbacks.
2. Minimal shape of a Linux PCI driver:

```
static struct pci_device_id ids[] = {
    { PCI_DEVICE(0x8086, 0x100e) }, /* Intel 82540EM */
    { 0, }
};
MODULE_DEVICE_TABLE(pci, ids);

static struct pci_driver drv = {
    .name      = "mydrv",
    .id_table  = ids,
    .probe     = my_probe,
    .remove    = my_remove,
};
```

3. probe is called by the bus core once a device matching the ID table is found. It maps BARs with `pci_iomap`, requests an interrupt with `pci_alloc_irq_vectors` and `request_irq`, sets bus mastering with `pci_set_master`, and registers with the relevant subsystem.
4. Vendor ID 0x8086 is Intel, chosen as a reference to the 8086 processor. 0x10DE is NVIDIA, 0x1002 is AMD graphics, 0x10EC is Realtek.
5. Windows layers the same idea as a device stack of driver objects: a bus driver creates a physical device object, and function and filter drivers attach device objects above it. Requests travel as IRPs (IO request packets) down the stack.
6. macOS and iOS use IOKit, an object-oriented C++ driver framework with a matching dictionary per driver, and a live registry visible with `ioreg`.
7. The system call boundary is the hard security line. On x86-64 Linux the instruction is `syscall`; on ARM64 it is `svc #0`. Everything above it runs unprivileged; everything below can touch any memory.
8. Timings that explain the layering, order of magnitude only:

Step	Typical cost
Function call	About 1 nanosecond
System call	50 to 400 nanoseconds
Interrupt entry/exit	Around 1 microsecond
NVMe read completion	20 to 100 microsecond

9. This cost table is why fast devices use polling again in some paths. Linux NAPI polls a busy network card instead of taking an interrupt per packet, and `io_uring`, merged in Linux 5.1 in 2019, exists to avoid system calls per operation. Polling returned not because interrupts got worse but because devices got faster than interrupt handling.
10. Observe the stack on Linux with `lsmod`, `modinfo nvme`, and `ls /sys/bus/pci/drivers/`. On Windows, `pnputil /enum-drivers` and the Device Manager driver tab. On macOS, `kmutil showloaded` and `ioreg -l`.

■ 13.8.6 WORDS – remember these

1. Driver — the translator for one device — code implementing a kernel interface and driving a specific register map.
2. Class driver — one driver for a whole family — a driver implementing a device class standard such as HID or NVMe.
3. Bus driver — the code that finds devices — enumerates a bus, reads IDs and allocates resources.
4. System call — the doorway into the kernel — a controlled privilege transition such as `syscall` or `svc`.
5. Probe — the moment a driver claims a device — the callback the bus core makes when an ID matches.
6. Device stack — the chain of code over one device — layered driver objects passing requests down and completions up.

13.9 Driver mechanics

■ 13.9.1 PLAIN – in simple words

1. Code on a machine runs in one of two worlds.
2. In user space, code is fenced in. If it goes wrong, only it dies.
3. In kernel space, code can touch everything. If it goes wrong, the machine stops.
4. Most drivers live in kernel space, because they need direct access to hardware and must respond in microseconds.
5. Modern systems move as many drivers as possible into user space, giving up a little speed for a lot of safety.
6. Drivers are usually not built into the operating system. They are separate files loaded when a matching device appears.
7. Because a driver has total power, operating systems now refuse to load one unless it carries a valid digital signature.
8. The signature does not prove the driver is good. It proves who made it and that nobody changed it since.
9. Each operating system has its own driver framework, with different names for the same ideas.
10. Matching is done by numbers: the device says “I am vendor 0x8086, product 0x100E”, and the system looks for a driver that claims that pair.

■ 13.9.2 PLAIN – a picture in your head

1. Think of a hospital. Visitors stay in the public corridors. Surgeons enter the operating theatre.
2. User space is the corridor. A visitor who faints causes a small incident.
3. Kernel space is the theatre. A surgeon who faints mid-operation is a disaster for the patient.
4. So hospitals check credentials at the theatre door, and keep as much work as possible out in the corridors.
5. Driver signing is the credential check. It says this person is who they claim, and their badge has not been forged.
6. It does not say they are a competent surgeon.

Where this comparison breaks: a hospital can eject a bad surgeon mid- operation. An operating system usually cannot stop a kernel driver that has already corrupted memory. By the time the fault is visible, the damage is done.

■ 13.9.3 PLAIN – a worked example

1. On Linux, you plug in a USB-to-serial adapter. Watch what happens.

```
$ lsusb
Bus 001 Device 007: ID 0403:6001 Future Technology
    Devices International, Ltd FT232 Serial (UART) IC

$ dmesg | tail -3
usb 1-2: new full-speed USB device number 7
usbcore: registered new interface driver ftdi_sio
usb 1-2: FTDI USB Serial Device now attached to ttyUSB0

$ ls -l /dev/ttyUSB0
crw-rw---- 1 root dialout 188, 0 Aug 13 10:22 /dev/ttyUSB0
```

2. Read that carefully. 0403 is the vendor ID for FTDI. 6001 is the product ID for the FT232 chip.
3. The kernel formed a modalias string from those numbers, looked it up, and loaded the module named ftdi_sio.
4. The module then created a device node, /dev/ttyUSB0.
5. 188, 0 are the major and minor numbers. 188 identifies the driver, 0 identifies which of its devices this is.
6. From now on, any program can just open /dev/ttyUSB0 like a file. It never needs to know the chip is an FT232.

■ 13.9.4 PLAIN – what is really happening inside

1. A loadable module is compiled machine code with a table of symbols it needs and a table of what it provides.
2. Loading it means: copy it into kernel memory, resolve its references to kernel functions, run its init function, and register it with the bus core.
3. The bus core then walks its list of known devices and calls the new driver's probe function for each match.
4. Unloading runs the reverse, but only if nothing is using the driver.
5. A kernel driver shares one address space with the whole kernel. There is no memory protection between it and everything else.
6. So a wild pointer write in a driver silently corrupts unrelated kernel data, and the crash may appear minutes later somewhere else entirely.

7. A user-space driver runs as an ordinary restricted process. It reaches hardware only through a narrow, checked path the kernel opens for it.
8. If it crashes, the kernel closes the handle, resets the device, and can restart the driver. The machine survives.
9. The trade is latency and copies. Every crossing between user space and the kernel costs time, so the fastest devices keep kernel drivers.

■ 13.9.5 TECHNICAL – the engineer’s version

1. **Windows.** WDM (Windows Driver Model), introduced with Windows 98 and Windows 2000, is the low-level model of IRPs and device objects. It is powerful and unforgiving.
2. WDF (Windows Driver Frameworks) sits above WDM and handles the hard parts, chiefly power management and plug-and-play state. It comes in two forms: KMDF (Kernel-Mode Driver Framework) and UMDF (User-Mode Driver Framework). As of Windows 11 the shipping versions are KMDF 1.33 and UMDF 2.33. Microsoft publishes the framework source on GitHub.
3. UMDF version 2 uses the same object model and function names as KMDF, so a driver can be moved between kernel and user mode with modest changes. UMDF version 1, which used COM, is superseded.
4. Windows drivers are described by an INF text file listing hardware IDs, which look like `PCI\VEN_10DE&DEV_2484&SUBSYS_...&REV_A1` or `USB\VID_046D&PID_C52B`. Matching is by longest, most specific ID first.
5. Windows driver signing history: Windows Vista x64 in 2006 first required kernel-mode drivers to be signed. Since Windows 10 version 1607, released in August 2016, new kernel-mode drivers must be signed by Microsoft through the hardware developer portal, submitted with an EV certificate, either as WHQL-certified or attestation-signed.
6. **Linux.** Drivers are usually loadable kernel modules, files ending in `.ko`, loaded with `modprobe` or automatically by `udev`. `lsmod` lists them; `modinfo` shows their aliases and parameters.
7. Character and block devices appear as nodes under `/dev`, created automatically by `devtmpfs` and managed by `udev`, in practice `systemd-udevd` on most distributions. Rules live in `/etc/udev/rules.d` and `/lib/udev/rules.d`.
8. `/sys`, the `sysfs` filesystem introduced in Linux 2.6 in 2003, exposes the whole device model as directories: `/sys/bus/pci/devices`, `/sys/class/net`, `/sys/devices/...` It is how userspace discovers hardware without `ioctl`s.
9. Matching uses `modalias` strings. Read one directly:


```
$ cat /sys/bus/usb/devices/1-2/modalias
usb:v0403p6001d0600dc00dsc00dp00ic FFisc FFip FFin00
```
10. Linux has no stable in-kernel driver ABI, by deliberate policy. Drivers are expected to be merged into the mainline tree, where they are updated with the rest of the kernel. Out-of-tree drivers, such as NVIDIA’s, must be rebuilt for each kernel, usually through DKMS.
11. Experts genuinely disagree here. Kernel developers argue the unstable ABI forces good code into the tree and lets interfaces improve. Vendors argue it makes proprietary drivers painful. Both are describing the same fact, valuing it differently.
12. Linux also supports user-space drivers: FUSE for filesystems, `libusb` and `usbfs` for USB devices, VFIO and UIO for direct device access with IOMMU protection, and SPDK and DPDK for polled user-space storage and network drivers.
13. **macOS.** Kexts (kernel extensions) built on IOKit were the traditional model. Apple announced their deprecation at WWDC in June 2019 and introduced DriverKit, which builds driver code as a user-space system extension using the same IOKit-style class model.

14. On Apple silicon Macs, loading a kext requires switching to Reduced Security in recoveryOS and approving it, then rebooting. As of 2026 kexts still load on macOS 26 under that reduced security setting, but Apple's documented guidance is to use DriverKit and system extensions.
15. Inspect on macOS with `kmutl showloaded, systemextensionsctl list` and `ioreg -l`.

System	Kernel driver	User-space driver
Windows	WDM, KMDF	UMDF 2
Linux	Kernel module .ko	FUSE, libusb, VFIO
macOS	Kext (IOKit)	DriverKit extension

16. The honest version: “user-space driver” rarely means no kernel code at all. A small kernel stub still owns the interrupt and the IOMMU mapping. What moves out is the large, complex, bug-prone part.

■ 13.9.6 WORDS - remember these

1. Kernel space — where trusted code runs — the privileged address space with no memory protection between components.
2. Kernel module — a driver file loaded on demand — a relocatable object linked into the running kernel.
3. Device node — the file that represents a device — a character or block special file with major and minor numbers.
4. `sysfs` — the folder view of hardware — the kernel object hierarchy exported under `/sys`.
5. `udev` — the thing that creates device files — the userspace device manager acting on kernel uevents.
6. Driver signing — a proof of who wrote it — a cryptographic signature the loader verifies against trusted keys.
7. DriverKit — Apple's safer driver system — a user-space system extension framework replacing kexts.
8. Hardware ID — the number that picks the driver — a vendor and product ID string used for driver matching.

13.10 Firmware: code that lives on the device

■ 13.10.1 PLAIN - in simple words

1. Firmware is software that lives inside a piece of hardware rather than on your disk.
2. It runs on a small processor inside the device itself, not on your main processor.
3. It is there before your operating system starts, and it keeps running after your operating system stops.
4. Almost every device has some. A drive, a keyboard, a network chip, a screen, a mouse, a charger.
5. It is called firm-ware because it sits between hardware, which cannot change, and software, which changes all the time.
6. Long ago it truly could not be changed. Today it is stored in flash memory and can be rewritten.
7. That is useful, because bugs in firmware are common and often serious.
8. It is also dangerous, because a failed rewrite can leave a device that cannot start at all.

■ 13.10.2 PLAIN – a picture in your head

1. Think of a vending machine. The metal box is the hardware.
2. The instructions inside its controller, deciding what happens when you press B4, are the firmware.
3. The price list you load in is data. The app you use to pay is software on your phone.
4. Changing the firmware changes what the machine does, permanently, even with nobody around.
5. And if the engineer's update fails halfway, the machine will not open its door for anyone, because the instructions are now half old and half new.

Where this comparison breaks: a broken vending machine can be opened with a key. Many devices have no recovery route once their only firmware image is damaged, which is what “bricked” means.

■ 13.10.3 PLAIN – a worked example

1. Here is where firmware sits in one ordinary laptop.

Device	Firmware does what
Motherboard flash	Starts the machine
SSD controller	Wear levelling, mapping
Keyboard chip	Scans the key matrix
WiFi module	Runs the radio timing
GPU (VBIOS)	Powers up, sets display
Monitor scaler	Scaling, menus, inputs

2. Count them. That is six separate processors running six separate programs, before your operating system even loads.
3. A modern laptop typically contains between ten and thirty small processors you never see.

■ 13.10.4 PLAIN – what is really happening inside

1. Firmware is stored in non-volatile memory, usually NOR flash for code that must run in place, or NAND flash with a small loader.
2. When power arrives, the device's own small processor starts at a fixed address inside that flash and begins executing.
3. Some devices do not store their firmware at all. They start empty, and the driver on your machine uploads the firmware image at every boot.
4. That is why some WiFi and graphics cards need a firmware package installed separately from the driver, and do nothing without it.
5. Updating firmware means erasing a block of flash and writing a new image, then verifying it.
6. If power is lost between the erase and the verify, the device has no valid image, and the small processor has nothing valid to run.
7. Careful designs avoid this with two images: write to the spare slot, verify it, then flip a pointer. If anything fails, the old image is untouched.

■ 13.10.5 TECHNICAL – the engineer's version

1. Firmware update mechanisms in use today: UEFI capsule updates for system firmware, vendor tools for SSD and GPU firmware, DFU (device firmware upgrade) class for USB devices, and I2C or DDC/CI for monitor scalers.
2. On Linux, fwupd with the Linux Vendor Firmware Service delivers signed firmware for participating vendors. Commands: `fwupdmgr get-devices` and `fwupdmgr update`. On Windows, firmware

arrives through Windows Update as a firmware driver package. On macOS, firmware ships inside system updates.

3. Real firmware bugs with real consequences. In November 2019 Hewlett Packard Enterprise issued an urgent bulletin: certain SAS solid state drives would fail permanently, with total data loss, at exactly 32,768 hours of power-on time. That number is 2 to the power 15, a signed 16-bit counter overflowing.
4. Samsung's 840 EVO drives, from 2013, slowed dramatically when reading old data, and the fix was a firmware change to the drive's read-retry and refresh behaviour, shipped in 2014 and again in 2015.
5. Intel and AMD ship processor microcode updates, which are firmware for the processor itself, loaded by the platform firmware or by the operating system at every boot. The Spectre and Melt-down mitigations of 2018 arrived partly this way.
6. Firmware is a security surface. It runs before any operating system protection exists, and it survives reinstalling the operating system. Attacks that persist in firmware are the reason for Boot Guard, signed capsules and measured boot into a TPM.
7. Redundancy schemes: dual-BIOS boards with a backup chip, A/B slots on phones and network gear, and USB BIOS Flashback, which rewrites the flash using a small always-on controller with no processor or memory installed.
8. The plain difference, stated once and clearly:

Thing	Where it lives	Who runs it
Firmware	Flash on the device	The device's chip
Driver	OS kernel or system	Your main CPU
Software	Your disk	Your main CPU

9. The honest version: the boundary is not sharp. A GPU driver contains blobs that get uploaded to the card and run there, which makes them firmware delivered by a driver. Classification follows what runs where, not which file it arrived in.

■ 13.10.6 WORDS - remember these

1. Firmware — code inside the device — non-volatile program executed by an embedded controller.
2. Flashing — rewriting firmware — erasing and reprogramming non-volatile memory with a new image.
3. Bricked — dead after a failed update — an unbootable device with no valid firmware image and no recovery path.
4. Microcode — firmware for the processor — patches to the CPU's internal instruction decoding, loaded at boot.
5. DFU — the standard USB update mode — Device Firmware Upgrade class.
6. A/B slots — two copies of the firmware — redundant images with an atomic pointer switch for safe updates.

13.11 BIOS and UEFI

■ 13.11.1 PLAIN - in simple words

1. When you press power, your processor has no operating system, no drivers, and no idea what is attached.
2. Something must wake it, test the basics, find a disk, and load the first piece of the operating system.

3. That something is the platform firmware, stored on the small flash chip on the board.
4. The old version, from 1981, was called BIOS, short for Basic Input Output System.
5. The modern replacement is called UEFI, short for Unified Extensible Firmware Interface.
6. BIOS was simple and limited: it loaded 512 bytes from the start of a disk and jumped into them.
7. UEFI is far larger. It can read a real filesystem, keep a list of things it can boot, and check signatures before running them.
8. Both do the same job. Only UEFI can handle large disks, many boot options and modern security.

■ 13.11.2 PLAIN – a picture in your head

1. BIOS is a caretaker with one instruction: go to the front desk of the building, read the note pinned there, and do whatever it says.
2. The note is tiny, so it usually says “go to room 4 and read the longer note there”.
3. UEFI is a caretaker with a real address book. It knows several buildings, remembers which one you preferred, and checks each visitor’s identity card before letting them in.

Where this comparison breaks: the identity check, Secure Boot, only verifies the visitor at the door. Once the operating system kernel is running, it can do whatever it likes. Secure Boot does not keep watching.

■ 13.11.3 PLAIN – a worked example

1. Here is the first second after you press the power button.

```

0 ms    power supply stabilizes, asserts POWER_GOOD
        the CPU is held in reset until then
~1 ms   reset released; CPU starts at its reset
        vector, the top of the address space
~2 ms   firmware runs from flash; no RAM works yet
        so cache is used as temporary memory
10-2000 memory training: the controller tunes the
ms      timing of every DDR5 signal, then tests RAM
then    POST: check CPU, RAM, basic chips
then    enumerate PCIe and USB, assign addresses
then    read boot order from NVRAM settings
then    load the boot file from the EFI partition
then    hand control to the bootloader

```

2. Memory training is why a machine with new DDR5 memory can appear dead for thirty seconds on its very first boot, and start in three seconds after.
3. The result of the training is cached, so it only happens once.

■ 13.11.4 PLAIN – what is really happening inside

1. POST means power-on self test. The firmware checks the processor, tests enough memory to work, and initializes the basic chips.
2. If it fails before a screen exists, it must report through beeps, board LEDs or a two-digit code display. That is why beep codes exist.
3. Then it enumerates buses. It walks every PCIe and USB port, reads the ID of whatever is there, and assigns address windows and interrupts.
4. Then it looks for something to boot.
5. Old BIOS did this by reading the first 512 bytes of each disk in a configured order, checking for the two marker bytes 0x55 0xAA at the end, and jumping into that code.
6. 512 bytes is almost nothing, so that code’s only job was to find and load a bigger loader.

7. UEFI does it differently. It reads a real FAT-formatted partition on the disk, called the EFI System Partition, and runs a normal executable file from it.
8. UEFI also keeps a list of boot entries in its own non-volatile memory, each naming a device and a file path, plus an order to try them in.
9. Settings and the clock survive power loss because of the coin cell. Remove it and the settings return to defaults, which is the standard fix for a forgotten firmware password on older boards.

■ 13.11.5 TECHNICAL – the engineer’s version

1. The IBM PC of 1981 shipped an 8 KB BIOS ROM, and IBM printed its full assembly listing in the technical reference manual. Compaq in 1982 and Phoenix Technologies in 1984 produced clean-room compatible versions, which is what created the PC clone industry.
2. BIOS ran in 16-bit real mode with a 1 MiB address space and offered services through software interrupts, notably INT 13h for disk and INT 10h for video.
3. The replacement effort began at Intel in 1998 as the Intel Boot Initiative, for Itanium, becoming EFI. Intel stopped at EFI 1.10 in 2005 and gave it to the Unified EFI Forum. UEFI 2.0 was published on 31 January

2006. Version 2.11 was published in December 2024.

4. UEFI runs in 32-bit or 64-bit mode, has a driver model of its own, a shell, and services split into boot services and runtime services. Runtime services survive into the running operating system, which is how a system can set its own next boot entry.
5. The EFI System Partition is FAT12, FAT16 or FAT32, conventionally at least 100 MB and often 512 MB, with GUID type C12A7328-F81F-11D2-BA4B- 00A0C93EC93B. The default fallback path is \EFI\BOOT\BOOTX64.EFI.
6. Boot entries are UEFI variables named Boot0000, Boot0001 and so on, with BootOrder listing the sequence and BootCurrent naming the one used.

```
$ efibootmgr -v
BootCurrent: 0002
BootOrder: 0002,0000,0001
Boot0000* Windows Boot Manager  HD(1,GPT,...)/File(\EFI\...)
Boot0002* ubuntu  HD(1,GPT,...)/File(\EFI\ubuntu\shimx64.efi)
```

7. On Windows the equivalent tool is `bcdedit /enum firmware`.
8. GPT (GUID Partition Table) is part of the UEFI specification. MBR with 512 byte sectors cannot address beyond 2 TiB; GPT uses 64-bit LBAs, keeps a backup table at the end of the disk, protects both with CRC32, and supports 128 entries by default. The UEFI specification requires firmware to support GPT; booting from MBR was handled by the Compatibility Support Module, which Intel removed from client platforms around 2020.
9. Secure Boot was added in UEFI 2.3.1, released on 6 April 2011, and enforced on Windows 8 logo machines from 2012. It verifies the signature of each executable the firmware loads against a key hierarchy: PK, the platform key; KEK, key exchange keys; db, allowed signatures; and dbx, the revocation list.
10. What Secure Boot actually verifies: the bootloader, the firmware drivers it loads, and, if the operating system continues the chain, the kernel. What it does not verify: your hardware, your applications, your data, or anything after the kernel takes over. It is a chain of trust for boot, not an antivirus.
11. Linux distributions boot under Secure Boot through shim, a small loader signed by Microsoft’s third-party UEFI CA, which then verifies the distribution’s own key.

12. Current and dated: Microsoft's 2011-generation Secure Boot certificates reach the end of their validity through 2026, and are being replaced by 2023-dated certificates via firmware and operating system updates. If you read this after 2026, check the current state rather than trusting this line.
13. Chapter 18 picks the story up here: from the moment the bootloader is running and begins to load an operating system kernel.

■ 13.11.6 WORDS - remember these

1. POST — the start-up self test — power-on self test of CPU, memory and core chipset before boot.
2. Reset vector — the first instruction address — the fixed address the CPU fetches from after reset.
3. BIOS — the old start-up firmware — 16-bit real-mode firmware with INT-based services and MBR booting.
4. UEFI — the modern replacement — a specification with its own executables, FAT system partition and boot variables.
5. ESP — the boot partition — the EFI System Partition holding UEFI executables.
6. GPT — the modern partition table — GUID Partition Table with 64-bit addressing and CRC-protected redundant headers.
7. Secure Boot — signature checking at boot — verification of loaded images against the PK, KEK, db and dbx key stores.

13.12 Plug and play, resources and conflicts

■ 13.12.1 PLAIN - in simple words

1. Every device needs three things reserved for it: an address range, an interrupt number, and sometimes a data-moving channel.
2. If two devices are given the same one, both misbehave, often in confusing ways.
3. In the 1980s and early 1990s you set these by hand, by moving tiny plastic caps called jumpers on the card itself.
4. You had to know what everything else in the machine already used, and read the manual for each card.
5. Plug and play was the promise that the machine would work this out by itself.
6. It works by having each device describe what it needs, rather than demand a fixed setting.
7. The firmware and the operating system then assign resources so nothing overlaps.
8. Today conflicts are rare, and when they appear they are almost always a driver problem rather than a real hardware clash.

■ 13.12.2 PLAIN - a picture in your head

1. Old cards were like guests who each insisted on a specific seat number printed on their own ticket.
2. If two guests printed the same number, you had to reprint one ticket by hand before the meal could start.
3. Plug and play is guests who say "I need one seat near a window" and let the host do the seating plan.
4. The host has the full list and can always find an arrangement.

Where this comparison breaks: a host can seat anyone anywhere. Some hardware still only works at fixed addresses, so the seating plan has permanent reserved places it must work around.

■ 13.12.3 PLAIN – a worked example

1. A 1993 machine with a sound card and a network card, set by jumpers.

Device	IO port	IRQ	DMA
Sound Blaster	0x220	5	1
Network card	0x300	10	-
LPT1 printer	0x378	7	-
COM1 serial	0x3F8	4	-

2. Set the network card to IRQ 5 by mistake and both cards break: the sound stutters and the network drops, because two devices pull one shared line.
3. The fix was physical. Power off, open the case, move a jumper, close, boot, and test.
4. Today the same information is assigned automatically at every boot, and can change between boots without anything breaking.

■ 13.12.4 PLAIN – what is really happening inside

1. A PCI or PCIe device has a small configuration space that anyone can read before the device is configured.
2. It contains the vendor ID, product ID, class code, and a set of base address registers describing how much address space it wants.
3. The firmware reads all of these, builds a map with no overlaps, and writes the chosen addresses back into the device.
4. Interrupts are assigned the same way, and with message-signalled interrupts there is no physical line to share at all, so classic conflicts simply cannot happen.
5. ACPI adds a description of everything that is not discoverable: which interrupt the embedded controller uses, what the power buttons are, how the fans and sleep states work.
6. So the operating system reads devices from the buses and reads non-discoverable parts from ACPI tables, and combines both into one tree.

■ 13.12.5 TECHNICAL – the engineer's version

1. ISA Plug and Play was specified by Intel and Microsoft in 1993, using an isolation protocol at ports 0x279 and 0xA79 to enumerate cards that could not otherwise be discovered. It was fragile enough to earn a bad reputation in the Windows 95 era.
2. PCI configuration space is 256 bytes per function, extended to 4096 bytes in PCIe. Access is through ports 0xCF8 and 0xCFC in the legacy mechanism, or through the memory-mapped ECAM region described by the ACPI MCFG table.
3. ACPI 1.0 was published in December 1996 by Intel, Microsoft and Toshiba. It is now maintained by the UEFI Forum, which took it over in 2013. Current revisions are 6.x.
4. ACPI tables to know: DSDT and SSDT contain the AML bytecode describing devices, MADT describes interrupt controllers, MCFG describes PCIe configuration space, and FADT holds fixed hardware descriptions.
5. Windows Device Manager error codes you will actually meet: Code 10, the device cannot start, usually a driver or firmware fault; Code 12, not enough free resources, the modern trace of a genuine resource conflict; Code 28, no drivers installed; Code 43, the driver reported a failure.
6. Code 12 today is usually not two devices fighting over one IRQ. It is normally exhausted PCIe lanes or, on older 32-bit systems, exhausted address space below 4 GiB.

7. Inspect resource assignment on Linux with `cat /proc/iomem`, `cat /proc/ioports` and `lspci -vv`. On Windows use `msinfo32` and look under Hardware Resources for Conflicts/Sharing.
8. The honest version: interrupt sharing is still normal and correct on legacy PCI lines. Several devices on one line is not a conflict. A conflict is when the same exclusive resource is claimed twice.

■ 13.12.6 WORDS – remember these

1. Jumper — a manual setting cap — a shorting block across two header pins selecting a hardware option.
2. Resource — something a device must be given — an IO range, memory window, interrupt or DMA channel.
3. Plug and play — automatic setup — enumeration and dynamic resource allocation without user configuration.
4. Configuration space — the device's self-description — the 256-byte or 4096-byte PCI header with IDs and BARs.
5. ACPI — the description of the rest of the board — tables and AML bytecode describing power, and non-discoverable devices.

13.13 System-on-chip: when the whole board fits on one die

■ 13.13.1 PLAIN – in simple words

1. A phone has no slots, no card connectors and no chipset you can point at.
2. Nearly everything discussed in this chapter has been folded onto a single piece of silicon called a system-on-chip, or SoC.
3. The processor, graphics, memory controller, camera processor, radio controller, video decoder and security block all sit on one die.
4. The memory is usually stacked directly on top of the same package.
5. Nothing is removable, so nothing needs a slot, a connector or a plug and play discovery process.
6. The parts still talk to each other over buses, but those buses are inside the chip, made of metal layers, not board traces.
7. Because nothing can be discovered by asking, the software must be told in advance what exists and where. That description is called a device tree.

■ 13.13.2 PLAIN – a picture in your head

1. A desktop is a town with separate buildings and roads between them.
2. A system-on-chip is a single very large building, where every department is a room on the same floor.
3. Walking between rooms is far quicker than driving between buildings, and uses far less energy.
4. But you cannot add a new department. The building is finished.
5. And there is no directory at the entrance, because the plan was fixed when it was built. Everyone must be handed a map instead.

Where this comparison breaks: parts of the chip can still be discovered. A PCIe controller inside an SoC enumerates its own devices exactly as a desktop does. It is the fixed on-chip blocks that need the map.

■ 13.13.3 PLAIN – a worked example

1. A device tree entry describing a serial port on an ARM board looks like this.

```
uart0: serial@fe215040 {  
    compatible = "brcm,bcm2835-aux-uart";  
    reg = <0xfe215040 0x40>;  
    interrupts = <0 93 4>;  
    clocks = <&aux_clk>;  
    status = "okay";  
};
```

2. Read it line by line. `compatible` is the matching key, the equivalent of a vendor and product ID.
3. `reg` says the registers start at address 0xFE215040 and occupy 0x40, that is 64, bytes.
4. `interrupts` says which interrupt line it uses.
5. `clocks` says which clock must be switched on before it works.
6. The kernel reads this, finds the driver whose compatible string matches, and calls its probe function with the address and interrupt filled in.
7. That is exactly the same probe mechanism as PCIe, with the tree taking the place of bus enumeration.

■ 13.13.4 PLAIN – what is really happening inside

1. Inside the chip, blocks connect to a shared on-chip fabric: a set of wide internal buses with an arbiter deciding who gets access.
2. Fast blocks such as the processor and graphics sit on a wide, high-speed part of the fabric. Slow blocks such as a serial port sit on a narrow, cheap part behind a bridge.
3. Wide parallel buses are fine here, because the distance is millimetres and fixed at design time, so skew is controlled by the layout tools.
4. The device tree is compiled into a binary blob and handed to the kernel by the bootloader at start-up.
5. So the boot chain is: chip's internal ROM, then a small first-stage loader, then a full bootloader, then the kernel plus the device tree.
6. On x86 machines the same job is done by ACPI tables from the firmware. Arm servers use ACPI too; Arm phones and embedded boards use device trees.

■ 13.13.5 TECHNICAL – the engineer's version

1. On-chip interconnect standards come mostly from Arm's AMBA family: AXI for high-performance memory traffic, AHB for medium speed, and APB for slow peripheral registers. Larger chips use a network-on-chip with routers rather than a single bus.
2. Device tree came from Open Firmware, standardized as IEEE 1275 in 1994 and used by Sun and Apple PowerPC machines. Linux adopted it for PowerPC and then for Arm, source files ending `.dts`, compiled by `dtc` into `.dtb`.
3. Arm Linux switched to device trees around Linux 3.x, from 2011 onward, replacing thousands of lines of hardcoded board files.
4. Why phone drivers age badly. An SoC vendor ships a board support package: a kernel fork with hundreds of out-of-tree drivers. Because Linux has no stable in-kernel driver ABI, moving to a newer kernel means porting all of them. Vendors support a chip for a few years and then stop.
5. Google's Project Treble, introduced with Android 8.0 in 2017, split the vendor implementation from the Android framework behind a stable interface called the HAL, so the framework could be updated without the vendor's code changing.

6. The Generic Kernel Image work, from Android 11 onward, went further: one common kernel binary with vendor code confined to loadable modules across a defined module interface. Google Pixel and Samsung Galaxy devices now advertise seven years of updates, which was not possible under the old model.

7. Inspect on a running Linux SoC:

```
ls /proc/device-tree/           # the live tree
cat /proc/device-tree/model     # the board name
dtc -I fs -O dts /proc/device-tree | less
```

8. The honest version: “no slots” is not quite true. Many SoCs expose PCIe and USB, and a laptop built on an SoC still has USB-C ports that enumerate normally. What disappeared is the socketed, replaceable, discoverable internal hardware, not discoverable buses in general.

■ 13.13.6 WORDS – remember these

1. SoC — the whole computer on one chip — a die integrating CPU, GPU, memory controller and peripherals.
2. Fabric — the buses inside the chip — an on-chip interconnect such as AMBA AXI or a network-on-chip.
3. Device tree — the map of fixed hardware — a data structure describing non-discoverable devices, compiled to a DTB.
4. Compatible string — the matching key — the device tree property a driver binds to, in place of a vendor and product ID.
5. Board support package — the vendor’s kernel fork — a chip-specific kernel plus out-of-tree drivers.

13.98 Common wrong ideas

1. Wrong: a driver is a program you install and run. Right: it is code the operating system loads and calls; you never start it yourself, and it has no window.
2. Wrong: more wires means faster, so parallel beats serial. Right: above a few hundred megahertz, skew and crosstalk make wide buses unusable, which is why every fast modern link is narrow and serial.
3. Wrong: a USB-C port supports fast data, video and 240 W charging. Right: USB-C is only a connector shape. A USB-C port may be limited to USB 2.0 at 480 Mbit/s and 7.5 W. Only the specification sheet tells you.
4. Wrong: a card in an x16 slot always gets sixteen lanes. Right: the connector size and the wired electrical width are separate. A full-length slot may be wired x4, or drop to x8 when another slot is used.
5. Wrong: firmware and drivers are the same thing. Right: firmware runs on the device’s own processor and lives in the device’s flash. A driver runs on your processor inside your operating system.
6. Wrong: Secure Boot stops malware. Right: it verifies signatures on boot-time executables against firmware key stores. It does nothing about anything that runs after the kernel starts.
7. Wrong: interrupts are always better than polling. Right: at very high event rates, interrupt overhead dominates. Fast network and storage paths deliberately poll, using NAPI, DPDK or SPDK.
8. Wrong: two devices sharing an interrupt is a conflict. Right: shared level-triggered lines are normal and correct. A conflict is two claims on the same exclusive resource, and message-signalled interrupts removed most of that possibility.

9. Wrong: USB 3.2 Gen 2x2 is newer and therefore faster than USB4. Right: Gen 2x2 is 20 Gbit/s. USB4 reaches 40 Gbit/s, and USB4 Version 2.0 reaches 80 Gbit/s. The version numbers do not sort by speed.
10. Wrong: signed drivers are safe drivers. Right: a signature proves origin and integrity, not correctness. Signed drivers with serious bugs and exploitable flaws are shipped regularly.

13.99 Chapter summary in 20 lines

1. A motherboard is a fixed set of sockets, slots, chips and printed wires, each with one job.
2. Fast parts sit close to the processor because high-speed wires must be short and length-matched.
3. A bus carries address, data and control, and a shared bus lets only one talker speak at a time.
4. Parallel buses died above a few hundred megahertz because of skew, crosstalk and electrical loading.
5. Serial point-to-point links won by moving the hard problem into silicon: scrambling, equalization and clock recovery.
6. Bandwidth is transfers per second times width times coding efficiency; PCIe 3.0 x1 is 8 GT/s x 128/130 / 8, about 985 MB/s each way.
7. The bus story runs ISA in 1981, VESA Local Bus in 1992, PCI in 1992, AGP in 1997, then PCI Express from 2003 to now.
8. Connector size and electrical width are separate; bifurcation splits one wide link into several narrower ones.
9. A processor reaches a device through memory-mapped registers or, on x86 only, a separate port space.
10. Polling burns cycles, interrupts free them, and direct memory access removes the processor from bulk transfers entirely.
11. The IOMMU translates and checks every device memory access, which is what makes Thunderbolt and virtual machine passthrough safe.
12. Interrupts run from line or message, through a controller, to a vector, to a short handler, with the slow work deferred to a bottom half.
13. USB has one host, a tiered star of hubs, endpoints, pipes and four transfer types, and enumerates in a fixed nine-step sequence.
14. USB naming has been rewritten twice; as of 2026 the guidance is speed labels: USB 5Gbps through USB 80Gbps.
15. USB-C is a 24-pin connector from 2014, not a protocol and not a speed.
16. Thunderbolt tunnels PCI Express and DisplayPort inside one link, reaching 80 Gbit/s in Thunderbolt 5 from 2023.
17. A driver translates a general operating system request into one specific chip's registers, sitting under a class driver and above the hardware.
18. Windows uses WDM and WDF with KMDF and UMDF, Linux uses kernel modules with sysfs and udev, macOS is replacing kexts with DriverKit.
19. Firmware runs on the device's own processor from its own flash, before and after your operating system, and a failed update can brick the device.
20. Platform firmware, BIOS then UEFI, runs POST, enumerates buses, reads its boot entries and loads a bootloader; Chapter 18 continues from there.

The Long Road — A History of Computing Machines

14.0 What this chapter gives you

1. You will be able to tell the story of computing from tally sticks to the machines of 2026, in order, with correct years and names.
2. You will be able to explain what a punched card did in a loom in 1804, and why that counts as the first machine following stored instructions.
3. You will be able to say what Charles Babbage designed, what Ada Lovelace wrote in 1843, and where the honest scholarly argument about her sits.
4. You will be able to explain a Turing machine, the halting problem and the phrase “Turing complete” to somebody who knows no mathematics.
5. You will be able to name the wartime machines of 1941 to 1945 and say exactly what each one could and could not do.
6. You will be able to explain why the stored-program idea of 1945 changed everything, and who deserves credit for it.
7. You will be able to trace the chain: transistor, integrated circuit, microprocessor, personal computer, graphical screen, network, phone.
8. You will be able to say why the industry stopped raising clock speed around 2005 and started adding cores instead.
9. You will be able to correct the five most common history mistakes people repeat about computers.
10. You will have one master timeline you can read in five minutes.

14.1 Counting before machines

■ 14.1.1 PLAIN - in simple words

1. For most of human history, counting aids were sticks, stones and beads.
2. A **tally stick** is a stick with notches cut into it. One notch, one thing.
3. Some tally sticks were split lengthwise so two people each kept half. The matching notches proved the debt. England used such sticks for tax records for centuries, right up to 1826.
4. An **abacus** is a frame of beads on rods. Each rod is a decimal place.
5. The beads do not calculate. The person calculates. The beads hold the partial answer so the person does not have to remember it.
6. That is the first big idea in this whole chapter: a machine that remembers for you is already useful, even if it does no thinking.

7. In 1901 sponge divers found a corroded lump of bronze in a shipwreck off the Greek island of Antikythera.
8. Inside were at least 30 meshing gears. It is called the **Antikythera mechanism**, and it modelled the sky.
9. Turn a handle and pointers showed the positions of the Sun and Moon, the phase of the Moon, and when eclipses were due.
10. Nothing of that mechanical complexity is known again for over a thousand years. It was a dead end, not a starting point.

■ 14.1.2 PLAIN – a picture in your head

1. Think of a kitchen with no notepad.
2. You are adding a long shopping list in your head. By item nine you have lost item three.
3. Now somebody hands you a row of ten cups and a bag of stones.
4. You drop stones into cups to hold each part of the sum. Your head is free.
5. The cups do not add. You add. The cups only refuse to forget.
6. Napier's bones, the slide rule and the abacus are all cups and stones. They store a partial result, or they turn a hard job into an easy one.
7. The Antikythera mechanism is different. It is a clockwork sky. The gear ratios *are* the astronomy, baked into metal once and for all.

Where this comparison breaks: cups and stones can hold any number you like, but a gear train can only ever do the one calculation its teeth were cut for. To change the question you must cut new gears. That difference, fixed function versus changeable instructions, is the whole story of the next 2,000 years.

■ 14.1.3 PLAIN – a worked example

1. John Napier published **Napier's bones** in 1617, in a book called *Rabdologiae*, printed in Edinburgh. He died that same year.
2. Each bone is a rod carrying one column of the multiplication table.
3. To multiply 4,875 by 7, lay out the bones for 4, 8, 7 and 5, and read row 7.
4. Row 7 gives, in each bone, the digits of 7 times that digit, split by a diagonal line into tens and units.

Bones for	4	8	7	5
Row 7:	2/8	5/6	4/9	3/5

Add along the diagonals, right to left:

units:	5			
next:	3 + 9 = 12	-> write 2, carry 1		
next:	1 + 4 + 6 = 11	-> write 1, carry 1		
next:	1 + 5 + 8 = 14	-> write 4, carry 1		
next:	1 + 2 = 3			

Result: 34125

5. Check: 4,875 times 7 is 34,125. The bones turned four multiplications into four small additions.
6. That is the trick every pre-electronic aid uses. Convert a hard operation into an easier one you can already do.

■ 14.1.4 PLAIN – what is really happening inside

1. Napier's other invention, logarithms, published in 1614, does the same trick at a deeper level.
2. A logarithm turns multiplication into addition. Multiply two numbers by adding their logarithms and looking up the answer.
3. Edmund Gunter engraved a logarithmic scale on a ruler around 1620. William Oughtred put two such scales side by side around 1622 and slid one along the other. That is the **slide rule**.
4. Sliding one scale by a distance *is* adding the logarithms. The mechanism does the addition by geometry.
5. In 1623 Wilhelm Schickard, a professor at Tübingen, described a “calculating clock” in letters to the astronomer Johannes Kepler.
6. It used Napier's bones mounted on cylinders for multiplication and a geared adder underneath. The machine being built for Kepler burned in a fire before delivery. Schickard died of plague in 1635.
7. In 1642 the nineteen-year-old Blaise Pascal built the **Pascaline** to help his father, a tax official, add long columns of money.
8. It added and subtracted with toothed wheels, and it carried automatically: when a wheel passed nine, a weighted lever tipped and nudged the next wheel.
9. Around fifty were made. About nine survive.
10. Gottfried Wilhelm Leibniz went further. He showed a wooden model of his **stepped reckoner** to the Royal Society in London on 1 February 1673.
11. It was the first design that could do all four operations: add, subtract, multiply and divide.
12. In 1703 Leibniz published a short paper, *Explication de l'Arithmétique Binaire*, setting out arithmetic using only 0 and 1.
13. He did not build a binary machine. But every machine in the rest of this book runs on the notation he wrote down in 1703.

■ 14.1.5 TECHNICAL – the engineer's version

1. The Antikythera mechanism carries at least 30 surviving bronze gears; the largest has 223 teeth, encoding the 223-month Saros eclipse cycle.
2. Dating is contested. Published estimates cluster around 150 to 100 BCE, with arguments for 205 BCE and for about 87 BCE. The shipwreck itself dates to roughly 70 to 60 BCE. Experts disagree; treat “around 100 BCE” as a reasonable midpoint, not a settled fact.
3. Its lunar train uses a pin-and-slot on two gears of 53 teeth to produce a varying angular rate, modelling the Moon's elliptical motion. That is a mechanical implementation of a first-order anomaly correction.
4. The Pascaline's carry mechanism, the *sautoir*, is gravity-driven. Its drawback is that a long carry chain, such as 999,999 plus 1, must lift several weights at once, so torque rises with carry length.
5. Leibniz's key part is the **stepped drum** (Staffelwalze): a cylinder with nine teeth of increasing length. Shifting a gear along the drum selects how many teeth engage per turn, which multiplies by a chosen digit.
6. The stepped drum stayed in production hardware until the twentieth century, in the Thomas Arithmometer from 1820 and later in the Curta of 1948.

Device	Year	Operations it did
Napier's bones	1617	Multiply, divide aid
Slide rule	c. 1622	Multiply, divide, roots
Schickard clock	1623	Add, subtract, aid mult

Device	Year	Operations it did
Pascaline	1642	Add, subtract
Stepped reckoner	1673	All four operations

- Precision limits, not ideas, blocked all of these. Leibniz's machine had a carry fault that was only fully understood when it was studied in the twentieth century.
- Standard versus convention: none of this is standardized. These are one-off artefacts. The first true standard in computing history is the punched card format, and that comes in section 14.4.

■ 14.1.6 WORDS – remember these

- Tally stick — a notched stick counting things — a unary record medium.
- Abacus — beads on rods — a positional manual accumulator.
- Antikythera mechanism — an ancient geared sky model — a bronze analogue astronomical calculator, roughly 100 BCE.
- Napier's bones — rods holding times tables — lattice multiplication aid, 1617.
- Logarithm — a number that turns times into plus — the exponent to which a fixed base must be raised.
- Slide rule — two sliding log scales — an analogue multiplier, about 1622.
- Pascaline — Pascal's adding box of 1642 — geared decimal adder with gravity carry.
- Stepped reckoner — Leibniz's four-function machine, 1673 — stepped-drum mechanical calculator.
- Binary — counting with only 0 and 1 — base-2 positional notation, set out by Leibniz in 1703.

14.2 Machines that follow instructions

■ 14.2.1 PLAIN – in simple words

- A calculator does one job. To change the job you change the machine.
- The next idea is bigger: leave the machine alone, and change a stack of cards instead.
- In 1804 Joseph Marie Jacquard, in Lyon in France, patented an attachment that sat on top of a weaving loom.
- Weaving a picture into cloth means lifting a different set of threads for every single row of the picture.
- Before Jacquard, a boy sat inside the loom pulling cords by hand, one row at a time, all day, for weeks.
- Jacquard replaced the boy with a chain of stiff cards, each punched with holes.
- Rods press against a card. Where there is a hole, a rod goes through and its thread lifts. Where there is card, the rod is blocked and the thread stays.
- One card equals one row of the picture. The chain of cards equals the whole picture.
- To weave a different picture you do not rebuild the loom. You hang a different chain of cards.
- That separation, machine on one side and instructions on the other, is the single most important idea in this book.

■ 14.2.2 PLAIN – a picture in your head

- Think of a music box with a spiked cylinder.
- The spikes pluck the metal teeth. The cylinder is the tune.
- A cheap music box has the cylinder fixed. It plays one tune for ever.

4. A better music box lets you pull the cylinder out and push a new one in. Same box, new tune.
5. Jacquard's loom is that better music box, except the tune is a picture and the cylinder is a loop of punched cards.
6. A player piano works exactly this way too, with a punched paper roll.

Where this comparison breaks: a music box or piano roll only ever plays forwards. It cannot skip back, repeat a bar four times, or choose a different bar depending on what happened. The Jacquard chain has the same limit. Real programs need to jump, loop and decide. That missing power is exactly what Babbage tried to add in 1837, and what nobody achieved in working hardware until the 1940s.

■ 14.2.3 PLAIN - a worked example

1. Take a tiny loom with 8 threads, weaving a picture 5 rows tall.
2. Punch 1 means "lift this thread". No punch, shown as 0, means "leave it".

Card 1:	0	0	1	1	1	1	0	0
Card 2:	0	1	1	0	0	1	1	0
Card 3:	1	1	0	0	0	0	1	1
Card 4:	0	1	1	0	0	1	1	0
Card 5:	0	0	1	1	1	1	0	0

3. Read the 1s down the page and a diamond shape appears in the cloth.
4. Five cards, eight holes each: 40 bits of information decide a picture.
5. Real Jacquard cards of the period commonly carried several hundred hole positions per card, and a single elaborate portrait could need tens of thousands of cards.
6. The famous woven portrait of Jacquard himself, made in the 1830s, used around 24,000 cards for one image.
7. Charles Babbage owned a copy of that portrait and showed it to visitors. That is a direct, documented link from weaving to computing.

■ 14.2.4 PLAIN - what is really happening inside

1. Jacquard did not invent this from nothing. He combined three earlier French ideas.
2. Basile Bouchon used a punched paper tape for loom control in 1725.
3. Jean-Baptiste Falcon replaced the fragile tape with a chain of stiff cards in 1728.
4. Jacques de Vaucanson built an automatic loom around 1740 using a perforated cylinder.
5. Jacquard's 1804 machine put the strong parts together and made it reliable and cheap enough for real workshops.
6. Mechanically: a rectangular card presses against a bank of spring-loaded needles. Each needle controls a hook. Each hook lifts one warp thread.
7. Hole present: needle passes through, hook stays engaged with the lifting knife, thread rises.
8. No hole: needle is pushed back, hook is displaced sideways, the knife misses it, thread stays down.
9. The card is therefore a row of yes/no decisions, read in parallel, once per row of cloth. It is a memory device made of cardboard.
10. Weavers understood the threat immediately. There were riots in Lyon, and looms were destroyed. This is where the word Luddite belongs, though the English Luddites of 1811 to 1816 were mostly attacking other machines.

■ 14.2.5 TECHNICAL – the engineer’s version

1. The Jacquard head is a parallel-read, sequential-advance control store. Card width sets the number of parallel control lines.
2. Typical nineteenth-century heads came in sizes such as 400, 600 and 900 hooks. A 900-hook head is a 900-bit-wide control word.
3. Advance is mechanical and unconditional: a four-sided prism rotates one quarter turn per pick, presenting the next card. There is no branch, no conditional and no addressing.
4. In formal terms the Jacquard head is a finite-state sequencer with an external, replaceable program tape, and no conditional transfer. It is not Turing complete and was never meant to be.
5. The honest version: calling a Jacquard card chain “the first program” is a useful teaching device, not a precise claim. It is the first widely used **stored, replaceable, machine-readable instruction sequence**. It is not a program in the modern sense because it cannot branch or loop.
6. The lineage is nonetheless direct and documented. Babbage planned card input for the Analytical Engine from 1837, Herman Hollerith used cards for the 1890 census, and IBM’s punched card business ran on that idea into the 1970s.

Machine	Year	Program medium
Bouchon loom	1725	Punched paper tape
Falcon loom	1728	Chain of cards
Vaucanson loom	c. 1740	Perforated cylinder
Jacquard head	1804	Chain of stiff cards
Analytical Engine	1837	Punched cards, planned

■ 14.2.6 WORDS – remember these

1. Punched card — stiff card with holes meaning yes or no — a parallel-read binary storage medium.
2. Warp — the threads held tight lengthwise on a loom — the addressable lines the control head selects.
3. Jacquard head — the card-reading box on top of a loom, 1804 — a wide parallel control-store sequencer.
4. Program — a list of instructions a machine follows — an encoded sequence of operations held separately from the mechanism.
5. Branch — choosing what to do next — a conditional transfer of control, which the Jacquard head lacks.

14.3 Charles Babbage and Ada Lovelace

■ 14.3.1 PLAIN – in simple words

1. In the early 1800s, sailors, bankers and astronomers all used printed books of numbers called tables.
2. The tables were worked out by hand by people whose job title was **computer**. That word meant a person until the 1940s.
3. People make mistakes. Printers make more. A wrong number in a navigation table could put a ship on rocks.
4. Charles Babbage, an English mathematician, wanted to remove the humans from both steps: calculate by machine, and print by machine, so no typesetter could introduce an error.
5. On 14 June 1822 he described his **Difference Engine** to the Royal Astronomical Society.

6. It could not multiply. It only added. That was enough, because of a mathematical trick explained in the worked example below.
7. The British government funded it from 1823. By 1842 it had spent over 17,000 pounds, a huge sum, and the machine was not finished.
8. Meanwhile Babbage had a much bigger idea. From 1837 he designed the **Analytical Engine**, a general-purpose machine.
9. It had a **mill** to do the arithmetic, a **store** to hold numbers, a card **reader** for instructions, and a **printer** for output.
10. Those four parts map exactly onto a modern computer: processor, memory, input, output.
11. Neither machine was completed in his lifetime. Babbage died in 1871.

■ 14.3.2 PLAIN - a picture in your head

1. Think of a very large mechanical cash register, the size of a small room, made of brass and iron, turned by a handle or a steam engine.
2. Inside are columns of numbered wheels. Each column holds one number.
3. Turning the handle makes the columns add themselves together in a fixed pattern, and a new answer appears at the front.
4. Keep turning and a whole table of numbers pours out, and a printing plate is stamped at the same time.
5. That is the Difference Engine: one fixed pattern, endlessly repeated.
6. Now imagine a second machine where a stack of cards tells the columns what to do this time, and where the machine can look at a result and choose a different card next. That is the Analytical Engine.

Where this comparison breaks: a cash register has one accumulator. Babbage's store was designed for 1,000 numbers of 40 decimal digits each, which is about 16.6 kilobytes. And no cash register can change its own next action based on a result. That ability to decide is what makes the Analytical Engine a computer and the Difference Engine a calculator.

■ 14.3.3 PLAIN - a worked example

1. Here is the trick that lets a machine that can only add produce a table of squares.
2. Write the squares: 1, 4, 9, 16, 25, 36.
3. Subtract each from the next. These are the first differences: 3, 5, 7, 9, 11.
4. Subtract those from each other. The second differences: 2, 2, 2, 2. Constant.
5. So you can run the process backwards using only addition.

Step	2nd diff	1st diff	Value
start	2	3	1
1	2	5	4
2	2	7	9
3	2	9	16
4	2	11	25

6. Each row: add the second difference to the first difference, then add the first difference to the value. Two additions per table entry.
7. Any polynomial of degree n has constant n th differences, so this works for a whole family of useful functions.

8. Logarithms and sines are not polynomials, but they can be approximated by polynomials over short ranges, which is how real tables were produced.
9. Difference Engine No. 1 was designed for 20-digit numbers and sixth-order differences. Difference Engine No. 2, designed between 1847 and 1849, handled 31-digit numbers and seventh-order differences.

■ 14.3.4 PLAIN – what is really happening inside

1. Now Ada Lovelace. She was born in 1815, the daughter of the poet Lord Byron, and was taught mathematics from childhood.
2. She met Babbage in 1833 and followed the Analytical Engine work closely.
3. In 1840 Babbage lectured in Turin. An Italian engineer, Luigi Federico Menabrea, wrote the lecture up in French and published it in 1842.
4. Lovelace translated Menabrea's article into English over about nine months in 1842 and 1843, and added seven notes of her own, labelled A to G.
5. Her notes are roughly three times longer than the article she translated.
6. **Note G** contains a step-by-step table showing how the Analytical Engine would compute the Bernoulli numbers, a sequence used in advanced arithmetic.
7. That table has variables, operations, and a loop. It is widely called the first published computer program.
8. Note A contains something arguably more important. Lovelace saw that the numbers in the machine need not stand for quantities at all.
9. She wrote that if the relations of pitched sounds could be expressed as numbers, the engine might compose music. That is the idea of general computing, written down in 1843.
10. She also wrote that the machine has no pretensions to originate anything, and can only do what we know how to order it to perform. That line is still quoted in arguments about artificial intelligence today.

■ 14.3.5 TECHNICAL – the engineer's version

1. Analytical Engine specification, as designed from 1837: store of 1,000 numbers of 40 decimal digits; decimal, not binary; mill performing the four operations; separate operation cards and variable cards; conditional branching via a mechanism Babbage called the "combinatorial" or barrel control; output to printer, curve plotter and bell.
2. Estimated speed, from Menabrea's account of Babbage's claims: an addition in about one second, and the product of two 20-digit numbers in about three minutes.
3. Babbage anticipated microprogramming: the barrel held sequences of micro-operations that a single instruction card triggered. Maurice Wilkes, who invented microprogramming in 1951, noted the resemblance.
4. The scholarly argument about Lovelace, stated fairly, both sides:
 1. Against: Allan Bromley wrote in 1990 that all but one of the programs in her notes had been prepared by Babbage three to seven years earlier. Bruce Collier, also in 1990, judged that she publicized rather than advanced the design. Eugene Eric Kim and Betty Alexandra Toole argued in 1999 that calling her the first programmer is incorrect.
 2. For: Doron Swade, who led the Science Museum's Babbage work, holds that she alone saw that numbers could represent things other than quantity, which is the conceptual leap to general-purpose computing. Stephen Wolfram, writing in 2016, judged her Bernoulli computation as sophisticated as anything Babbage produced.
5. The defensible statement: Babbage wrote earlier program-like traces that were not published;

Lovelace produced the first published, fully worked algorithm intended for a machine, and the clearest statement of what a general-purpose computer would mean. Both facts are true at once.

6. Neither engine was finished, for three reasons: cost, Babbage's 1833 quarrel with his engineer Joseph Clement over tool ownership, and Babbage's habit of redesigning before finishing.
7. The vindication came late. The Science Museum in London built Difference Engine No. 2 to the original 1847 to 1849 drawings, completing the calculating section in 1991 for Babbage's 200th birthday, and the printer in 2002. It has about 8,000 parts and weighs around 5 tonnes. It works.

Item	Babbage's design	Modern equivalent
Mill	Arithmetic unit	ALU
Store	1,000 x 40 digits	RAM, about 16.6 kB
Operation cards	What to do	Instruction stream
Variable cards	Which store cell	Operand addresses

■ 14.3.6 WORDS – remember these

1. Difference Engine — Babbage's table-printing adder, 1822 — a fixed-function polynomial evaluator using finite differences.
2. Analytical Engine — Babbage's general machine, 1837 — a programmable decimal computer with separate mill, store and card input.
3. Mill — the part that does the sums — the arithmetic and logic unit.
4. Store — the part that remembers numbers — main memory.
5. Method of differences — building a table by adding only — evaluating a degree-n polynomial from constant nth differences.
6. Note G — Lovelace's Bernoulli number table, 1843 — the first published algorithm written for a machine.

14.4 Hollerith, the 1890 census, and the birth of IBM

■ 14.4.1 PLAIN – in simple words

1. The United States counts its whole population every ten years. The constitution requires it.
2. The 1880 count took about eight years to process by hand.
3. The country was growing so fast that the 1890 count risked not being finished before the 1900 count began. That is a real crisis, not a joke.
4. Herman Hollerith, a young engineer who had worked on the 1880 census, had an idea borrowed from railway tickets and from Jacquard's loom.
5. Punch each person's answers as holes in a card. One card, one person.
6. Then build a machine that reads the holes electrically and counts them.
7. He was granted his key United States patent, number 395,782, on 8 January 1889.
8. His machines were used for the 1890 census. The main population count was announced within about six weeks, and the full processing took roughly six years instead of eight, on a much larger and more detailed job.
9. Hollerith founded the Tabulating Machine Company in 1896.
10. In 1911 it merged with three other firms to form the Computing-Tabulating-Recording Company, or CTR.

11. In 1924, under Thomas J. Watson, CTR was renamed International Business Machines. That is IBM.

■ 14.4.2 PLAIN – a picture in your head

1. Imagine a hotel with 200 rooms and a wall of pigeonholes behind reception.
2. Every guest hands in a card with holes punched in it: nationality, age group, room type.
3. The receptionist drops each card into a slot. A lid comes down. Little metal pins press against the card.
4. Where a hole is, a pin dips through and touches mercury in a cup underneath. That completes an electric circuit and a dial clicks up by one.
5. At the same moment, a lid pops open on one pigeonhole, telling the clerk where to file that card.
6. A hundred dials, a hundred questions, one card at a time, all day.

Where this comparison breaks: the real machine was not automatic in the way we would expect. An operator fed every card by hand. The sorting box only opened the correct lid; the human still moved the card. Fully automatic card feeding came later, with Hollerith's own 1900 machines and their successors.

■ 14.4.3 PLAIN – a worked example

1. Say we want the count of people in a city who are both over 60 and born abroad.
2. Each card has a position for “age band 60 plus” and a position for “born abroad”.
3. The tabulator is wired so a counter advances only when both pins make contact at once.
4. That wiring is a physical AND gate, built from relays, decades before anyone used the word.

```
Card position 34 punched -> pin 34 dips -> circuit A closed  
Card position 61 punched -> pin 61 dips -> circuit B closed
```

```
Counter 7 advances only if A AND B are closed.
```

5. Change the question by rewiring the plugboard, not by rebuilding the machine.
6. Speed in practice: a trained operator could process roughly 80 cards a minute on the 1890 equipment, so about 1,000 cards an hour with breaks.
7. Roughly 62.9 million people were counted in the 1890 census. The scale is what made the machinery pay for itself.

■ 14.4.4 PLAIN – what is really happening inside

1. The card is the data. The machine is a fixed reader. The plugboard is the program.
2. This is the same split as the Jacquard loom, but now the cards hold facts rather than instructions, and a separate wiring panel holds the instructions.
3. Hollerith's business insight mattered as much as his engineering. He rented the machines rather than selling them, and he sold the blank cards.
4. That model, cheap machine, endless consumable, is the razor-and-blades business. It funded IBM for sixty years.
5. IBM standardized an 80-column rectangular-hole card in 1928. The 80-column card is a genuine industry **standard**, and it is why terminal screens were 80 characters wide for decades, and why many text tools still default to 80.
6. There is a dark chapter that honesty requires naming. IBM's German subsidiary Dehomag supplied punched card equipment used by the Nazi state in the 1930s and 1940s, including for censuses and record-keeping. Historian Edwin Black documented this in his 2001 book *IBM and the*

Holocaust. The degree of knowledge and control at IBM headquarters is disputed by historians; the supply of equipment is not.

■ 14.4.5 TECHNICAL – the engineer’s version

1. The 1890 card was about 3.25 by 7.375 inches, chosen to match the United States dollar bill of the day so existing cash drawers could store it.
2. The 1890 census card carried up to 288 punch positions in a 12 by 24 grid.
3. Reading was electromechanical: a spring-loaded pin array descends; a pin passing through a hole contacts a mercury cup, closing a circuit that steps a clockwork-style counter dial reading 0 to 9,999.
4. The sorter box had 26 compartments with electrically released lids.
5. The IBM card of 1928 has 80 columns by 12 rows, rectangular holes, and encodes one character per column using Hollerith code: a zone punch in rows 12, 11 or 0 combined with a digit punch in rows 1 to 9.
6. The card measures 7.375 by 3.25 inches by 0.007 inches thick. That thickness is why a box of 2,000 cards is about 14 inches tall.

Year	Machine or event	Significance
1889	US patent 395,782	Electric card tabulating
1890	US census run	First large data processing
1896	Tabulating Machine Co	Hollerith’s firm
1911	CTR formed	Four-company merger
1924	Renamed IBM	Name that dominated 60 yrs
1928	80-column card	Format standard to 1970s

7. Card equipment was not a dead end. Unit record machines, sorters, collators, reproducers and accounting machines, ran commercial data processing until magnetic tape displaced them through the 1960s.
8. The last widespread public use of punched cards was in voting machines. The 2000 United States presidential election recount in Florida turned on partly detached card chads, which is why the word “chad” briefly became famous.

■ 14.4.6 WORDS – remember these

1. Tabulator — a machine that counts punched cards — an electromechanical counting unit driven by card-hole contacts.
2. Plugboard — a wiring panel that sets the question — a removable control panel defining data paths, the program of a unit record machine.
3. Unit record — one card, one thing — the data model where each card is a complete record.
4. Hollerith code — the hole pattern for a character — zone plus digit punch encoding on an 80-column card.
5. CTR — the 1911 merger — Computing-Tabulating-Recording Company, renamed IBM in 1924.
6. Chad — the bit punched out of a card — the removed rectangle of card stock, famous from the 2000 Florida recount.

14.5 The theory arrives: Hilbert, Gödel, Church and Turing

■ 14.5.1 PLAIN – in simple words

1. Before anyone built a working computer, mathematicians worked out what a computer could never do. That is a strange and wonderful order of events.
2. Around 1920 the German mathematician David Hilbert proposed a plan for all of mathematics.
3. He wanted a fixed set of starting rules from which every true statement could be proved, with no contradictions, and with a mechanical method for deciding whether any given statement was provable.
4. That last part, the mechanical decision method, had a German name: the **Entscheidungsproblem**, the decision problem. Hilbert and Wilhelm Ackermann stated it clearly in a 1928 textbook.
5. In 1931 Kurt Gödel, a 25-year-old Austrian, destroyed the first two parts.
6. His **incompleteness theorems** showed that any consistent set of rules strong enough to describe ordinary arithmetic must contain true statements it cannot prove, and cannot prove its own consistency.
7. The third part, the decision problem, was still open. Two people answered it independently in 1936.
8. Alonzo Church at Princeton invented a symbol-rewriting system called the **lambda calculus** and used it to prove no such decision method exists.
9. Alan Turing in Cambridge invented an imaginary machine and proved the same thing, in a way that anybody can picture.
10. Turing's imaginary machine turned out to describe every computer that has ever been built.

■ 14.5.2 PLAIN – a picture in your head

1. Picture an endless paper tape, ruled into squares.
2. Each square is blank or holds one symbol from a small fixed alphabet.
3. A small box sits over one square. The box can read that square, rub it out and write a new symbol, and move one square left or right.
4. The box has a **state**, which you can think of as a mood. It has a small fixed number of moods, written on a card inside it.
5. The card is a table of rules of the form: if my mood is 3 and I see a 1, then write 0, move left, and change to mood 7.
6. That is the whole machine. Tape, head, moods, rule table.
7. There is nothing else. No screen, no memory chip, no operating system.
8. Turing's shocking claim is that anything a human being can compute by following a written procedure, this box can compute too.

Where this comparison breaks: the tape is infinite, and no real machine has infinite memory. Real computers are finite-state machines. In practice the tape just needs to be long enough for the problem, so the model is useful anyway. The second break: the model says nothing about speed. A Turing machine takes absurdly many steps for tasks a real processor does in one instruction. It answers “possible or impossible”, never “fast or slow”.

■ 14.5.3 PLAIN – a worked example

1. Here is a complete Turing machine that adds 1 to a binary number.
2. The number is written on the tape, most significant bit on the left. The head starts on the leftmost digit. Blank squares are shown as _.

State	Read	->	Write	Move	New state
right	0		0	R	right
right	1		1	R	right

right	—	—	L	carry
carry	0	1	L	done
carry	1	0	L	carry
carry	—	1	L	done
done	any	same	-	halt

3. Run it on the tape `_1011_`, which is 11 in decimal.
4. In state `right` the head walks over 1, 0, 1, 1 and hits the blank.
5. It steps left into state `carry` on the last 1. Rule: write 0, move left, still carrying.
6. Next square is 1. Write 0, move left, still carrying.
7. Next square is 0. Write 1, move left, done.
8. The tape now reads `_1100_`, which is 12. Correct.
9. Six rules and two moods add 1 to any binary number of any length. That is the power of the model in one small picture.

■ 14.5.4 PLAIN – what is really happening inside

1. Turing's paper is called *On Computable Numbers, with an Application to the Entscheidungsproblem*. He submitted it on 28 May 1936. It was read to the London Mathematical Society on 12 November 1936 and printed in their Proceedings across 1936 and 1937.
2. Church's paper *An Undecidable Problem of Elementary Number Theory* appeared in the American Journal of Mathematics in April 1936, a few months earlier.
3. Turing added something Church did not have: the **universal machine**.
4. A universal Turing machine is a Turing machine whose rule table is written to read *another machine's rule table* off the tape and imitate it.
5. That is the stored-program computer, described in 1936 as a piece of mathematics, nine years before anybody built one.
6. Now the famous impossibility. Ask: can we write a program that reads any program plus its input, and always answers correctly whether that program will eventually stop or run for ever?
7. Suppose such a checker exists. Build a troublemaker program that runs the checker on itself, and then does the opposite: if the checker says "stops", the troublemaker loops for ever; if the checker says "loops", it stops.
8. Now ask the checker about the troublemaker. Either answer makes the checker wrong. So no such checker can exist.
9. That is the **halting problem**, and it is not a limit of today's computers. It is a limit of computing itself, for ever.
10. The **Church-Turing thesis** says that lambda calculus, Turing machines, and every other reasonable definition of "effectively computable" describe exactly the same set of problems.
11. It is a thesis, not a theorem. It cannot be proved, because "what a human could compute with paper and pencil" is not a formal object. Ninety years of failed attempts to beat it are the evidence.

■ 14.5.5 TECHNICAL – the engineer's version

1. Formally a deterministic single-tape Turing machine is a 7-tuple: states Q , input alphabet, tape alphabet, transition function, start state, accept state, reject state.
2. Turing's own 1936 proof was about "circle-free" machines and the printing problem, not literally about halting. The name **halting problem** and the modern formulation are due to Martin Davis, in his 1958 book *Computability and Undecidability*.
3. The name **Church-Turing thesis** was coined by Stephen Kleene in his 1952 book *Introduction to Metamathematics*.

4. **Turing complete** means a system can simulate a universal Turing machine, given unbounded storage. **Turing equivalent** means two systems simulate each other.
5. Turing completeness needs surprisingly little: sequencing, a way to store and read unbounded data, and a conditional branch. Anything with those three is almost always complete.
6. Because the bar is so low, systems become Turing complete by accident. Real documented examples, with attributions:

System	Shown by	Year
Conway's Game of Life	Conway, then others	1970 on
C++ templates	Erwin Unruh, others	1994 on
Rule 110 automaton	Matthew Cook	2004
x86 MOV instruction	Stephen Dolan	2013
PowerPoint animations	Tom Wildenhain	2017
Magic: The Gathering	Churchill, Biderman, Herrick	2019

7. Read that table carefully. The PowerPoint result was presented at SIGBOVIK, a deliberately humorous conference, but the construction is real. The Magic result is a genuine peer-reviewed style construction using legal cards.
8. Practical consequences of undecidability, which working engineers meet:
 1. No compiler can decide in general whether your loop terminates, so termination checkers are conservative and refuse some correct programs.
 2. No antivirus can decide in general whether a program is malicious, which is why detection is heuristic and signature-based.
 3. Rice's theorem, proved by Henry Gordon Rice in 1951, generalizes this: every non-trivial property of a program's behaviour is undecidable.
9. The honest version: "undecidable" does not mean "hopeless in practice". It means no method works for every input. Tools such as static analysers, model checkers and the Coq and Lean proof assistants succeed on huge classes of real programs by giving up on the rest.

■ 14.5.6 WORDS - remember these

1. Entscheidungsproblem — Hilbert's question of 1928, is there a mechanical test for provability — the decision problem for first-order logic, answered no in 1936.
2. Incompleteness — some true things cannot be proved — Gödel's 1931 theorems about consistent recursively axiomatized systems containing arithmetic.
3. Turing machine — tape, head, moods, rules — an abstract automaton with unbounded tape and a finite transition function.
4. Universal machine — a machine that imitates any other from its description — the theoretical stored-program computer, Turing 1936.
5. Halting problem — you cannot always tell if a program will stop — the undecidability of the halting set, named by Davis in 1958.
6. Church-Turing thesis — all reasonable definitions of computing agree — an unprovable but unbeaten identification of effective calculability with Turing computability.
7. Turing complete — powerful enough to compute anything computable — able to simulate a universal Turing machine given unbounded memory.

14.6 War machines, 1939 to 1945

■ 14.6.1 PLAIN – in simple words

1. The Second World War pushed several countries to build calculating machines at a scale nobody would have funded in peacetime.
2. Germany encrypted its radio messages with the **Enigma**, a typewriter-sized machine with rotating wheels.
3. Press a key and an electrical path runs through the wheels and lights a different letter. The wheels then step, so the next letter is enciphered differently.
4. Polish mathematicians broke into Enigma first. Marian Rejewski worked out the internal wiring mathematically in December 1932.
5. In July 1939, weeks before the invasion, Poland handed everything it knew to British and French intelligence. That gift saved years.
6. Britain's codebreaking moved to **Bletchley Park**, a country house north of London, in August 1939.
7. Alan Turing and Gordon Welchman designed the **Bombe**, a machine that hunted for the day's Enigma settings.
8. A second, much harder German cipher, called Lorenz by the Germans and Tunny by the British, protected messages between high command and field marshals.
9. To attack Tunny, a Post Office engineer named Tommy Flowers built **Colossus**, the first large-scale electronic digital machine.
10. At the same time Konrad Zuse in Berlin, John Atanasoff in Iowa and Howard Aiken at Harvard were each building very different machines.

■ 14.6.2 PLAIN – a picture in your head

1. Think of a hotel with a combination lock on the front door, and the combination is changed every midnight.
2. You cannot try all combinations, because there are about 159 million million million of them.
3. But you know one thing: the porter always writes the current date on the pad by the door in the same format.
4. So instead of testing combinations against the lock, you test them against that known scrap of text. Any setting that could not have produced it is thrown away instantly.
5. The Bombe is a machine that throws away wrong settings at high speed. It does not decrypt anything. It hands humans a short list to try by hand.
6. The known scrap of text has a name in this trade: a **crib**.

Where this comparison breaks: the Bombe did not simply test and compare. It exploited a specific weakness, that Enigma never enciphered a letter as itself, and it used Welchman's diagonal board to propagate a single contradiction through the whole wiring at once, killing thousands of settings per test rather than one. It was a logic-propagation engine, not a brute-force counter.

■ 14.6.3 PLAIN – a worked example

1. Enigma's setting count, for the standard army machine of the war years:

Choice	Count	Notes
Rotor order	60	3 chosen from 5, in order
Rotor start positions	17,576	26 x 26 x 26

Choice	Count	Notes
Ring settings	676	Two that matter
Plugboard, 10 pairs	150.7 billion	The dominant term

2. Multiply those and you get about 1.59 times 10 to the power 20 settings.
3. Written out: roughly 159,000,000,000,000,000,000.
4. Testing one per second would take longer than the age of the universe.
5. Now use the crib. Suppose the intercepted text contains WETTERBERICHT, German for weather report, and you can guess where.
6. Enigma never maps a letter to itself. Slide the crib along the ciphertext. Any position where a letter lines up with itself is impossible, and is dropped without any machine at all.
7. The Bombe then took the surviving positions, wired up the logical chain of letter-to-letter implications, and spun until the electrical current found no contradiction. It stopped, and a human checked the stop.
8. A single Bombe run took around 20 minutes. By May 1945 Britain had about 155 three-rotor Bombes plus later four-rotor machines, running day and night.

■ 14.6.4 PLAIN – what is really happening inside

1. The first British Bombe, code-named Victory, was installed at Bletchley Park on 18 March 1940.
2. The improved version with Gordon Welchman's diagonal board, called Agnus and then Agnes, ran from 8 August 1940. That addition made the whole approach practical.
3. Bombes were built by the British Tabulating Machine Company at Letchworth, with Harold Keen as chief engineer. About 200 were built in Britain.
4. Be clear on this: the Bombe was electromechanical, not electronic, and it was not a computer. It ran one fixed search. It could not be programmed.
5. Colossus is a different animal. The Lorenz cipher used twelve wheels and was attacked statistically, not by simulation.
6. Bill Tutte worked out the complete internal structure of the Lorenz machine in 1941 and 1942 without ever seeing one, from intercepted traffic alone. It is one of the great feats of the war.
7. Max Newman designed a statistical attack. The first machine to run it, in 1943, was electromechanical and nicknamed Heath Robinson. It was too slow and its paper tapes tore.
8. Tommy Flowers, of the Post Office Research Station at Dollis Hill, argued that valves, what Americans call vacuum tubes, could do the job if you never switched them off. His superiors doubted it. He built it anyway, partly with his own money.
9. Colossus Mark 1, with 1,600 valves, was working at Bletchley Park in January 1944 and attacked its first real message on 5 February 1944.
10. Colossus Mark 2, with 2,400 valves, started work at 8 in the morning on 1 June 1944, five days before the Normandy landings. Ten Colossi were running by the end of the war.
11. Colossus was programmable only by switches and plugs, and it had no stored program and no general arithmetic. It was a special-purpose electronic machine. That is still a first.
12. Most Colossi were broken up on Churchill's orders. The work stayed secret until the mid-1970s, which is why the machine is missing from older history books. A working rebuild led by Tony Sale was completed in 2007.

■ 14.6.5 TECHNICAL – the engineer’s version

1. Konrad Zuse completed the **Z3** in Berlin and presented it on 12 May 1941.

Z3 property	Value
Technology	2,600 relays
Word length	22 bits, floating point
Clock	5 to 10 Hz
Memory	64 words
Add / multiply time	0.8 s / 3 s

2. The Z3 used binary floating point with a 14-bit mantissa, 7-bit exponent and sign bit, decades ahead of American practice. Program came from punched 35 mm film. It had no conditional branch.
3. Raúl Rojas showed in 1998 that the Z3 is Turing complete in principle, but only by an artificial construction that computes all branches. That is a theoretical result, not a description of how it was used.
4. The original Z3 was destroyed on 21 December 1943 in an Allied air raid on Berlin. A working replica was built in 1961.
5. The **Atanasoff-Berry Computer** was built at Iowa State College by John Vincent Atanasoff and Clifford Berry between 1939 and 1942.
6. It used about 280 vacuum tubes for arithmetic, was binary, held 60 numbers of 50 bits in rotating capacitor drums, and did about 30 additions per second.
7. Its regenerative capacitor memory, refreshed each rotation, is the direct ancestor of the refresh cycle in modern DRAM.
8. It solved systems of up to 29 simultaneous linear equations. It was not programmable and it never worked fully automatically; the card output unit was unreliable.
9. On 19 October 1973, in *Honeywell v. Sperry Rand*, United States District Judge Earl R. Larson invalidated the ENIAC patent, ruling that Eckert and Mauchly “derived that subject matter from one Dr. John Vincent Atanasoff”.
10. The honest version: that ruling is a legal finding about patent validity, not a historical consensus about who invented the computer. Historians still argue about how much Mauchly took from his 1941 visit to Atanasoff.
11. The **Harvard Mark I**, built by IBM as the Automatic Sequence Controlled Calculator, was presented to Harvard on 7 August 1944.
12. It was 51 feet long, 8 feet high, weighed about 4.7 short tons, and did 3 additions per second, a multiplication in 6 seconds and a division in 15.3 seconds. It was decimal, driven by a 50-foot shaft, and read its program from punched paper tape.
13. Howard Aiken led it. Grace Hopper, Richard Bloch and Robert Campbell were among its first programmers. John von Neumann ran implosion calculations on it for the Manhattan Project from 29 March 1944.

■ 14.6.6 WORDS – remember these

1. Enigma — the German rotor cipher machine — a stepping-rotor polyalphabetic substitution device with a plugboard.
2. Crib — a guessed piece of the plain message — known plaintext used to constrain a key search.
3. Bombe — the Bletchley setting-hunting machine, 1940 — an electromechanical parallel logic engine testing rotor hypotheses for contradiction.

4. Lorenz or Tunny — the high command teleprinter cipher — a 12-wheel additive stream cipher on 5-bit Baudot code.
5. Colossus — the 1943 to 1944 valve machine — the first large-scale electronic digital special-purpose computer, 1,600 then 2,400 valves.
6. Z3 — Zuse's 1941 relay machine — the first working programmable automatic binary floating-point computer.
7. ABC — the Atanasoff-Berry Computer of 1942 — the first electronic binary digital calculating device, non-programmable, with regenerative memory.

14.7 ENIAC and EDVAC

■ 14.7.1 PLAIN - in simple words

1. Artillery needs firing tables: for this gun, this shell, this angle, this wind, where does it land.
2. In 1943 the United States Army was drowning in the arithmetic. Each table needed thousands of trajectory calculations.
3. The Army funded John Mauchly and J. Presper Eckert at the Moore School of Electrical Engineering, University of Pennsylvania, to build a fully electronic calculator.
4. The result was **ENIAC**, the Electronic Numerical Integrator and Computer.
5. It filled a room, used nearly 18,000 vacuum tubes, weighed 30 tons and drew 150 kilowatts.
6. It did 5,000 additions a second, roughly a thousand times faster than the relay machines of the day.
7. It was finished in 1945 and shown to the press on 14 February 1946.
8. It had one enormous flaw: to change the problem, you rewired the machine.
9. That work took days or weeks of moving cables and setting thousands of switches by hand.
10. The people who did that work were six women, and for forty years almost nobody knew their names.

■ 14.7.2 PLAIN - a picture in your head

1. Think of an old telephone exchange, the kind with a wall of sockets and a human operator pushing plugs into them.
2. Now imagine that instead of connecting callers, each plug decides which number flows into which adding unit.
3. To ask the machine a different question you do not type anything. You unplug forty cables and plug them somewhere else, and turn 3,000 switches.
4. Then you spend two days finding the one cable you put in the wrong hole.
5. That is ENIAC. The program was the physical shape of the wiring.

Where this comparison breaks: a telephone exchange connects two points at a time and then releases them. ENIAC's plugboard connections were the permanent data paths for the whole run, and the timing of pulses down those cables was the control flow. It is closer to laying out a factory production line than to placing a call.

■ 14.7.3 PLAIN - a worked example

1. Here are ENIAC's real numbers, from the Moore School records.

Property	Value
Vacuum tubes	17,468

Property	Value
Crystal diodes	7,200
Relays	1,500
Resistors	70,000
Weight	Over 30 short tons
Floor space	About 1,800 sq ft
Power	150 kW
Additions per second	5,000
Cost	About \$487,000

2. You will often see 18,000 tubes quoted. The precise figure is 17,468. Both appear in reputable sources; the rounder number came from press releases.
3. Arithmetic detail that surprises people: ENIAC was **decimal**, not binary. It had 20 accumulators, each holding 10 decimal digits and a sign.
4. A digit was stored as a ring of ten flip-flops, one of which was on. That is ten tubes to hold what one bit-pair would hold in binary.
5. Reliability was the real problem. With 17,468 tubes, a failure every day or two was expected. Eckert ran the tubes far below rated voltage and never switched the machine off, since switch-on is when tubes die.
6. ENIAC's first serious job, in December 1945, was a feasibility calculation for the hydrogen bomb, run for six weeks. Its very first task was not ballistics at all.
7. It was switched off for the last time at 11:45 in the evening on 2 October 1955.

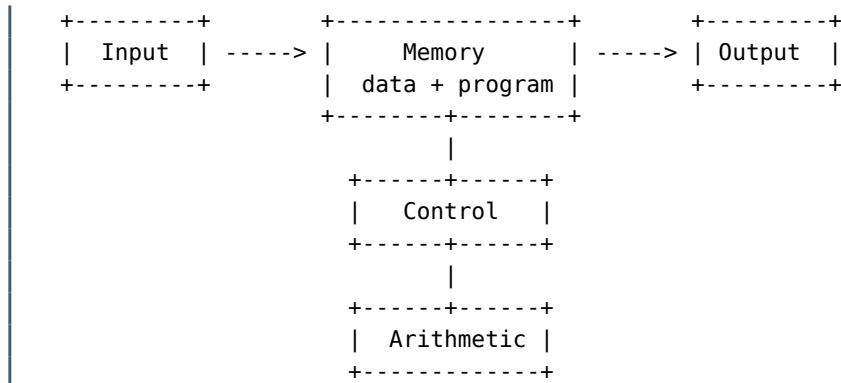
■ 14.7.4 PLAIN - what is really happening inside

1. The Army had already hired about eighty women as human computers, working out trajectories on desk calculators.
2. Six of them were selected in 1945 to program ENIAC: Kathleen McNulty, Frances Bilas, Betty Jean Jennings, Ruth Lichterman, Elizabeth Snyder and Marlyn Wescoff.
3. They are also known by their later married names: McNulty Mauchly Antonelli, Bilas Spence, Jennings Bartik, Lichterman Teitelbaum, Snyder Holberton, and Wescoff Meltzer.
4. They were given wiring diagrams and told to work it out. There was no manual, no programming language and no course, because none existed.
5. They learned the machine from its blueprints, invented debugging techniques, and worked out how to break a calculation into parallel units.
6. Betty Snyder Holberton went on to write the first sort-merge generator and worked on the standards for COBOL and Fortran. Kathleen McNulty helped devise subroutine techniques on later machines.
7. On the day ENIAC was shown to the press in February 1946, the men who built the hardware were named. The women were not invited to the celebratory dinner.
8. For decades their photographs were captioned as models posing with the equipment. The phrase used later to describe this is "refrigerator ladies", because women in technical photographs were assumed to be advertising props.
9. Kathryn Kleiman began tracking them down from 1986. Their story reached a wide audience with the 1997 Women in Technology International Hall of Fame inductions and the 2014 documentary *The Computers*.

10. This is not a footnote about fairness. Programming as a distinct discipline, separate from building hardware, starts here, and these six people invented much of its practice.

■ 14.7.5 TECHNICAL – the engineer’s version

1. While ENIAC was still being wired, its team already knew rewiring was unacceptable. The successor project was called EDVAC.
2. John von Neumann joined the group as a consultant in 1944. On 30 June 1945 Herman Goldstine distributed a 101-page document titled *First Draft of a Report on the EDVAC*.
3. It described a machine with five parts: a central arithmetic unit, a central control unit, memory, input and output.
4. Its crucial claim was that instructions and data live in the same read-write memory, so a program is just numbers, and a program can therefore be loaded, changed and even modified by another program.



5. The naming controversy is real and worth stating precisely.
 1. The document carried only von Neumann’s name. It was a draft circulated for comment, not a finished credited paper.
 2. Eckert and Mauchly said the stored-program concept came out of Moore School group discussions in 1944, before von Neumann arrived, and that he wrote up shared ideas in logical notation.
 3. Because the draft was circulated publicly more than a year before the patent filing, it counted as prior public disclosure. That helped invalidate the ENIAC patent claims later.
 4. Turing’s 1936 universal machine had already stated the idea mathematically, and von Neumann had read Turing’s paper.
6. Many historians therefore prefer the neutral terms “stored-program computer” or “Princeton architecture”. The label “von Neumann architecture” is a **convention**, and a slightly unfair one.
7. What the name refers to today is a specific structural property: a single address space and a single bus for both instructions and data. The alternative, separate instruction and data memories, is the **Harvard architecture**, named after the Harvard Mark I.
8. The consequence engineers still fight is the **von Neumann bottleneck**, a phrase John Backus coined in his 1977 Turing Award lecture: the processor can only go as fast as the single path to memory allows.
9. Nearly every modern chip is a hybrid. Level 1 caches are usually split into separate instruction and data caches, which is Harvard-like, over a unified main memory, which is von Neumann. You can see this on Linux with `lscpu`, which lists L1d and L1i separately.
10. EDVAC itself was delivered in 1949 and became properly operational in 1951, by which time British machines had beaten it into service.

■ 14.7.6 WORDS – remember these

1. ENIAC — the 1945 American electronic calculator — a decimal, plugboard programmed, 17,468-tube general-purpose electronic computer.
2. Accumulator — a box that holds a running total — a register that stores an arithmetic result, ten decimal digits on ENIAC.
3. Stored program — instructions kept in memory like data — a single writable address space holding both code and operands.
4. von Neumann architecture — the 1945 five-part design — a stored-program machine with unified instruction and data memory.
5. Harvard architecture — separate stores for code and data — physically distinct instruction and data memories and buses.
6. von Neumann bottleneck — the narrow pipe to memory — the throughput limit imposed by a single shared processor-memory path, named by Backus in 1977.

14.8 The first stored-program computers

■ 14.8.1 PLAIN – in simple words

1. After 1945 the race was to build a machine that kept its program in memory.
2. The hard part was not logic. It was memory. Nobody had a fast, cheap, rewritable store.
3. Two solutions appeared. One used sound. One used a television tube.
4. The sound one is the **mercury delay line**. Send a pulse of sound down a tube of mercury, catch it at the far end, and send it round again. The data is literally in flight.
5. The television one is the **Williams tube**. Draw dots on the inside of a cathode ray tube. The charge left behind is the bit. Read it with a metal plate on the glass.
6. At Manchester, Freddie Williams and Tom Kilburn made the tube work, and then built the smallest possible computer to prove it.
7. That machine, the Small-Scale Experimental Machine, nicknamed the **Baby**, ran the world's first stored program on 21 June 1948.
8. At Cambridge, Maurice Wilkes built **EDSAC**, which first ran on 6 May 1949 and was the first machine to give a regular computing service to other people.
9. Manchester's full-size machine became the Manchester Mark 1, and the company Ferranti turned it into a product.
10. The **Ferranti Mark 1**, delivered in February 1951, was the first commercially available general-purpose electronic computer in the world.
11. In America, **UNIVAC I** was accepted by the Census Bureau on 31 March 1951 and became famous on election night in 1952.

■ 14.8.2 PLAIN – a picture in your head

1. Picture a long queue of people passing a whispered message down the line and back to the front, over and over, so it is never forgotten.
2. That is a mercury delay line. The message exists only as sound travelling through liquid metal.
3. To read a particular bit, you wait for it to come round. If you just missed it, you wait a full lap.
4. Now picture instead a chalk mark on a blackboard that fades in a fraction of a second, so a helper walks along constantly redrawing every mark.
5. That is a Williams tube. Any dot you can see, you can read at once, in any order.

Where this comparison breaks: the queue is not just slow, it is strictly sequential, which changed how programs were written. On EDSAC, and even more on drum-memory machines, programmers

placed instructions at positions on the loop so the next one arrived just as it was needed. That practice is called optimum coding, and it is the ancestor of today's cache-aware programming.

■ 14.8.3 PLAIN – a worked example

1. The Baby's first program, written by Tom Kilburn, found the highest proper factor of 2 to the power 18, which is 262,144.
2. The method was the crudest possible: try every candidate downwards from 262,143 and test by repeated subtraction.
3. It was 17 instructions long and it lived in a Williams tube holding 32 words of 32 bits, which is 1,024 bits in total.
4. It ran for about 52 minutes and performed roughly 3.5 million operations, which is around 1,100 operations per second.
5. The answer, of course, is 131,072. The point was never the answer. The point was that the program was in memory.
6. Compare that with EDSAC's first run on 6 May 1949, which printed a table of the squares of 0 to 99, and then a list of prime numbers.

Machine	First ran	Memory
Manchester Baby	21 Jun 1948	32 words, Williams tube
EDSAC	6 May 1949	512 words, mercury
Manchester Mark 1	Apr 1949	Tube plus drum
Ferranti Mark 1	Feb 1951	256 words, plus drum
UNIVAC I	1951	1,000 words, mercury

■ 14.8.4 PLAIN – what is really happening inside

1. EDSAC's real importance is not the hardware. It is that Cambridge treated computing as a service and a discipline.
2. David Wheeler invented the **subroutine** there: a block of instructions you can call from anywhere and return from. The return mechanism is still called the Wheeler jump.
3. Wilkes, Wheeler and Gill published *The Preparation of Programs for an Electronic Digital Computer* in 1951. It is the first programming textbook, and it introduced the idea of a library of reusable routines.
4. Wheeler also received what is generally described as the world's first doctorate in computer science, in 1951.
5. EDSAC also produced the first business computing. The catering company J. Lyons and Company built a copy called LEO, the Lyons Electronic Office, which ran its first routine business job on 17 November 1951, valuing a bakery's output.
6. A tea shop company thus ran payroll on a computer before most governments did. That is a genuine and under-told fact.
7. The Manchester Mark 1 contributed the **index register**, which Manchester called a B-line: a register whose value is added to an address before use.
8. That one idea makes loops over arrays possible without a program rewriting its own instructions, and every processor since has it.

■ 14.8.5 TECHNICAL – the engineer’s version

1. Williams-Kilburn tube: a standard CRT storing charge on the phosphor. Typical capacity 1,024 to 2,048 bits per tube. Access time about 10 to 30 microseconds and, crucially, random access. It needed constant refresh and was sensitive to nearby electrical noise, even to someone opening a door.
2. Mercury delay line: a tube roughly 1.5 metres long. Sound travels in mercury at about 1,450 metres per second, giving a loop time near 1 millisecond and holding several hundred bits. Average latency is half a loop.
3. EDSAC: 3,000 valves, about 11 kW, 512 words of 17 bits initially, cycle time 1.5 ms for most instructions and 6 ms for multiplication. It ran until 11 July 1958.
4. Ferranti Mark 1: delivered to the University of Manchester in February 1951, publicly demonstrated in July 1951. Around nine machines including the improved Mark 1 star were built between 1951 and 1957.
5. Priority claims, stated carefully, because people argue about them:
 1. First stored program executed: Manchester Baby, 21 June 1948.
 2. First regular computing service: EDSAC, from 1949.
 3. First commercially available general-purpose computer delivered: Ferranti Mark 1, February 1951.
 4. First American commercial delivery: UNIVAC I, accepted by the Census Bureau on 31 March 1951, physically installed later that year.
 5. Zuse’s Z4 was delivered to ETH Zurich in July 1950 and was electromechanical, so it is usually excluded from electronic firsts.
6. UNIVAC I specifications: 5,200 vacuum tubes, about 125 kW, 1,000 words of 12 characters in mercury memory, about 1,905 operations per second, 46 machines built, and a price that rose from a quoted \$159,000 to well over a million dollars.
7. The 1952 election. On CBS, on 4 November 1952, UNIVAC was given 5.5 per cent of the returns and predicted a landslide for Dwight Eisenhower over Adlai Stevenson, at 438 electoral votes to 93.
8. Nobody believed it, because pollsters expected a close race. CBS staff adjusted a trend factor from 40 per cent to 4 per cent to produce a respectable-looking 268 to 263, and broadcast that instead.
9. The actual result was 442 to 89. The original prediction was within four electoral votes. Late that night Remington Rand’s Arthur Draper admitted on air that the machine had been right and had been overruled.
10. That broadcast, more than any technical paper, is what put the word “computer” into ordinary speech.

■ 14.8.6 WORDS – remember these

1. Mercury delay line — memory made of sound in a tube — a sequential-access acoustic recirculating store, about 1 ms loop time.
2. Williams tube — memory made of dots on a screen — a CRT charge-storage random-access memory, about 1 to 2 kilobits per tube.
3. Baby — the Manchester test machine of 1948 — the Small-Scale Experimental Machine, first to execute a program held in memory, 21 June 1948.
4. Subroutine — a reusable block of instructions — a callable routine with a return mechanism, invented by David Wheeler on EDSAC.
5. Index register — a register that shifts an address — a B-line, added to an operand address before memory access, from the Manchester Mark 1.

6. LEO — the tea company's computer, 1951 — Lyons Electronic Office, the first computer used for routine commercial work.

14.9 Transistors, mainframes and minicomputers

■ 14.9.1 PLAIN – in simple words

1. A vacuum tube is a small glass bulb that controls electricity. It works, but it is hot, fragile, power-hungry and it burns out.
2. On 23 December 1947, John Bardeen and Walter Brattain at Bell Labs demonstrated the **transistor**: a lump of germanium doing the same job with no glass, no heater and almost no power.
3. William Shockley followed with the more manufacturable junction transistor, published in 1951. The three shared the Nobel Prize in Physics in 1956.
4. Machines built from transistors were smaller, cooler and ran for months instead of hours between failures.
5. Through the 1950s IBM sold its 700 series, built from tubes, and then the 7000 series, built from transistors.
6. Every one of those machines had its own instruction set, so software written for one would not run on the next.
7. On 7 April 1964 IBM announced the **System/360**: one family of machines, from small to huge, all running the same programs.
8. That single decision made software an asset you keep instead of a cost you repeat, and it made IBM dominant for twenty years.
9. Meanwhile a small company called Digital Equipment Corporation went the other way, building machines cheap enough for one laboratory to own.
10. Those were **minicomputers**, and they took computing out of the glass room and put it next to the people doing the work.

■ 14.9.2 PLAIN – a picture in your head

1. A mainframe is a city power station. It is enormous, it is behind a fence, and you apply for a connection.
2. Programs arrive as decks of cards, are run overnight in a queue, and results come back as printout the next morning. That is **batch processing**.
3. A minicomputer is a generator in your own workshop. Smaller, noisier, less efficient, but yours, and switched on when you want it.
4. Nobody schedules you. You sit at the machine, you type, it answers.
5. System/360 is a third idea: not one power station but a whole product range, from a small generator to a station, all taking the same plug.

Where this comparison breaks: minicomputers were not personal. A PDP-11 cost tens of thousands of dollars and served a whole department, usually through several terminals at once. The truly personal machine is still a decade away in this story.

■ 14.9.3 PLAIN – a worked example

1. Here is why compatibility mattered, in money.
2. Say a bank has 200 programs, each taking 6 months of one programmer's time to write. That is 100 programmer-years of investment.
3. Before 1964, buying a bigger machine meant rewriting most of that. The software bill could exceed the hardware bill.

4. With System/360, the same program ran on a Model 30 and on a Model 75, which was roughly fifty times faster.
5. You bought a bigger box and your software simply ran faster. Nothing else changed.

Machine	Year	Note
IBM 701	1952	Tubes, 19 built
IBM 650	1954	Drum, about 2,000 built
IBM 704	1954	Fortran's first host
IBM 7090	1959	Transistorized 709
IBM S/360 Model 30	1965	34,500 instr/sec
IBM S/360 Model 91	1967	16.6 million instr/sec

■ 14.9.4 PLAIN – what is really happening inside

1. The trick that made one family possible is **microprogramming**, invented by Maurice Wilkes at Cambridge in 1951.
2. Instead of building each instruction directly out of logic gates, you build a small, fast inner machine, and store a tiny program for each instruction in read-only memory.
3. A cheap model runs the same instruction set with slow, simple hardware and more microcode steps. An expensive model uses fast hardware and fewer steps.
4. Same behaviour, wildly different cost. That is how one architecture spanned a fifty-to-one performance range.
5. Digital Equipment Corporation, founded in 1957 by Ken Olsen and Harlan Anderson, attacked from below.
6. The **PDP-1** of 1959 cost \$120,000 when mainframes cost millions. Fifty-three were built. On one of them, in 1962, Steve Russell and friends at MIT wrote the game Spacewar, one of the first video games.
7. The **PDP-8**, launched on 22 March 1965 at \$18,500, was the first computer sold for under \$20,000. Over 50,000 were sold, and more than 300,000 across the whole family.
8. The **PDP-11**, announced in January 1970, sold around 600,000 units over its life. Unix was first written for it, and the C language grew up on it.

■ 14.9.5 TECHNICAL – the engineer's version

1. System/360 announcement, 7 April 1964: six models initially, numbered 30, 40, 50, 60, 62 and 70. The last three were replaced before shipping by the 65 and 75.
2. Development cost is usually quoted as about \$5 billion, from a September 1966 Fortune article calling it IBM's five-billion-dollar gamble. Adjusted for inflation that is roughly \$50 billion today.
3. Architectural decisions from System/360 that are still with us:
 1. The 8-bit **byte** as the standard addressable unit. Before 360, machines used 6-bit characters and word sizes of 36, 48 or 60 bits.
 2. 32-bit words, 24-bit addressing, and general-purpose registers.
 3. A separate channel subsystem for input and output, which is the ancestor of DMA controllers.
4. Gene Amdahl was chief architect. Fred Brooks managed the OS/360 software project, and later wrote *The Mythical Man-Month* in 1975 about how badly it went. Its most quoted line: adding people to a late software project makes it later.
5. IBM's dominance drew the United States Department of Justice, which filed an antitrust suit on 17 January 1969. It ran for thirteen years and was dropped on 8 January 1982 as being without merit.

6. The rest of the industry was known as the BUNCH: Burroughs, UNIVAC, NCR, Control Data and Honeywell. The older joke was “IBM and the seven dwarfs”.
7. Mainframes did not die. IBM z16 systems, announced in April 2022, still run code compiled for System/360 in 1964. That is 60 years of binary compatibility, the longest unbroken run in the industry.

■ 14.9.6 WORDS – remember these

1. Transistor — a solid switch with no glass bulb — a semiconductor device controlling current, demonstrated at Bell Labs in December 1947.
2. Mainframe — a big shared central computer — a high-throughput, high-reliability system optimized for input and output volume.
3. Minicomputer — a machine a department can afford — a mid-range system, typically 12 to 16 bits, priced from about \$18,500 in 1965.
4. Batch processing — hand in your job, come back tomorrow — non-interactive scheduled execution of queued jobs.
5. Microprogram — a tiny program inside the processor — microcode in control store implementing each machine instruction, invented by Wilkes in 1951.
6. Binary compatibility — old programs still run — the ability to execute unmodified object code on a newer implementation.

14.10 The integrated circuit and Silicon Valley

■ 14.10.1 PLAIN – in simple words

1. By 1958 a computer was thousands of separate parts joined by hand-soldered wires. Every joint was a chance to fail.
2. Engineers called this the tyranny of numbers: to make a machine more capable you needed more parts, and more parts meant it broke sooner.
3. On 12 September 1958, Jack Kilby at Texas Instruments demonstrated a different idea: build all the parts out of one piece of semiconductor, so there are no joints at all.
4. His device worked but used tiny gold wires bonded by hand across the surface. It was a proof, not a product.
5. In 1959 Robert Noyce at Fairchild Semiconductor solved the manufacturing problem, using a silicon wafer and printing the connections on as a flat layer of aluminium.
6. That is the **integrated circuit**, or chip.
7. Both men are credited. Kilby received the Nobel Prize in Physics in 2000; Noyce had died in 1990 and Nobel prizes are not awarded posthumously.
8. Noyce’s version won commercially because it could be made in quantity.
9. The reason it could be made in quantity was a process invented by his colleague Jean Hoerni, also in 1959, called the **planar process**.
10. All of this happened in a strip of orchards south of San Francisco, which a journalist named Silicon Valley in 1971.

■ 14.10.2 PLAIN – a picture in your head

1. Think of printing a newspaper instead of writing each copy by hand.
2. Old way: draw every letter individually, then glue the pages together.
3. New way: make one master plate, then print thousands of identical copies at almost no extra cost per copy.

4. A chip is printed. Light is shone through a mask onto a light-sensitive coating on the silicon, and the pattern is etched in.
5. Printing 10 components costs the same as printing 10 million. That is the entire economics of the computer industry in one sentence.

Where this comparison breaks: newspapers do not care about a single dust speck. A chip does. One particle in the wrong place kills the die, so yield, the fraction of good chips per wafer, dominates cost. That is why chips are made in cleanrooms with fewer particles per cubic metre than an operating theatre.

■ 14.10.3 PLAIN – a worked example

1. The planar process in five steps, which is still roughly what happens today.
 1. Oxidize : grow a glass layer of SiO₂ on the silicon
 2. Mask : coat with resist, shine light through a mask
 3. Etch : dissolve the exposed oxide, opening windows
 4. Diffuse : drive dopant atoms in through the windows
 5. Metallize : evaporate aluminium on top, then pattern it
2. The key point is step 1. The oxide stays on as a permanent protective and insulating layer.
3. Because the surface is flat and insulated, metal wiring can be laid across the top without shorting to anything underneath.
4. Kilby's germanium device had no such layer, so his connections were flying wires. Hoerni's flat oxide is what made mass production possible.
5. Kurt Lehovec at Sprague Electric added the other missing piece in 1959: isolating neighbouring components with reverse-biased junctions.

■ 14.10.4 PLAIN – what is really happening inside

1. William Shockley left Bell Labs and set up Shockley Semiconductor Laboratory in Mountain View, California, in 1956, the year he won the Nobel Prize.
2. He hired brilliant people and managed them badly. He posted salaries publicly and required staff to take lie detector tests.
3. On 18 September 1957 eight of them resigned together. Shockley called them the **traitorous eight**.
4. They were Julius Blank, Victor Grinich, Jean Hoerni, Eugene Kleiner, Jay Last, Gordon Moore, Robert Noyce and Sheldon Roberts.
5. Sherman Fairchild financed them with a loan of \$1.38 million, and Fairchild Semiconductor was born.
6. Fairchild then leaked people the way Shockley had. Companies founded by ex-Fairchild staff are called Fairchildren, and there are dozens.
7. On 18 July 1968 Noyce and Moore left to found Intel, backed by the venture capitalist Arthur Rock.
8. On 1 May 1969 Jerry Sanders, Fairchild's head of marketing, founded Advanced Micro Devices with seven colleagues.
9. Eugene Kleiner went on to co-found the venture firm Kleiner Perkins in 1972, which is how the engineering diaspora became a financial one too.
10. The name arrived on 11 January 1971, when the journalist Don Hoefler ran a series titled "Silicon Valley U.S.A." in the trade paper Electronic News.
11. Silicon is the element the chips are made from; the valley is the Santa Clara Valley. The name did not become common speech until the early 1980s.

■ 14.10.5 TECHNICAL – the engineer’s version

1. Kilby’s 12 September 1958 device: a phase-shift oscillator on a germanium bar about 11 by 1.6 millimetres, with gold wire interconnect. Texas Instruments filed the patent in February 1959.
2. Noyce’s Fairchild device: silicon, oxide-passivated by Hoerni’s planar process, with evaporated aluminium interconnect. Patent filed 30 July 1959.
3. The two companies litigated for a decade and settled with cross-licensing in 1966. Courts eventually favoured Noyce on the interconnect claims and Kilby on the integration concept.
4. Gordon Moore’s observation was published in Electronics magazine on 19 April 1965. He noted the number of components per chip had roughly doubled each year, and predicted it would continue for a decade.
5. In 1975 Moore revised the doubling period to about two years. The often-quoted eighteen months is a later paraphrase attributed to Intel colleague David House, combining transistor count and speed.
6. Standard, convention or observation? Moore’s law is none of a standard, a law of physics or a theorem. It is an economic observation that became a planning target the industry then organized itself to meet.

Item	Year	Transistors
Kilby prototype IC	1958	About 1
Intel 4004	1971	2,300
Intel 80386	1985	275,000
Intel Pentium	1993	3.1 million
Apple M1	2020	16 billion
Nvidia Blackwell B100	2024	208 billion

■ 14.10.6 WORDS – remember these

1. Integrated circuit — many parts on one piece of silicon — a monolithic circuit fabricated on a single semiconductor die.
2. Planar process — the flat oxide-layer way of making chips — Hoerni’s 1959 oxide passivation and diffusion method enabling deposited interconnect.
3. Wafer — the silicon disc chips are printed on — a monocrystalline substrate, today 300 mm in diameter.
4. Yield — how many chips per wafer actually work — the ratio of functional die to total die, the dominant cost factor.
5. Moore’s law — chips double in capacity every couple of years — the 1965 observation on component count per die, revised to two years in 1975.
6. Traitorous eight — the group who quit Shockley in 1957 — the founders of Fairchild Semiconductor and, indirectly, of Silicon Valley.

14.11 The microprocessor

■ 14.11.1 PLAIN – in simple words

1. By 1969 you could put thousands of transistors on one chip, but a computer's processor was still spread over many chips and several boards.
2. In April 1969 a Japanese calculator company called Basicom asked Intel, then a young memory company, for a set of chips for a printing calculator.
3. Basicom's own design used twelve custom chips, each hardwired for one part of the job.
4. An Intel engineer, Ted Hoff, said that was backwards. Build one small general-purpose processor, and put the calculator's behaviour in a program in read-only memory.
5. Stanley Mazor helped work out the instruction set. Basicom's engineer Masatoshi Shima wrote much of the calculator software and checked the design.
6. Nobody at Intel could actually build it until Federico Faggin joined in April 1970 and invented the circuit techniques needed to fit it on one die.
7. The result was the **Intel 4004**, announced on 15 November 1971.
8. It had 2,300 transistors, a 4-bit word, ran at up to 740 kilohertz, and cost \$60.
9. That single chip had roughly the computing power of ENIAC, which had filled a room in 1946.
10. Intel then bought back the exclusive rights from Basicom, which was in financial trouble, by re-funding \$60,000 of development cost. That is how a calculator part became a product anyone could buy.

■ 14.11.2 PLAIN – a picture in your head

1. Think of a kitchen that makes one dish. Every worktop, jig and mould is shaped for that dish alone. Change the menu and you rebuild the kitchen.
2. Now think of a kitchen with one general chef, a knife, a pan and a recipe card.
3. The second kitchen is slower at the one dish, but it can cook anything, and changing the menu costs the price of a card.
4. The 4004 is the general chef. The read-only memory chip is the recipe card.
5. Because one design serves a thousand products, you can afford to make ten million of them, and the price collapses.

Where this comparison breaks: general-purpose chips did not simply beat custom ones. Custom silicon is still faster and cheaper per unit for very high volume fixed jobs, which is why your phone contains dozens of fixed-function blocks for video, radio and camera work alongside its general processors.

■ 14.11.3 PLAIN – a worked example

1. Compare the 4004 with ENIAC, which is a fair fight on arithmetic rate.

Property	ENIAC, 1946	Intel 4004, 1971
Size	About 1,800 sq ft	12 square mm
Weight	Over 30 tons	Under 1 gram
Power	150 kW	Under 1 watt
Additions per second	5,000	About 92,000
Cost	About \$487,000	\$60

2. Twenty-five years took a room-sized machine to a fingernail-sized chip, at about one eight-thousandth of the price.
3. The 4004 was made on a 10 micrometre pMOS silicon-gate process. Today's leading process is around 3 nanometres, which is roughly 3,000 times finer in each direction.

■ 14.11.4 PLAIN – what is really happening inside

1. The 4004 was a starting gun, not a finish line. The useful chips came next.
2. The **Intel 8008** arrived in April 1972. It was 8-bit and came from a separate contract for the Data-point 2200 terminal, whose maker rejected it.
3. The **Intel 8080** arrived in April 1974, with about 4,500 transistors and a 2 megahertz clock. It was fast enough to be the heart of a real computer, and it powered the Altair 8800.
4. Motorola answered with the **6800** in 1974. It needed only a single 5-volt supply, which made board design far simpler.
5. Federico Faggin left Intel at the end of 1974 and founded Zilog. With Masatoshi Shima he designed the **Z80**, released in July 1976.
6. The Z80 ran all 8080 software, added instructions and registers, needed one power supply, and refreshed dynamic memory itself. It became the most widely used 8-bit processor of the era.
7. The turning point for ordinary people was price. In August 1974 the Motorola 6800 and Intel 8080 were introduced at around \$360 each.
8. Chuck Peddle had worked on the 6800 at Motorola. In August 1974 he and seven colleagues left for MOS Technology.
9. In September 1975, at the Wescon show in San Francisco, they sold the **MOS 6502** for \$25, out of jars on a table, because nobody believed the price otherwise.
10. Twenty-five dollars instead of three hundred and sixty is what made a home computer possible. The 6502 went into the Apple I and II, the Commodore PET and 64, the BBC Micro, the Atari 2600 and the Nintendo Entertainment System.

■ 14.11.5 TECHNICAL – the engineer's version

1. Faggin's enabling technique was **silicon gate MOS**, which he had developed at Fairchild in 1968. Using polysilicon rather than aluminium for the gate makes it self-aligning, which shrinks parasitic capacitance and allows buried contacts. Without it the 4004 would not have fitted on one die.
2. The 4004 shipped as part of the MCS-4 chip set: 4001 ROM, 4002 RAM, 4003 shift register and 4004 CPU. On its own the processor is not a computer.
3. It used a 4-bit data bus, a 12-bit address space of 4 kilobytes of program memory, 46 instructions, and a 16-level address stack in hardware.

Processor	Introduced	Bits	Transistors
Intel 4004	Nov 1971	4	2,300
Intel 8008	Apr 1972	8	3,500
Intel 8080	Apr 1974	8	4,500
Motorola 6800	1974	8	About 4,100
MOS 6502	Sep 1975	8	About 3,510
Zilog Z80	Jul 1976	8	About 8,500

4. The 6502 was cheap for two engineering reasons, not just aggressive pricing. First, MOS Technology fixed mask defects rather than discarding the wafer, raising yield dramatically. Second, the design is minimal: three 8-bit registers, a hardware stack fixed at page one, and no microcode.
5. Its zero-page addressing mode treats the first 256 bytes of memory as quasi-registers, which is how a chip with so few registers stayed fast.

6. Legal note: MOS Technology first shipped the 6501, which was pin-compatible with the Motorola 6800. Motorola sued in November 1975. MOS withdrew the 6501 and settled in 1976, paying \$200,000. The 6502 was not pin-compatible and survived.

■ 14.11.6 WORDS – remember these

1. Microprocessor — a whole processor on one chip — a monolithic CPU, first commercially released as the Intel 4004 in November 1971.
2. Silicon gate — a way of building transistors with polysilicon gates — self-aligned MOS process reducing gate capacitance, Faggin 1968.
3. Instruction set — the list of things a chip understands — the architectural contract between hardware and compiled code.
4. Read-only memory — the chip holding the fixed program — ROM, non-volatile storage written at manufacture or once thereafter.
5. Zero page — the first 256 bytes of memory on a 6502 — a short-addressed region used as extended register space.

14.12 The personal computer revolution

■ 14.12.1 PLAIN – in simple words

1. In December 1974 a magazine cover changed everything. The January 1975 issue of Popular Electronics showed a blue box called the **Altair 8800**.
2. It was made by MITS, a small company in Albuquerque run by Ed Roberts, and it cost \$439 as a kit or \$621 assembled.
3. It had an Intel 8080, 256 bytes of memory, no keyboard, no screen and no storage. You entered programs with switches and read answers as lights.
4. MITS hoped to sell a few hundred. They had 1,000 orders in February 1975 alone, and had shipped over 5,000 by August.
5. Two young men in Boston saw the cover. Bill Gates and Paul Allen phoned MITS and claimed to have a BASIC language for it. They did not.
6. They wrote it in eight weeks on a simulated 8080 running on a PDP-10, having never touched an Altair. Paul Allen flew to Albuquerque, and it worked first time.
7. They founded **Microsoft** on 4 April 1975 to sell it.
8. On 5 March 1975 about thirty hobbyists met in Gordon French's garage in Menlo Park, California. That was the first **Homebrew Computer Club**.
9. Two members were Steve Wozniak and Steve Jobs. Apple came out of that room.
10. From there the story accelerates: Apple in 1976 and 1977, three rival home computers in 1977, the first spreadsheet in 1979, and IBM's entry in 1981.

■ 14.12.2 PLAIN – a picture in your head

1. Think of the first cheap cars. They were sold as parts, bought by mechanics, and driven by people who could fix them at the roadside.
2. Then somebody sold a car with a roof, a starter motor and a key, and ordinary people bought it.
3. The Altair is the kit. The Apple II with a plastic case and a power supply is the car with a roof.
4. But nobody buys a car because it has a roof. They buy it to get somewhere.
5. The place people wanted to get to turned out to be a spreadsheet.

Where this comparison breaks: cars had obvious uses before they were easy to drive. In 1977 nobody could say what a home computer was for. That question was genuinely open, and shop owners struggled to answer it, until VisiCalc gave a concrete answer for businesses in 1979.

■ 14.12.3 PLAIN – a worked example

1. The three machines of 1977, often called the 1977 trinity, arrived within months of each other and defined the market.

Machine	Launched	Price and CPU
Apple II	10 Jun 1977	\$1,298, 6502
Commodore PET 2001	Oct 1977	\$795, 6502
TRS-80 Model I	Aug 1977	\$599.95, Z80

2. All three included BASIC in read-only memory, so the machine was usable the moment it was switched on.
3. The TRS-80 sold about 55,000 in its first year, helped by Radio Shack's 3,000 stores. Over 200,000 were sold in total.
4. The Apple II sold about 6 million across all models between 1977 and 1993.
5. Apple Computer had been founded on 1 April 1976 by Steve Jobs, Steve Wozniak and Ronald Wayne. Wayne sold his ten per cent share back for \$800 twelve days later.
6. The Apple I went on sale in July 1976 at \$666.66. About 200 were built. The first order was 50 units from Paul Terrell's Byte Shop at \$500 each, on condition they were assembled boards.

■ 14.12.4 PLAIN – what is really happening inside

1. VisiCalc was released on 17 October 1979 for the Apple II, at under \$100.
2. Dan Bricklin, a Harvard business student, imagined it. Bob Frankston wrote it. Personal Software, later VisiCorp, published it.
3. It is the first **killer application**: a program so useful that people buy the hardware in order to run it.
4. More than 700,000 copies sold in six years. Reports at the time suggested over a quarter of Apple IIs sold in 1979 were bought to run it.
5. That commercial signal is what brought IBM in. IBM normally took years to develop a product. This time it gave a team in Boca Raton, Florida, one year.
6. The project was code-named Chess, led by William C. Lowe and then Don Estridge. To hit the deadline they broke every IBM rule and bought parts from outside.
7. The **IBM PC model 5150** was announced on 12 August 1981: an Intel 8088 at 4.77 megahertz, 16 kilobytes of memory as standard, from \$1,565.
8. The decision that made history was openness. IBM published a technical reference manual containing the full circuit diagrams and the complete source listing of the BIOS, the start-up firmware.

9. That let anyone build add-in cards and software. It also let anyone build a copy of the machine, provided they could produce a BIOS without copying IBM's code.
10. Compaq did exactly that. One team of engineers documented what the IBM BIOS did; a second team, who had never seen IBM's code, wrote a new one from that description alone. That procedure is called **clean room** design.
11. The Compaq Portable was announced in November 1982 and shipped in March 1983 at \$2,995. Compaq sold 53,000 units and made \$111 million in its first year, an American record at the time.
12. Once Phoenix Technologies began selling a compatible BIOS to anyone in 1984, the clone industry exploded, and the PC stopped being IBM's product and became an industry standard.

■ 14.12.5 TECHNICAL – the engineer's version

1. How Microsoft got MS-DOS, with dates, because this is constantly told wrong.
 1. IBM approached Digital Research, maker of CP/M, the dominant 8-bit operating system. No deal was reached. The famous story that Gary Kildall went flying instead of meeting IBM is disputed by participants and should not be stated as fact.
 2. Microsoft agreed to supply an operating system it did not have.
 3. Tim Paterson at Seattle Computer Products had written 86-DOS, nicknamed QDOS for Quick and Dirty Operating System, starting in April 1980 and shipping in August 1980. It was modelled on CP/M but for the 8086.
 4. In December 1980 Microsoft licensed it non-exclusively for \$25,000.
 5. In July 1981 Microsoft bought all rights for a further \$50,000, and hired Paterson.
 6. IBM shipped it in August 1981 as PC DOS 1.0.
2. The commercial masterstroke was not the purchase. It was the contract: Microsoft licensed to IBM non-exclusively and kept the right to sell MS-DOS to anyone else. Within a year it had licensed it to more than 70 companies.
3. So Bill Gates did not write MS-DOS, and Microsoft did not originally own it. Both facts are commonly got wrong.
4. IBM PC 5150 hardware, for the record: Intel 8088, 4.77 MHz, 8-bit external bus with 16-bit internal registers, 1 megabyte address space, five expansion slots, cassette port, and 16 to 64 kilobytes on the system board.
5. The 8088 was chosen over the faster 8086 precisely because its 8-bit external bus let IBM reuse cheap existing 8-bit support chips.
6. Three operating systems were offered at launch: PC DOS at \$40, CP/M-86 at \$240, and UCSD p-System. Price decided the outcome.
7. The legal foundation for clones is that copyright protects the expression of the BIOS code, not the interface it presents. Clean room procedure documents that the new expression was created independently. This principle was reinforced in *Computer Associates v. Altai* in 1992.

■ 14.12.6 WORDS – remember these

1. Kit computer — a computer you solder together — an unassembled system such as the Altair 8800 of January 1975.
2. Killer application — the program worth buying a computer for — software whose demand drives hardware adoption, first VisiCalc in 1979.
3. BIOS — the start-up firmware in a PC — Basic Input/Output System, the low-level routines between hardware and operating system.
4. Clean room design — copying behaviour without copying code — reimplementing from a specification written by a separate team with no access to the original code.

5. Clone — a machine that runs the same software — a hardware-compatible system using an independently written BIOS.
6. Open architecture — published plans anyone can build for — documented interfaces and bus specifications permitting third-party hardware.

14.13 The graphical interface

■ 14.13.1 PLAIN – in simple words

1. Until the 1980s you talked to a computer by typing commands and reading text.
2. The alternative was demonstrated on 9 December 1968 in San Francisco, by Douglas Engelbart of the Stanford Research Institute.
3. In ninety minutes he showed a mouse, windows on screen, clickable linked text, shared editing of a document with a colleague miles away, and video of that colleague on the same screen.
4. His system was called NLS, the oN-Line System. The demonstration is now called the Mother of All Demos.
5. Almost nothing he showed was available to buy for fifteen years.
6. In 1970 Xerox opened a research centre in Palo Alto, called PARC, and hired many of Engelbart's people.
7. On 1 March 1973 PARC had the **Alto** working: a personal computer with a screen made of individually controlled dots, a mouse, and a network socket.
8. PARC also produced **Smalltalk**, a programming system by Alan Kay's group in which overlapping windows, pop-up menus and the whole desktop idea were worked out.
9. Steve Jobs visited PARC in December 1979 and saw all of it.
10. Apple shipped the **Lisa** on 19 January 1983 at \$9,995, and the **Macintosh** on 24 January 1984 at \$2,495.
11. Microsoft shipped **Windows 1.0** on 20 November 1985 at \$99, and Apple sued in 1988.

■ 14.13.2 PLAIN – a picture in your head

1. A command line is like ordering food by writing a note in a language the kitchen understands, and getting a note back.
2. A graphical interface is like walking to a counter where the dishes are laid out, and pointing.
3. Pointing needs three things: a picture of the choices, a way to point, and instant feedback when you do.
4. The picture needs a screen where every dot is separately controlled, which is called a **bitmap** display. That is far more memory than a text screen.
5. The pointing needs a mouse. The feedback needs the machine to redraw fast enough that the arrow feels attached to your hand.
6. That combination is why the idea waited from 1968 to 1984. It needed memory to become cheap.

Where this comparison breaks: pointing at a counter shows you only what is on display. A command line can express things no menu contains, such as “rename every file recorded before March and move it”. Neither replaced the other, which is why every professional operating system in 2026 still has a terminal.

■ 14.13.3 PLAIN – a worked example

1. Memory is the whole reason for the delay. Count the bits.
2. A 1970s text terminal: 80 columns by 24 rows of characters, one byte each. 80 times 24 is 1,920 bytes.

3. The Alto's screen: 606 by 808 dots, one bit each. That is 489,648 bits, about 61 kilobytes.
4. So a graphical screen needs about 32 times the memory of a text screen just to hold the picture, before any program runs.
5. The original Macintosh screen is 512 by 342, one bit per dot, which is 21,888 bytes out of a total of 128 kilobytes of memory.
6. Seventeen per cent of the whole machine was the picture on the screen.

Machine	Screen	Frame buffer
Text terminal	80 x 24 chars	About 2 kB
Xerox Alto, 1973	606 x 808 mono	About 61 kB
Apple Lisa, 1983	720 x 364 mono	About 33 kB
Macintosh, 1984	512 x 342 mono	About 22 kB

■ 14.13.4 PLAIN – what is really happening inside

1. Engelbart and Bill English built the first mouse prototype in 1964, a wooden box with two wheels at right angles. The patent, number 3,541,541, was granted on 17 November 1970.
2. Xerox did sell a graphical machine. The Xerox Star, formally the 8010 Information System, launched in April 1981 with icons, windows, folders and a two-button mouse.
3. It cost \$16,595 per workstation and needed a network of them to be useful. It sold poorly. Xerox invented the future and failed to price it.
4. The Apple visit of December 1979 was not theft. Xerox was given the right to buy 100,000 Apple shares before Apple's public offering, in exchange for showing its work.
5. Apple's engineers then built things PARC had not: the pull-down menu bar fixed at the top, click-and-drag direct manipulation, overlapping windows on a cheap machine, and the trash can.
6. The Lisa failed commercially. It was slow, its own disk format was unreliable, and it cost as much as a house deposit. Estimates of units sold range from about 10,000 to 60,000.
7. The Macintosh succeeded, helped by the "1984" advertisement directed by Ridley Scott, shown once during the Super Bowl on 22 January 1984.
8. Microsoft's Windows 1.0 could not overlap its windows. They were tiled, side by side, partly for technical reasons and partly because of Apple's licensing terms. Overlapping arrived with Windows 2.0 in December 1987.

■ 14.13.5 TECHNICAL – the engineer's version

1. Alto specification: introduced 1 March 1973, designed largely by Charles P. Thacker; 128 kilobytes of memory expandable to 512; a 2.5 megabyte removable disk cartridge; a portrait bitmap display; Ethernet at 2.94 megabits per second, invented at PARC by Robert Metcalfe and David Boggs, whose founding memo is dated 22 May 1973. About 2,000 were built.
2. Smalltalk-72, Smalltalk-76 and Smalltalk-80 came from Alan Kay's Learning Research Group. Smalltalk is where object-oriented programming, the model-view-controller pattern and the modern integrated development environment were worked out together.
3. Bravo, written by Charles Simonyi in 1974 for the Alto, is the first what-you-see-is-what-you-get document editor. Simonyi later joined Microsoft and led Word and Excel.
4. Windows release history, with dates:

Version	Released	Notable change
Windows 1.0	20 Nov 1985	Tiled windows only
Windows 2.0	Dec 1987	Overlapping windows
Windows 3.0	22 May 1990	First big commercial hit
Windows 3.1	Apr 1992	TrueType fonts
Windows 95	24 Aug 1995	Start menu, taskbar

5. Windows 95 sold about one million copies in four days and was supported by an advertising campaign costing hundreds of millions of dollars. It is the moment the graphical interface became the default for ordinary computers.
6. *Apple Computer, Inc. v. Microsoft Corp.*, the look-and-feel case:
 1. Filed in 1988 against Microsoft and Hewlett-Packard.
 2. Apple listed 189 interface elements it said were copied.
 3. Microsoft's defence rested on a 1985 agreement licensing Apple interface elements for Windows 1.0. The district court found 179 of the 189 elements were covered by that licence.
 4. For the rest, the court applied a "virtual identity" standard, because the remaining similarities were unprotectable ideas or the only practical way of expressing them.
 5. The Ninth Circuit affirmed in 1994. The Supreme Court declined to hear the case in 1995.
 6. Xerox sued Apple during the same period and lost, mainly on timing.
 7. All remaining disputes were settled in August 1997, with Microsoft investing \$150 million in non-voting Apple stock.
7. The lasting legal principle: an interface idea, such as overlapping windows or icons for files, is not copyrightable. Its specific artwork and code are.

■ 14.13.6 WORDS - remember these

1. Graphical user interface — pointing at pictures instead of typing — a GUI, using windows, icons, menus and a pointer.
2. Bitmap display — a screen where every dot is separately set — a raster framebuffer with one or more bits per pixel.
3. Mouse — the pointing box — an X-Y position indicator, prototyped by Engelbart and English in 1964, patented in 1970.
4. WYSIWYG — the screen looks like the printout — what you see is what you get, first in Bravo on the Alto in 1974.
5. Desktop metaphor — files as paper on a desk — the interface model of the Xerox Star, 1981, and its successors.
6. Look and feel — the overall feel of an interface — the contested subject of Apple v Microsoft, largely held unprotectable in 1994.

14.14 Networks, the web and mobile: the spine only

■ 14.14.1 PLAIN - in simple words

1. This section is deliberately short. Chapter 30 tells the story of networks and the web in full, and Chapter 31 tells the story of mobile.
2. Here you get only the spine, so the timeline in this chapter is not missing a limb.
3. Networks: in 1969 the United States funded ARPANET, a network joining four university computers so researchers could share machines.

4. The first message was sent on 29 October 1969, from UCLA to Stanford Research Institute, by a student named Charley Kline. He typed L, then O, and the system crashed before the G.
5. The web: in March 1989 Tim Berners-Lee, a British engineer at CERN in Switzerland, wrote a proposal for a linked document system. His boss's note on it read "vague but exciting".
6. Mobile: phones went from car-boot-sized analogue radios in 1983 to pocket computers with a screen in 2007.

■ 14.14.2 PLAIN – a picture in your head

1. A network is a postal system that never asks who you are, only where the letter goes.
2. The web is a library laid on top of that postal system, where every book can contain the address of another book.
3. Mobile is the same postal system with the address being a moving person rather than a fixed building.
4. Each layer is built on the one below and does not know how it works.

Where this comparison breaks: the post office delivers whole letters, while a network chops your message into small numbered pieces that may travel by different routes and arrive out of order. Re-assembly is the receiver's job.

■ 14.14.3 PLAIN – a worked example

1. You can watch this history with one command on your own machine.

```
tracert github.com
 1  192.168.0.1
 2  172.31.0.17
 3  137.97.29.249
 ...
 7  ae66-0.dcl01-96cbe-1b.ntwk.msn.net
 8  be23.rwa02.bom01.ntwk.msn.net
10  be5.ibr02.pnq21.ntwk.msn.net
```

2. Every line is a machine that looked at your packet, decided where next, and passed it on.
3. That "look and forward" behaviour is exactly what the first ARPANET nodes did in 1969, running on Honeywell minicomputers called Interface Message Processors.
4. The addresses beginning 172 are private ranges reserved by RFC 1918. The names ending msn.net are one company's naming convention, not a standard.
5. Fifty-seven years separate the first hop and the last, and the basic idea has not changed.

■ 14.14.4 PLAIN – what is really happening inside

1. The spine of networking, in eight dates.
2. 1969: ARPANET's first four nodes.
3. 1974: Vinton Cerf and Robert Kahn publish the design of TCP, the protocol that lets separate networks join into one internet.
4. 1 January 1983: the whole ARPANET switches to TCP/IP on a single day, still called the flag day.
5. 1983 and 1984: Paul Mockapetris designs the Domain Name System, so people can type names instead of numbers.
6. March 1989: Berners-Lee's proposal at CERN. December 1990: the first web server and browser run.
7. 30 April 1993: CERN places the web software in the public domain, charging nothing, for ever. That decision is why the web is not a product.

8. 1993: the Mosaic browser from NCSA, written by Marc Andreessen and Eric Bina, puts images and text on one page and the public arrives.
9. 1998: Google is founded, and finding things becomes the main problem.

■ 14.14.5 TECHNICAL – the engineer’s version

1. Mobile generations, with the years the first commercial networks opened:

Generation	From	Key change
1G analogue	1979 to 1983	Voice only, no security
2G GSM	1991	Digital, SMS, SIM cards
3G	2001	Packet data at scale
4G LTE	2009	All-IP, no voice circuit
5G	2019	Low latency, dense cells

2. The first GSM call was made in Finland in 1991. The first SMS text message was sent in December 1992 and read “Merry Christmas”.
3. 3G opened commercially with NTT DoCoMo’s FOMA service in Japan in October 2001. The first commercial 4G LTE networks opened in Stockholm and Oslo in December 2009.
4. Two structural facts worth carrying forward. First, the internet has no centre and no owner; it is an agreement between networks to pass traffic. Second, the web is one application running on it, alongside email, video calls and file transfer.
5. Chapter 30 covers packets, routing, TCP, DNS, HTTP and TLS properly. Chapter 31 covers radio, cells, handover and the mobile stack.

■ 14.14.6 WORDS – remember these

1. ARPANET — the 1969 research network — the packet-switched predecessor of the internet, funded by the US Advanced Research Projects Agency.
2. Packet — a small numbered piece of your message — a routed protocol data unit with header and payload.
3. TCP/IP — the agreement that joins networks together — the Transmission Control Protocol and Internet Protocol suite, universal from 1 January 1983.
4. DNS — the phone book turning names into numbers — the Domain Name System, designed by Paul Mockapetris in 1983 and 1984.
5. World Wide Web — linked pages on top of the internet — the hypertext system proposed by Berners-Lee at CERN in March 1989, released freely in 1993.

14.15 The last twenty years

■ 14.15.1 PLAIN – in simple words

1. For thirty years computers got faster in the simplest way: the clock ticked faster every year.
2. Around 2004 that stopped. Pushing the clock higher made chips too hot to cool. Intel cancelled its next high-clock designs in May 2004.
3. The answer was to put several complete processors, called **cores**, on one chip. Mainstream dual-core desktop chips arrived in May 2005.
4. This shifted the burden onto software. A single-threaded program gets no faster on a four-core chip.
5. On 9 January 2007 Apple announced the **iPhone**, which shipped on 29 June

2007. In September 2008 the first Android phone was announced.

6. From 2006, Amazon began renting computing by the hour. Storage, called S3, opened on 14 March 2006, and rentable servers, called EC2, in August 2006. That is **cloud computing**.
7. Around 2011 the world began shipping more smartphones than personal computers, and never went back.
8. Graphics chips, built to draw triangles for games, turned out to be ideal for the arithmetic behind machine learning.
9. Since 2012 that has driven the largest hardware build-out in the industry's history.
10. In 2026 the fastest computer on the public TOP500 list is a Chinese machine called LineShine, at about 2.2 exaflops.

■ 14.15.2 PLAIN – a picture in your head

1. Imagine a restaurant kitchen with one chef who cooks faster every year.
2. Eventually the chef cannot move any faster without catching fire.
3. So you hire four chefs. The kitchen can now serve four tables at once.
4. But one complicated dish still takes one chef the same time. Four chefs cannot chop one onion four times as fast.
5. That is why your computer has eight or sixteen cores and some programs still feel exactly as slow as they did in 2010.

Where this comparison breaks: chefs share one kitchen. Cores share caches and one path to memory, so adding cores can even slow things down through contention. Splitting work also costs time in coordination, which is why Amdahl's law, stated by Gene Amdahl in 1967, says the sequential part of a program sets a hard ceiling on any speed-up.

■ 14.15.3 PLAIN – a worked example

1. Amdahl's law in numbers. Suppose 90 per cent of a program can be split across cores and 10 per cent cannot.
2. Speed-up with N cores is 1 divided by (0.1 plus 0.9 over N).

Cores	Speed-up	Efficiency
1	1.00	100%
2	1.82	91%
4	3.08	77%
16	6.40	40%
1,000	9.91	1%

3. With a tenth of the work stuck in sequence, a thousand cores buy you less than ten times the speed.
4. That table explains the whole software industry since 2005. It is also why machine learning suits GPUs so well: its core operation, multiplying large matrices, is nearly all parallel.

■ 14.15.4 PLAIN – what is really happening inside

1. Why the clock stopped rising: a rule called Dennard scaling used to say that as transistors shrank, power per unit area stayed constant. It broke down around 2005 because leakage current stopped shrinking with size.
2. So power density rose with frequency, and the heat became unremovable with air cooling.
3. Clock speed comparison: a Pentium 4 reached 3.8 gigahertz in 2004. High-end desktop chips in 2026 boost to roughly 5 to 6 gigahertz. That is under twice as fast in twenty-two years.

4. Core counts in the same period went from 1 to 16 or 24 on a desktop, and to well over 100 on server parts.
5. Cloud computing changed who owns machines. Before 2006, needing 500 servers for one week meant buying 500 servers. After 2006 it meant a credit card and ten minutes.
6. Phones changed what a computer looks like. The processor in a modern phone uses the ARM instruction set, designed in Cambridge in 1985 for the Acorn Archimedes, and licensed rather than manufactured by its owner.
7. Apple moved its own laptops and desktops to ARM chips of its own design with the M1 in November 2020, ending nearly fifteen years of Intel Macs.

■ 14.15.5 TECHNICAL – the engineer’s version

1. Multicore timeline: IBM POWER4, the first commercial dual-core general purpose processor, shipped in 2001. Mainstream x86 dual-core parts, the AMD Athlon 64 X2 and the Intel Pentium D, both launched in May 2005.
2. GPU computing timeline: Nvidia released the CUDA software development kit on 15 February 2007 for the G80 generation, starting with the GeForce 8800.
3. The result that changed the field was AlexNet, which won the ImageNet competition in 2012 by a large margin, trained on two consumer graphics cards. The 2017 paper *Attention Is All You Need* introduced the transformer architecture behind current language models.
4. Where things stand in 2026, separated honestly:
 1. **Established fact:** Nvidia’s Blackwell B100, from 2024, carries about 208 billion transistors. Apple’s M3 Ultra, from 2025, carries about 184 billion. TSMC’s 3-nanometre class process is in high-volume production and 2-nanometre class parts are entering it.
 2. **Established fact:** on the June 2026 TOP500 list the leading machine is LineShine at the National Supercomputing Centre in Shenzhen, China, at 2,198 petaflops, ahead of El Capitan at Lawrence Livermore at 1,809 and Frontier at Oak Ridge at 1,353.
 3. **Active research:** whether current model architectures continue to improve with more data and compute, and what the true limits of low precision arithmetic are for training.
 4. **Active research:** quantum computing. Real machines exist and real results are published, but no general commercial workload runs faster on one today.
 5. **Marketing claim:** process node names such as “3 nanometre” no longer describe any physical dimension on the chip. They are commercial labels. Compare vendors by transistor density, not by node name.
5. Process node names are the clearest example in the industry of a **marketing convention** dressed as a specification. Say so when you hear it.
6. Tools that let you observe your own machine’s place in this story:

```
lscpu           # cores, threads, caches, flags
cat /proc/cpuinfo # per-core model and speed
nvidia-smi      # GPU model, memory, utilization
sysctl -n machdep.cpu.brand_string # macOS CPU name
```

■ 14.15.6 WORDS – remember these

1. Core — a complete processor inside the chip — an independent execution unit with its own registers and level-1 caches.
2. Dennard scaling — smaller transistors used to mean cooler chips — the constant power-density rule that failed around 2005.
3. Amdahl’s law — the slow part sets the limit — the speed-up bound imposed by the non-parallel fraction of a program, stated in 1967.

4. Cloud computing — renting computers by the hour — on-demand elastic provisioning of remote compute and storage, from AWS S3 in March 2006.
5. GPU — the graphics chip now used for mathematics — a throughput-oriented many-core processor, made general-purpose by CUDA in 2007.
6. Exaflop — a billion billion sums a second — 10 to the power 18 floating point operations per second.

14.16 One master timeline, 1801 to 2026

■ 14.16.1 PLAIN – in simple words

1. A timeline is not a list of gadgets. It is a list of moments when something became possible that had not been possible the day before.
2. Read it once for the shape, not for the detail.
3. You will see three kinds of entry: an idea written down, a machine that worked, and a product that sold.
4. All three are needed. An idea nobody builds is philosophy. A machine nobody buys is a museum piece.
5. You will also see long gaps. Between Babbage in 1837 and the Z3 in 1941 there are 104 years in which almost nothing happened in general computing.

■ 14.16.2 PLAIN – a picture in your head

1. Think of a relay race where the baton is dropped for a century and picked up by somebody who never met the first runner.
2. Babbage's designs were largely forgotten. Turing did not build on them. Zuse had not heard of them.
3. The same idea was found independently three or four times.
4. That is normal in this history. Ideas arrive when the parts to build them become cheap, not when someone first thinks of them.

Where this comparison breaks: it was not a single race. Several teams ran in parallel in different countries, often in secret, and the winner was often decided by who was allowed to publish.

■ 14.16.3 PLAIN – a worked example

1. Take one row and unpack it: 1948, Manchester Baby runs.
2. The idea came from Turing in 1936 and the design from the EDVAC report in 1945, so the idea is 12 years older than the machine.
3. The enabling part was the Williams tube, a memory made from a war-surplus radar display.
4. So the row hides three things: a mathematician's paper, a wartime component, and an engineering team.
5. Every row in the table below hides a story of that shape.

■ 14.16.4 PLAIN – what is really happening inside

1. Four forces recur throughout this timeline.
2. **Cost per operation** falls, roughly by a factor of ten every five to seven years across the whole period.
3. **Physical size** falls, from a room, to a cabinet, to a board, to a chip, to part of a chip.
4. **Who is allowed to use it** widens: governments, then corporations, then departments, then individuals, then everyone with a pocket.

5. **What it is used for** widens from arithmetic, to records, to text, to pictures, to communication, to everything.
6. If you can name those four trends you can place any unfamiliar machine in this story within a decade of its real date.

■ 14.16.5 TECHNICAL – the engineer’s version

Year	Event	Why it mattered
1801	Jacquard shows loom in Paris	Cards begin to steer machines
1804	Jacquard head patented	Program separate from machine
1822	Babbage’s difference engine	Machine-made number tables
1837	Analytical Engine designed	Mill, store, cards, printer
1843	Lovelace publishes Note G	First published algorithm
1854	Boole’s Laws of Thought	Logic becomes algebra
1889	Hollerith patent 395,782	Electric card counting
1890	US census tabulated	First mass data processing
1911	CTR formed by merger	The firm that becomes IBM
1924	CTR renamed IBM	Sixty years of dominance
1931	Gödel’s incompleteness	Limits of proof shown
1936	Turing and Church papers	Limits of computing shown
1937	Shannon’s relay thesis	Boolean algebra meets wires
1941	Zuse’s Z3 runs	First program-driven machine
1942	Atanasoff–Berry Computer	Electronic binary arithmetic
1943	Colossus Mark 1 built	First big electronic machine
1944	Harvard Mark I presented	US sequence-controlled machine
1945	First Draft on EDVAC	Stored-program design set out
1946	ENIAC shown to press	Electronic speed made public
1947	Transistor at Bell Labs	End of the vacuum tube
1948	Manchester Baby runs	First stored program executed
1949	EDSAC service starts	Subroutines and a real service
1951	Ferranti Mark 1 delivered	First commercial computer
1951	UNIVAC I accepted	US commercial computing
1952	UNIVAC predicts election	Computer enters common speech
1953	IBM 701 shipped	IBM enters electronic computing
1956	IBM 350 disk drive	Random-access storage arrives
1957	Fortran released	Programming above the machine
1957	Eight leave Shockley	Fairchild and Silicon Valley
1958	Kilby’s integrated circuit	Parts without wires
1959	Noyce’s planar IC	Chips can be mass produced
1959	DEC PDP-1 delivered	Interactive computing begins
1964	IBM System/360 announced	One compatible product family
1964	BASIC at Dartmouth	Beginners can program
1965	Moore’s law published	The industry gets a roadmap
1965	DEC PDP-8 at \$18,500	Minicomputer era opens
1968	Intel founded	The chip giant begins
1968	Engelbart’s demo in SF	Mouse and windows shown
1969	ARPANET first message	Networking begins
1969	Unix begun at Bell Labs	Portable operating system
1969	AMD founded	A second source for chips
1970	DEC PDP-11 announced	Unix and C grow up on it

Year	Event	Why it mattered
1971	Intel 4004 announced	A processor on one chip
1971	Silicon Valley named	The region gets its identity
1972	C language at Bell Labs	Portable systems software
1973	Xerox Alto working	Graphical personal computer
1973	Ethernet memo at PARC	Local networks
1974	Intel 8080 released	The first hobby computer CPU
1975	Altair 8800 on sale	Personal computing starts
1975	Microsoft founded	Software becomes an industry
1975	MOS 6502 at \$25	Cheap CPUs for home machines
1976	Apple I on sale	Apple begins
1976	Zilog Z80 released	The dominant 8-bit CPU
1977	Apple II, PET, TRS-80	Ready-to-use home computers
1978	Intel 8086 released	The x86 line starts
1979	VisiCalc released	The first killer application
1981	IBM PC 5150 announced	The business standard
1981	Xerox Star launched	Desktop metaphor for sale
1982	Compaq Portable announced	Clean-room clones arrive
1983	ARPANET moves to TCP/IP	One internet
1983	Apple Lisa launched	GUI reaches the market
1984	Macintosh launched	GUI becomes affordable
1985	Windows 1.0 released	Microsoft enters graphics
1985	ARM design starts at Acorn	The CPU in your phone
1989	Web proposed at CERN	Linked documents
1991	Linux kernel announced	Free operating system
1993	Mosaic browser released	The public web
1993	CERN frees web software	No owner, no licence fee
1995	Windows 95 released	GUI becomes the default
1998	Google founded	Search organizes the web
2001	IBM POWER4 dual core	Multicore begins
2003	AMD Opteron ships x86-64	64-bit on commodity chips
2005	Dual-core desktop chips	Clock speed race ends
2006	AWS S3 and EC2 launch	Computing rented by the hour
2007	iPhone released	The pocket computer
2007	CUDA SDK released	GPUs become general
2008	First Android phone	Smartphones become universal
2011	Smartphones outsell PCs	The centre of gravity moves
2012	AlexNet wins ImageNet	GPUs become learning engines
2017	Transformer paper	Modern language models
2020	Apple M1 ships	ARM reaches the desktop
2022	ChatGPT released	Language models reach public
2024	Nvidia Blackwell announced	208 billion transistors
2025	Apple M3 Ultra ships	184 billion transistors
2026	LineShine tops TOP500	2.2 exaflops in Shenzhen

1. A caution on any timeline: dates in computing are slippery because announcement, first working unit, first delivery and general availability can be years apart.
2. Where they differ in this chapter, the body text says which one is meant.
3. Where a claim is contested, such as who first built a computer, the chapter says so rather than picking a winner.

■ 14.16.6 WORDS – remember these

1. Announcement date — when it was told to the world — the marketing date, often the earliest and least meaningful.
2. First working unit — when it actually ran — the engineering milestone used for priority claims.
3. General availability — when anyone could buy one — the commercial date, the one that changes an industry.
4. Priority claim — the argument about who was first — a disputed assertion that depends entirely on the definition chosen.

14.98 Common wrong ideas

1. Wrong: Babbage built the Analytical Engine. Right: he designed it from 1837 and built only fragments. Neither of his engines was completed in his lifetime; he died in 1871. The Science Museum in London built Difference Engine No. 2 from his drawings, finishing the calculating section in 1991.
2. Wrong: the first computer bug was a moth. Right: the moth found in a Harvard Mark II relay on 9 September 1947 was taped into a logbook with the note “First actual case of bug being found”. The joke only works because engineers already said “bug” for a fault. Thomas Edison used the word that way in 1878.
3. Wrong: Grace Hopper coined the word bug. Right: she popularized the moth story. She did lead the team that built the first compiler, the A-0 system in 1952, and shaped COBOL from 1959, which is achievement enough.
4. Wrong: Bill Gates wrote MS-DOS. Right: Tim Paterson at Seattle Computer Products wrote 86-DOS in 1980. Microsoft licensed it for \$25,000 in December 1980 and bought it outright for \$50,000 in July 1981. Gates and Allen did write Altair BASIC themselves in 1975.
5. Wrong: Apple invented the graphical interface. Right: Douglas Engelbart demonstrated the core ideas on 9 December 1968 and Xerox PARC built them into the Alto in 1973 and sold the Star in 1981. Apple licensed, refined and priced it for ordinary buyers with the Lisa in 1983 and the Macintosh in 1984.
6. Wrong: Apple stole the interface from Xerox. Right: the December 1979 visit was arranged, and Xerox received the right to buy Apple shares in exchange. Xerox later sued Apple over the interface and lost, largely on timing.
7. Wrong: Al Gore said he invented the internet. Right: on 9 March 1999, on CNN, he said “I took the initiative in creating the Internet”, meaning legislation. Vinton Cerf and Robert Kahn, who designed TCP/IP, publicly defended his record. His 1991 High Performance Computing Act helped fund the Mosaic browser.
8. Wrong: ENIAC was the first computer. Right: that depends entirely on the definition. The Z3 of 1941 was program-driven, the Atanasoff-Berry machine of 1942 was electronic, Colossus of 1943 was electronic and digital, and the Manchester Baby of 1948 was the first to run a stored program. ENIAC was the first general-purpose electronic machine put to broad use.
9. Wrong: computers have always used binary. Right: ENIAC, the Harvard Mark I, the IBM 650 and many other early machines were decimal. Binary won because it halves the components needed per digit and matches switching circuits.
10. Wrong: Silicon Valley is named after a company. Right: the journalist Don Hoefler used the phrase in Electronic News on 11 January 1971. Silicon is the element the chips are made from, and the valley is the Santa Clara Valley in California.

14.99 Chapter summary in 20 lines

1. Counting aids came first: tally sticks, the abacus, and the geared Antikythera mechanism of roughly 100 BCE, found in a wreck in 1901.
2. Napier's bones of 1617, the slide rule of about 1622 and Leibniz's 1703 paper on binary set the mathematical stage.
3. Schickard in 1623, Pascal in 1642 and Leibniz in 1673 built calculators that worked but could not be reprogrammed.
4. Jacquard's loom head, patented in 1804, separated the machine from its instructions. That is the founding idea of the whole field.
5. Babbage designed the Difference Engine from 1822 and the Analytical Engine from 1837, with mill, store, card reader and printer. Neither was finished.
6. Ada Lovelace's 1843 notes contain the first published algorithm and the first clear statement that a computer could handle symbols, not just quantities.
7. Hollerith's punched card machines processed the 1890 United States census. His company merged in 1911 and was renamed IBM in 1924.
8. Gödel showed in 1931 that proof has limits. Church and Turing showed in 1936 that computing has limits too, and Turing's model described every later machine.
9. Turing complete means able to simulate a universal Turing machine, and the bar is so low that card games and spreadsheet formulas clear it by accident.
10. The war produced the Bombe in 1940, Zuse's Z3 in 1941, the Atanasoff-Berry machine in 1942, Colossus in 1943 and 1944, and the Harvard Mark I in 1944.
11. ENIAC, shown in 1946, was electronic and general-purpose but was programmed by rewiring, work done by six women who went uncredited for forty years.
12. The 1945 First Draft of a Report on the EDVAC set out the stored-program design that everything since has followed, under a disputed name.
13. The Manchester Baby ran the first stored program on 21 June 1948, EDSAC gave the first service in 1949, and the Ferranti Mark 1 was sold in 1951.
14. UNIVAC I predicted the 1952 election correctly and was overruled live on air, which is how the public learned what a computer was.
15. The transistor of 1947 and IBM's System/360 of 1964 turned computing into an industry with compatible product families and portable software.
16. Kilby in 1958 and Noyce in 1959 put whole circuits on one chip, and Hoerni's planar process made them manufacturable.
17. The Intel 4004 of 1971 put a processor on one chip; the \$25 MOS 6502 of 1975 made home computers affordable.
18. The Altair of 1975, Apple II of 1977, VisiCalc of 1979 and the IBM PC of 1981 with its published design created the personal computer industry.
19. Engelbart in 1968 and Xerox PARC in 1973 invented the graphical interface; Apple sold it in 1984 and Windows 95 made it universal in 1995.
20. Since 2005 progress has come from many cores, rented cloud machines, phones and graphics chips repurposed for machine learning, with the June 2026 TOP500 led by LineShine at about 2.2 exaflops.



PART D

Software

From punched cards to compilers, languages, operating systems, the terminal, and what an executable file really is.

The Birth of Software — Punch Cards to the First Compiler

15.0 What this chapter gives you

1. You will be able to say exactly what a program is at the lowest level, and show a real one as raw numbers.
2. You will be able to take a handful of bytes and work out, by hand, what the machine will do with them.
3. You will be able to describe how ENIAC was programmed in 1946 with cables and switches, and how long a change took.
4. You will be able to explain a punched card completely: its size, its 80 columns, its 12 rows, and how a letter was stored as holes.
5. You will be able to say what an assembler is and why a label beats a hand-counted address.
6. You will be able to answer “who wrote the first compiler” honestly, and say why the honest answer has several parts.
7. You will be able to say what Grace Hopper’s A-0 actually did in 1952, and what it did not do.
8. You will be able to explain why FORTRAN had to be fast or it would have failed, and why COBOL still runs banks in 2026.
9. You will be able to tell the moth story correctly, including the part most retellings get wrong.
10. You will be able to trace the line from a human carrying card decks to a modern operating system, and from Multics to UNIX to C.

15.1 What a program is at the bottom

■ 15.1.1 PLAIN – in simple words

1. A computer’s memory is a very long row of numbered boxes.
2. Each box holds one small number, from 0 to 255. That is one **byte**.
3. A program is a run of boxes that the processor has agreed to read as orders instead of as data.
4. The processor keeps a marker pointing at one box. It reads the number there, does what that number means, and moves the marker on.
5. That is the whole show. Read a number, do the thing, move on. Forever.
6. The same number can be an order in one place and data in another. Only position and agreement decide which.
7. Writing a program in 1948 meant choosing every one of those numbers yourself, by hand, and getting every one right.

■ 15.1.2 PLAIN – a picture in your head

1. Think of a long street of numbered houses, and one postman.
2. Each house has one card in the letterbox with a number on it.
3. He carries a notebook with a few slots. Those are the processor's **registers**, its scratch space.
4. He never wonders whether a card is an order or a note. Where he is standing decides that. Walk him into a shopping list and he will obey the shopping list.

Where this comparison breaks: a real processor does not work in one tidy line. Modern chips fetch many instructions ahead, guess which way branches go, and run several at once, then hide the mess so it looks sequential.

■ 15.1.3 PLAIN – a worked example

1. Here is a real program for the MOS 6502, the chip in the Apple II, the Commodore 64 and the Nintendo Entertainment System.
2. In memory it is nothing but these seven numbers.

Address	Bytes	What the processor does
0600	A9 05	put 5 into the accumulator
0602	69 03	add 3 to the accumulator
0604	8D 00 02	write accumulator into address 0200
0607	00	stop

3. At 0600 the byte is A9, meaning “load the accumulator with the next byte”. The next byte is 05, so the accumulator becomes 5.
4. At 0602 the byte is 69, meaning “add the next byte”. That is 03, so the accumulator becomes 8.
5. At 0604 the byte is 8D, meaning “store the accumulator at the two-byte address that follows”. Those bytes are 00 then 02.
6. The 6502 stores addresses low byte first, so 00 02 means address 0200, not
 2. That ordering is called **little-endian**.
7. Address 0200 now holds 8. At 0607 the byte 00 is BRK, and the machine stops.
8. That is a complete program. Seven numbers. Nothing else exists.

■ 15.1.4 PLAIN – what is really happening inside

1. Inside the processor is a register called the **program counter**, holding the address of the next instruction.
2. That address goes onto the address wires, and memory puts the byte at that address onto the data wires.
3. The byte lands in the instruction register, and a block of logic called the **decoder** turns on a specific set of control wires.
4. Meanwhile the program counter has already advanced to the next byte.
5. A9 does not “mean” load in any deep sense. The decoder was built so the bit pattern 1010 1001 turns on the load-immediate wires. That is all.
6. Different chip, different wiring, different meanings for the same numbers. That is why code for one processor family is noise to another.

■ 15.1.5 TECHNICAL – the engineer's version

1. The 6502 is an 8-bit processor with a 16-bit address bus, so it addresses 65,536 bytes. MOS Technology introduced it in September 1975 at 25 United States dollars, against roughly 179 dollars for competing parts.

2. User-visible registers: A (accumulator), X and Y (index), SP (stack pointer, fixed to page 01), PC (16 bits) and P (status flags).
3. A9 is LDA immediate, 2 bytes, 2 cycles. 69 is ADC immediate, 2 bytes, 2 cycles. 8D is STA absolute, 3 bytes, 4 cycles. 00 is BRK, 1 byte, 7 cycles.
4. ADC adds the carry flag as well as the operand. That is not optional. If carry is set from earlier work, the example above yields 9, not 8. Correct 6502 code clears carry first with CLC, opcode 18.
5. Little-endian ordering for 16-bit operands is an architectural rule of the 6502, not a convention you may vary.

Item	6502 (1975)	x86-64 (today)
Opcode length	1 byte	1 to 4 bytes
Instruction length	1 to 3 bytes	1 to 15 bytes
Address bus	16 bits	48 bits used
Documented opcodes	151	over 1,000

6. To see this today: `objdump -d` on Linux, `otool -tv` on macOS, `xxd` or `hexdump -C` for raw bytes, and `gdb` with `x/8xb` to dump memory.

The honest version: “the CPU reads a byte and does it” is a fair description of a 1975 chip and a poor one of a 2026 chip. A modern x86 core decodes variable-length instructions into fixed micro-operations, renames registers, reorders execution, and retires results in order to keep up the appearance of sequence. The programmer’s model is still one at a time; the hardware stopped working that way around 1995.

■ 15.1.6 WORDS – remember these

1. Byte — a number from 0 to 255 — 8 bits, the standard addressable unit.
2. Machine code — the raw numbers a chip obeys — the binary encoding of an instruction set architecture.
3. Opcode — the number saying which operation — the operation field of an encoded instruction.
4. Program counter — the marker for the next order — a register holding the address of the next instruction to fetch.
5. Register — the processor’s scratch slots — small fast storage inside the core, named in the instruction encoding.
6. Little-endian — low part of a number stored first — least significant byte at the lowest address.

15.2 Before any language existed: programming by rewiring

■ 15.2.1 PLAIN – in simple words

1. In 1945 there was no such thing as a programming language, and barely such a thing as a program.
2. ENIAC, finished in late 1945 at the University of Pennsylvania, did not read instructions from memory. It had no memory for instructions.
3. To make it do a calculation you physically rebuilt the connections between its parts.
4. You plugged heavy cables between panels, exactly like an old telephone switchboard, and set banks of rotary switches by hand.
5. The program existed as the arrangement of cables and the positions of switches. Nothing was stored anywhere.
6. A new problem took weeks of planning on paper, days of physical work on the machine, and days more of finding what was wired wrong.

7. The people doing this were not called programmers. They were called operators, because until then “computer” meant a person who computed.

■ 15.2.2 PLAIN – a picture in your head

1. Think of a model railway with no track laid down.
2. To run a train from the station to the mine you lay every piece of track, set every point and connect every signal wire yourself.
3. Now you want the train to go to the quarry instead. There is no button for that. You lift the track and lay it again.
4. The layout is the program. A stored-program computer is the opposite: the track is fixed forever, and you hand the driver a route on a slip of paper.

Where this comparison breaks: ENIAC’s units could repeat and could branch on a condition, so it was more capable than fixed track. And in 1948 it was converted so its function tables held a stored program, cutting setup from days to hours at the cost of running about six times slower.

■ 15.2.3 PLAIN – a worked example

1. ENIAC had 20 accumulators, each holding a 10-digit decimal number.
2. To compute “add accumulator 3 and accumulator 5 into accumulator 7” you did the following.
3. Run a program cable from a digit output socket on accumulator 3 to a digit input socket on accumulator 7.
4. Run a program pulse cable so the right control pulse triggers both transmissions at the same moment in time.
5. Do that for every step of the calculation. There were hundreds of steps.
6. Then test it, and when the answer is wrong, work out whether the fault is your wiring, a bad switch, or one of 18,000 vacuum tubes that has failed since breakfast.

■ 15.2.4 PLAIN – what is really happening inside

1. ENIAC was a set of separate calculating units that could be joined in any pattern.
2. Two kinds of cable ran between them. Digit trunks carried the numbers. Program trunks carried timing pulses meaning “now”.
3. So a program was a graph: which unit sends numbers to which, and in what order the “now” pulses fire. That is closer to circuit design than to writing instructions.
4. The three function tables were racks of switches, 1,200 ten-way switches per table, set by hand to hold constants and lookup values.
5. Because the units ran in parallel, ENIAC could do several things at once. It was a parallel machine before it was a programmable one.
6. In 1948 it was rewired once, permanently, so the function tables held a list of coded orders that the machine fetched and obeyed.
7. That change turned a rewiring job into a switch-setting job, and it is the moment ENIAC became something we would recognize as programmable.

■ 15.2.5 TECHNICAL – the engineer’s version

1. ENIAC: Electronic Numerical Integrator and Computer, built at the Moore School of Electrical Engineering, University of Pennsylvania, for the US Army Ballistic Research Laboratory.
2. First serious work: December 1945, Los Alamos thermonuclear calculations. Publicly announced and dedicated 14 to 15 February 1946.

3. About 18,000 vacuum tubes, 7,200 crystal diodes, 1,500 relays, over 30 short tons, roughly 30 by 3 metres of floor space, 150 kilowatts.
4. Clock rate 100 kHz, giving a 200-microsecond addition cycle: around 5,000 additions per second.
5. The six principal programmers were Kathleen McNulty, Jean Jennings, Betty Snyder, Marlyn Wescoff, Frances Bilas and Ruth Lichterman. They were hired as human computers and taught themselves the machine from wiring diagrams.
6. The 1948 converter-code modification, with input from John von Neumann and implemented with Richard Clippinger and Adele Goldstine, gave ENIAC a 60-order instruction set read from function table switches.

Task	ENIAC 1946	ENIAC after 1948
Set up a new program	days	hours
Fix a coding mistake	hours	minutes
Additions per second	about 5,000	about 800

The honest version: the popular line “ENIAC took two weeks to reprogram” compresses several different things. Planning could take weeks, physical setup days, and debugging days more. The 1948 conversion is the line that matters, and after it ENIAC was slower but far more usable.

■ 15.2.6 WORDS – remember these

1. Plugboard — a panel you wire by hand — a patch panel defining data and control routing.
2. Accumulator — a box holding a running total — a register that is both operand source and result destination.
3. Function table — a rack of setting switches — read-only storage for constants, later used as instruction memory.
4. Stored program — orders kept in memory like data — the von Neumann model, with instructions and data in one addressable store.
5. Setup time — the wait before the machine can start — non-productive time between jobs, the cost batch systems later attacked.

15.3 Programming in raw numbers: switches, tape and cards

■ 15.3.1 PLAIN – in simple words

1. Once machines stored their programs in memory, the question became how to get the numbers in there.
2. The first answer was the crudest possible: a row of switches on the front of the machine, one switch per bit.
3. The second answer was **paper tape**: a ribbon of paper with rows of holes punched across it. A hole is a 1, no hole is a 0.
4. The third answer, which ruled computing for forty years, was the **punched card**: a stiff paper rectangle holding one line of up to 80 characters.
5. A program was a stack of cards, called a **deck**, held with a rubber band.
6. You gave your deck to an operator. Hours or days later you got a printout back. One mistake meant fixing one card and queueing again.

■ 15.3.2 PLAIN – a picture in your head

1. Imagine writing an essay where each line goes on its own index card, and you may not see the essay until a stranger reads all the cards aloud.
2. The cards must be in perfect order, there are 2,000 of them, and they are not fastened together.
3. So you draw a diagonal ink line across the top edge of the whole stack. If a card moves, the line has a jog in it.
4. You punch a fresh card 412, slide it in, and go back to the shelf. One full day for one typing mistake.

Where this comparison breaks: index cards can be read by anyone. A punched card carried its content as holes, and the characters printed along the top edge were an optional courtesy from the keypunch. Without that printing, a card was unreadable by a human.

■ 15.3.3 PLAIN – a worked example

1. Here is exactly how characters were stored as holes. The coding is called **Hollerith code**, after Herman Hollerith, who patented punched-card tabulating in 1889 for the 1890 United States census.
2. A card has 12 rows. From the top edge down they are row 12, row 11, row 0, then rows 1 to 9.
3. Rows 12, 11 and 0 are the **zone** rows. Rows 0 to 9 are the **digit** rows. Row 0 does double duty.
4. A digit is a single punch in its own row. A letter is two punches: one zone plus one digit.
5. A to I are zone 12 with digits 1 to 9. J to R are zone 11 with digits 1 to 9. S to Z are zone 0 with digits 2 to 9.
6. Notice S starts at digit 2, not 1. There are 26 letters and 27 slots, so the last group is short by one. That quirk is real.

row		'A'	'J'	'S'	'5'
12	->	X	.	.	.
11	->	.	X	.	.
0	->	.	.	X	.
1	->	X	X	.	.
2	->	.	.	X	.
3	->	.	.	.	.
4	->	.	.	.	.
5	->	.	.	.	X
6	->	.	.	.	.
7	->	.	.	.	.
8	->	.	.	.	.
9	->	.	.	.	.

7. So A is holes in rows 12 and 1 of one column, S is holes in rows 0 and 2, and the digit 5 is one hole in row 5.
8. Eighty columns gives eighty characters per card, and that is where the traditional 80-character terminal width comes from.

■ 15.3.4 PLAIN – what is really happening inside

1. A keypunch is a typewriter that makes holes instead of ink. It also prints the characters along the top edge, which is called **interpreting**.
2. A card reader pulls cards through one at a time and shines light or presses metal brushes at each of the 960 hole positions.
3. Where there is a hole, light passes or a brush completes a circuit. That is one bit.
4. So a card is not read as characters. It is read as 12 bits per column, and software decides what those 12 bits mean.

5. Your deck joined a queue with everyone else's, and the machine ran jobs one after another with no human in between. That is **batch processing**.
6. Turnaround, from handing in your deck to collecting your printout, was a few hours in a well-run university and overnight or worse elsewhere.
7. Because turnaround was slow, programmers did something we have nearly lost. They checked their code line by line on paper first, called **desk checking**.

```

write code on a coding sheet          (30 minutes)
  |
  v
keypunch each line onto a card        (1 hour)
  |
  v
hand the deck in at the window        (join the queue)
  |
  v
operator loads it, machine runs it    (2 seconds of CPU)
  |
  v
printout goes into a pigeonhole
  |
  v
you collect it next morning           (turnaround: 20 hours)
  |
  v
one missing comma -> do it all again

```

■ 15.3.5 TECHNICAL – the engineer's version

1. The IBM 80-column card, introduced in 1928, measures 7 and 3/8 by 3 and 1/4 inches, that is 187 by 83 millimetres, on stock about 0.007 inches or 180 micrometres thick. Roughly 143 cards stack to one inch.
2. Rectangular holes replaced earlier round holes precisely so columns could be packed tighter, doubling capacity from 45 to 80 columns.
3. Columns 73 to 80 were reserved by convention for a card sequence number. That is why FORTRAN, by rule, ignores columns 73 to 80 of every line.
4. A dropped deck was recoverable if and only if you had punched those sequence numbers, by running it through a card sorter such as the IBM 82, 83 or 84.

Device	Role	Speed
IBM 026 keypunch	punch cards by hand	typing speed
IBM 711 (for 704)	card reader	250 cards/min
IBM 1402	card reader/punch	800 read, 250 punch
IBM 2540	card reader/punch	1,000 read, 300 punch

5. Front-panel switch entry survived into the microcomputer era. The Altair 8800 of January 1975 shipped with no keyboard: you entered the bootstrap loader on toggle switches, one byte at a time.
6. Job control was itself punched on cards. IBM's Job Control Language, JCL, survives, and its statements still begin with two slashes in columns 1 and 2 because that is where a card reader saw them.

The honest version: “turnaround was a day” is a rough average that varied enormously. A physics group with its own machine at 3 a.m. might get five runs an hour; a student at a busy service bureau might get one run a day, and in bad commercial cases two or three days.

■ 15.3.6 WORDS - remember these

1. Punched card — stiff card storing data as holes — 80-column, 12-row IBM card using Hollerith encoding.
2. Deck — a stack of cards making one program — an ordered card file, sequence-numbered in columns 73 to 80.
3. Keypunch — the machine that punches cards — a card punch with a keyboard, for example the IBM 026 or 029.
4. Batch processing — jobs run one after another with no user present — non-interactive execution under a resident monitor.
5. Turnaround — how long you wait for your answer — elapsed time from job submission to output delivery.
6. Desk checking — reading your own code before running it — manual static verification, the ancestor of code review.

15.4 The first step up: mnemonics and assembly language

■ 15.4.1 PLAIN - in simple words

1. Writing programs as numbers works, and it is unbearable.
2. You must remember that 169 means load and 141 means store, for hundreds of operations, with no help.
3. The fix is small and it changed everything: let people write short words instead of numbers, and let a program do the translation.
4. Instead of A9 you write LDA. Instead of 8D you write STA. These short words are **mnemonics**, which just means memory aids.
5. Instead of “jump back 10 bytes” you write “jump to loop”, and put the word loop beside the line you meant. That word is a **label**.
6. The program that turns this text into numbers is an **assembler**.
7. An assembler is not clever. Mostly it is one line in, one instruction out. But it removes the two things humans are worst at: remembering arbitrary numbers, and counting.
8. This is the first time in history that a program’s job was to write another program.

■ 15.4.2 PLAIN - a picture in your head

1. Imagine giving directions using only distances: go 412 metres, turn, go 78 metres, turn, go 1,050 metres.
2. Now someone adds a roundabout early in the route. Every distance after it is wrong and must be recomputed.
3. Labels are street names: go to Mill Street, turn towards the bridge. Add a roundabout and the directions still work, because names do not shift when things move.
4. The assembler is the local who converts street names back into exact distances, correctly, in a second.

Where this comparison breaks: an assembler does not choose a route. It has no freedom at all. Every line you write becomes the instruction you asked for. A compiler, later in this chapter, does choose.

■ 15.4.3 PLAIN – a worked example

- Here is a real 6502 program that adds the numbers 1 to 5. On the left is what the assembler produces; on the right is what you write.

Addr	Bytes	Label	Assembly	Comment
0600	A9 00		LDA #\$00	running total = 0
0602	A2 01		LDX #\$01	counter = 1
0604	18	loop:	CLC	clear the carry flag
0605	86 10		STX \$10	put counter in memory \$10
0607	65 10		ADC \$10	total = total + counter
0609	E8		INX	counter = counter + 1
060A	E0 06		CPX #\$06	is counter now 6?
060C	D0 F6		BNE loop	if not, go round again
060E	8D 00 02		STA \$0200	save the total
0611	00		BRK	stop

- Trace it. Total starts at 0, counter at 1.
- Pass 1: total = 0 + 1 = 1, counter becomes 2.
- Pass 3: total = 3 + 3 = 6, counter becomes 4.
- Pass 5: total = 10 + 5 = 15, counter becomes 6, the compare matches, and the branch is not taken. Address 0200 ends up holding 15.
- Now look at the byte F6 on the BNE line. That is minus 10 in two's complement: jump back 10 bytes from the instruction after the branch.
- Here is the whole point. Insert one extra instruction anywhere between loop: and the branch, and F6 becomes wrong.
- With a label you insert the line and reassemble. The assembler recounts, and you never see the number F6 at all.

■ 15.4.4 PLAIN – what is really happening inside

- An assembler reads your text twice, which is why they are called **two-pass assemblers**.
- On the first pass it walks the lines, keeps a running address counter, and writes down where every label lands. That list is the **symbol table**.
- It has to walk first because of forward references: you can jump to a label that appears later, and on first sight its address is unknown.
- On the second pass it converts each line to bytes, looking up every label in the symbol table.
- For a relative branch it does one more step: subtract the address of the next instruction from the target, and check the answer fits in a signed byte.
- And they handle **macros**: define a named block of lines once, then write its name wherever you want the block expanded.

■ 15.4.5 TECHNICAL – the engineer's version

- The earliest known symbolic coding scheme is in Kathleen Booth and Andrew Donald Booth's 1947 work "Coding for A.R.C." at Birkbeck College, London. Kathleen Booth is widely credited with inventing assembly language.
- EDSAC at Cambridge first ran a program on 6 May 1949. Its **Initial Orders**, written by David Wheeler, were a bootstrap routine held permanently in uniselectors switches.
- Those Initial Orders read instructions as a single letter mnemonic, a decimal address and a length code, and turned them into binary on load. The IEEE Computer Society credits Wheeler as creator of the first assembler.

4. Wheeler also devised the **Wheeler Jump**: place the return address in the accumulator, jump to the subroutine, and let the subroutine plant that address into its own final jump. That is the first practical closed subroutine call.
5. EDSAC's subroutine library reached 87 routines by 1951, covering floating-point arithmetic, trigonometry, differential equations and matrix work. That is the first software library.
6. Maurice Wilkes, David Wheeler and Stanley Gill published "The Preparation of Programs for an Electronic Digital Computer" in 1951, the first programming textbook. It introduced the word "assemble" for joining separately written sections into one program.
7. SOAP, the Symbolic Optimal Assembly Program, written by Stan Poley for the IBM 650 in 1955, placed instructions around the rotating drum so the next one arrived under the read head just as it was needed. That is optimization by an assembler.

Year	Milestone	Who and where
1947	Coding for A.R.C.	K. and A. Booth, Birkbeck
1949	EDSAC Initial Orders	D. Wheeler, Cambridge
1951	First programming book	Wilkes, Wheeler, Gill
1955	SOAP for IBM 650	Stan Poley, IBM

8. Modern equivalents to observe: `as` (GNU assembler), `nasm`, `objdump -d` to disassemble, and `gcc -S` to see the assembly a compiler emits.

The honest version: "assembly is one-to-one with machine code" is nearly true and not exactly true. Pseudo-instructions, macros and assembler-chosen branch widths mean one written line can become several instructions, or none. On MIPS and RISC-V this is routine and documented.

■ 15.4.6 WORDS - remember these

1. Mnemonic — a short word for an operation — the textual name of an opcode.
2. Label — a name for a place in the program — a symbol bound to an address by the assembler.
3. Assembler — the program turning mnemonics into numbers — a translator with a largely one-to-one instruction mapping.
4. Symbol table — the list of names and addresses — the assembler's map from identifiers to values, built in pass one.
5. Subroutine — code you can call from many places — a closed routine entered with a return linkage.
6. Macro — a named block that gets pasted in — textual expansion performed before assembly.

15.5 Why a higher language was needed

■ 15.5.1 PLAIN - in simple words

1. Assembly made programming possible for ordinary mortals. It did not make it quick, and it did not make it safe.
2. Problem one: assembly is welded to one machine. Code for an IBM 704 is worthless on a UNIVAC I. Buy a new computer, rewrite everything.
3. Problem two: it is slow to write. One line is one machine operation, so a page of algebra becomes hundreds of lines.
4. Problem three: it hides your intent. Reading it tells you what the machine does, never what you meant.

5. So people asked the obvious question. Could you write the formula the way a mathematician writes it, and have a program produce the assembly?
6. Many experienced programmers said no, and they were not being silly. Machines were tiny, machine time cost more than people, and a program that wasted 30 per cent of the machine was unacceptable.
7. So the real question was never “can a machine translate”. It was “can a machine translate well enough that we do not lose the machine”.

■ 15.5.2 PLAIN - a picture in your head

1. Imagine a workshop where every screw is filed by hand by a master, and the masters are proud of it.
2. Someone brings in a screw-cutting machine. The masters are not afraid of losing their jobs. They are afraid the screws will be worse.
3. If the machine’s screws are 30 per cent weaker, the objection is correct and the machine should be rejected.
4. If they are as good and made in a tenth of the time, the argument ends overnight and nobody files screws again. That is exactly what happened with compilers between 1952 and 1957.

Where this comparison breaks: code quality has many dimensions. A compiler can be worse on speed and far better on correctness, portability and maintenance, and for most work those matter more. The 1950s programmer with 4,096 words of memory did not have that luxury.

■ 15.5.3 PLAIN - a worked example

1. Take one line of mathematics: the roots of a quadratic need $d = b*b - 4*a*c$.
2. In a high-level language that is one line and you can see the formula. In 1950s assembly it is roughly this shape of work.

load b	get b into the accumulator
multiply b	now accumulator holds b*b
store temp1	park it
load four	get the constant 4
multiply a	now 4*a
multiply c	now 4*a*c
store temp2	park it
load temp1	get b*b back
subtract temp2	b*b - 4*a*c
store d	save the answer

3. Ten lines, two temporary locations you invented and must not reuse by accident, and one constant you placed in memory by hand.
4. Nothing in the assembly says “this is the discriminant of a quadratic”. You must comment it, and comments drift out of date.

■ 15.5.4 PLAIN - what is really happening inside

1. The gap a compiler must cross is bigger than it looks. Four problems hide inside “translate the formula”.
2. **Parsing**: work out from flat text that in $a + b * c$ the multiplication binds tighter, so it happens first.
3. **Temporaries**: the machine has a handful of registers and your expression has many intermediate values. Something must decide what lives where.

4. **Addressing:** $X(I)$ in a loop means a different memory address every time round, so the compiler must generate address arithmetic you never wrote.
5. **Optimization:** a naive translation reloads the same value from memory again and again. A human would not, and the compiler must not either.
6. The fourth is the hard one, and it decided whether high-level languages would be accepted at all.
7. A compiler producing code twice as slow as hand assembly would have been a curiosity. One that came within a few per cent would end the argument.

■ 15.5.5 TECHNICAL – the engineer’s version

1. The economics are the whole story. In 1957 an IBM 704 rented for roughly several tens of thousands of United States dollars per month, while a programmer earned a few hundred dollars per month. Machine time dominated.
2. John Backus, who led the FORTRAN project, described the prevailing view plainly: most programmers of the time believed no automatic system could produce object code comparable to a good hand coder’s.
3. The FORTRAN team’s target was output within roughly a factor of one to two of hand-written assembly for typical scientific code. They largely achieved it, and that is why FORTRAN spread.
4. Earlier interpreted “automatic programming” systems, such as Backus’s own Speedcoding for the IBM 701 in 1953, ran perhaps 10 to 20 times slower than hand code. Useful, but easy to dismiss.

Concern	Assembly	High-level language
Tied to one machine	yes	mostly no
Lines per formula	10 to 30	1
Type checking	almost none	some to strong
1957 speed penalty	zero	a few per cent to 2x

The honest version: the sceptics were not simply wrong and later proved foolish. For a decade they were right about specific cases, and even in 2026 there are hot loops in codecs, cryptography and kernels where hand-written assembly still beats a compiler. What changed is that those cases went from “most code” to a fraction of one per cent of it.

■ 15.5.6 WORDS – remember these

1. Portability — code that runs on more than one machine — source-level independence from a specific instruction set.
2. Parsing — working out the structure of written text — building a syntax tree from tokens according to a grammar.
3. Optimization — making generated code faster or smaller — semantics preserving transformation of an intermediate representation.
4. Object code — the machine code a translator produces — a relocatable module containing code, data and relocation entries.
5. Intermediate value — a part-finished result — a temporary held in a register or spilled to a stack slot.

15.6 Grace Hopper and the first compilers

■ 15.6.1 PLAIN – in simple words

1. Grace Hopper joined the United States Navy Reserve in 1943 and in 1944 was assigned to the Harvard Mark I under Howard Aiken.
2. The Mark I was an electromechanical calculator over 15 metres long. Hopper was one of its first programmers and wrote its manual, published in 1946 and running to over 500 pages.
3. In 1949 she moved to the Eckert-Mauchly Computer Corporation to work on UNIVAC I, the first commercial computer sold in the United States.
4. There she noticed something dull and important. Programmers kept copying the same routines out of notebooks by hand, and kept making copying errors.
5. Her idea: keep those routines on magnetic tape, and let a program fetch them and stitch them together.
6. She called that program a **compiler**, using the ordinary English meaning. To compile means to gather things from sources into one collection, as one compiles an anthology of poems.
7. The system was called A-0 and it worked in 1951 and 1952.
8. Here is the careful part. A-0 did not translate formulas into machine code. It gathered, positioned and joined ready-made routines.
9. By today's meanings, A-0 is much closer to what we now call a **linker** or **loader** than to a compiler.
10. That is not a criticism. The word changed meaning afterwards, partly because of her, and we are judging 1952 by a 2026 dictionary.

■ 15.6.2 PLAIN – a picture in your head

1. Picture a cookbook where each recipe is on its own card in a filing drawer.
2. You want a three-course dinner, so you write a short list: recipe 47, recipe 12 for 4 servings, recipe 90.
3. An assistant takes the list, pulls those cards, rewrites the quantities for the servings you asked, and staples them into one set of instructions.
4. That is A-0. The assistant invented no cooking. The assistant gathered, adjusted and joined.
5. A modern compiler is a different assistant. You say “a light fish dinner for four” and it writes the recipes itself.

Where this comparison breaks: A-0 did more than staple. It adjusted the addresses inside each routine so they worked at their new position in memory, and substituted the arguments you supplied. That address adjustment is called relocation, and it is real work, not clerical work.

■ 15.6.3 PLAIN – a worked example

1. In A-0 you wrote a program as a list of call numbers with arguments, not as machine instructions. The shape of it was like this, in spirit.

your input	what A-0 did with it
-----	-----
subroutine 015, X, Y	find routine 015 on the tape copy it into place fix its internal addresses plug in X and Y
subroutine 032, Y	same again for routine 032
subroutine 007, Y, Z	same again for routine 007
-----	-----
result: one runnable program on tape, ready to load	

2. Notice what is present: a library, selection from it, relocation, argument substitution, and output of a single runnable program.
3. Notice what is absent: no arithmetic expressions, no variables in the modern sense, no control statements, no code generation from a grammar.
4. Hopper described the work in a 1952 paper to the Association for Computing Machinery titled “The Education of a Computer”.
5. A-2 followed in 1953. Remington Rand UNIVAC shipped it to customers by the end of that year with the source code included, and invited customers to send improvements back. That is an early ancestor of open source practice.
6. Then came ARITH-MATIC (A-3), MATH-MATIC (AT-3), and the important one, B-0, better known as **FLOW-MATIC**.

■ 15.6.4 PLAIN – what is really happening inside

1. FLOW-MATIC, worked on from 1955 and substantially complete by 1959, is where Hopper’s real revolution sits.
2. Her observation was about people, not machines. Business managers threw out anything full of mathematical symbols.
3. So FLOW-MATIC used English words as its statements. It was the first programming language to do so.
4. Statements looked like COMPARE PRODUCT-NO (A) WITH PRODUCT-NO (B) and TRANSFER A TO D and READ-ITEM A ; IF END OF DATA GO TO OPERATION 14.
5. Her management initially thought English-language programming impractical. She proposed it in 1953, and the compiler became publicly available in early 1958. It fed almost directly into COBOL.

■ 15.6.5 TECHNICAL – the engineer’s version

1. Now the credit question, precisely, because it is asked constantly and answered badly. There are at least four serious claims, and they answer different questions.

Claim	Year	What it was
Corrado Bohm, thesis	1951	translator described on paper
Hopper, A-0	1952	library linker and loader
Glennie, Autocode	1952	real translator, ran on Mark 1
Backus et al, FORTRAN	1957	first widely used compiler

2. **Corrado Bohm**, doctoral thesis at ETH Zurich, submitted 1951, defined a language and described a translator for it written in that same language. It is arguably the first self-describing compiler. It was not implemented on a machine at the time.
3. **Grace Hopper, A-0**, 1951 to 1952 on UNIVAC I. She coined and popularized the term “compiler”. The system selected subroutines from a tape library by call number, relocated them, substituted arguments and produced a single runnable program.
4. In modern terminology, A-0 is a linking loader.
5. **Alick Glennie, Autocode**, 1952, for the Manchester Mark 1 at the Royal Armament Research Establishment. It accepted a symbolic, algebra-like notation and generated machine code, and many historians call it the first compiler in the modern sense.
6. Glennie’s own manual claimed the efficiency loss was no more than 10 per cent. It was barely used, even at Manchester.

7. **J. Halcombe Laning and Neal Zierler**, MIT Whirlwind, from 1952 and operational by 1954. It accepted algebraic formulas in near-mathematical notation and produced machine code, with subroutine linkage, arrays and indexing, and a Runge-Kutta differential equation solver.
8. It is often called the first operating algebraic compiler.
9. **R. A. Brooker's Mark 1 Autocode**, planned 1954 and working 1955, was the Manchester system actually used heavily, unlike Glennie's.
10. So the correct answers, depending on the question asked:
11. First thing called a compiler: Hopper's A-0, 1952.
12. First compiler described in the modern sense: Bohm's 1951 thesis, on paper only.
13. First running translator from a higher notation to machine code: Glennie's Autocode, 1952.
14. First algebraic compiler in practical use: Laning and Zierler, 1954.
15. First compiler that displaced hand assembly at scale: FORTRAN, 1957.
16. Where experts disagree: some historians, including commentary published by the Association for Computing Machinery, argue plainly that A-0 was not a compiler and the popular claim overstates it.
17. Others argue that in 1952 the word meant what Hopper made it mean, and that judging her by a later definition is unfair. Both positions are defensible.
18. What is not in dispute, and is enough on its own: Hopper coined the vocabulary of automatic programming, built the first practical subroutine library system, led FLOW-MATIC, the first English-language programming language, and shaped COBOL.
19. She reached the rank of Rear Admiral, retired in 1986, and died on 1 January 1992.

The honest version: "Grace Hopper wrote the first compiler" is defensible only if you use her 1952 definition of the word and not yours. The accurate sentence is that she wrote the first system anyone called a compiler, and it did linking and loading rather than translation. The exaggeration is unnecessary. Her uncontested achievements are larger than the disputed one.

■ 15.6.6 WORDS – remember these

1. Compiler — a program that turns your writing into machine code — a translator from a source language to machine or object code.
2. Linker — the tool joining separate pieces into one program — resolves external symbol references between object modules.
3. Loader — the tool putting a program into memory to run — performs relocation and transfers control to the entry point.
4. Relocation — fixing addresses when code moves — adjusting address fields for a load-time base other than the assembled one.
5. Subroutine library — a stored collection of reusable routines — an archive of object modules selected by name at link time.
6. FLOW-MATIC — the first English-worded language — B-0, Remington Rand, 1955 to 1959, direct ancestor of COBOL.

15.7 FORTRAN: the compiler that had to be fast

■ 15.7.1 PLAIN – in simple words

1. In late 1953 John Backus, aged 29 and working at IBM, sent his managers a proposal to let scientists write formulas instead of assembly.
2. The project ran from 1954. A draft specification was finished in November 1954, the first manual appeared in October 1956, and the compiler was delivered in April 1957 for the IBM 704.

3. The name is short for FORMula TRANslation.
4. The team knew the project would fail unless the output was fast. Programmers would not accept a 30 per cent penalty on a machine costing a fortune per hour.
5. So most of the effort went not into the language but into the optimizer, the part that makes the generated code good.
6. It worked. FORTRAN produced code close enough to hand-written assembly that experienced programmers accepted it.
7. Within a few years, writing scientific programs in assembly became a strange thing to do. That is the moment high-level programming won.

■ 15.7.2 PLAIN – a picture in your head

1. Imagine the first automatic gearbox. Every racing driver says a human changes gear better.
2. If it is clumsy and slow they are right, and it fails.
3. The FORTRAN team spent their effort making the gearbox change gear as well as a good driver, not on making the steering wheel prettier.
4. Once it did, the argument stopped being about speed and became about who wants to change gear 40,000 times on a journey.

Where this comparison breaks: an automatic and a manual gearbox drive the same car. A compiler changes what you can express, not just your comfort. Whole classes of program became possible because nobody had to track a thousand addresses by hand.

■ 15.7.3 PLAIN – a worked example

1. Here is a small, period-authentic FORTRAN program that adds 1 to 5 and prints the total. Compare it with the 6502 assembly earlier in this chapter.

```

C      ADD THE WHOLE NUMBERS 1 TO 5 AND PRINT THE TOTAL
      ITOTAL = 0
      DO 10 I = 1, 5
      ITOTAL = ITOTAL + I
10 CONTINUE
      PRINT 20, ITOTAL
20 FORMAT (9H TOTAL = , I3)
      STOP
      END

```

2. Line 1 starts with C in column 1, which marks a comment. That is a card convention: the whole language is laid out for 80-column cards.
3. Columns 1 to 5 hold a statement number, column 6 marks a continuation card, columns 7 to 72 hold the statement, and columns 73 to 80 are ignored because that is where the card sequence number went.
4. ITOTAL is an integer with no declaration. In FORTRAN a name beginning with I, J, K, L, M or N is an integer by default; everything else is real.
5. DO 10 I = 1, 5 means repeat down to statement 10 with I taking the values 1 to 5.
6. 9H TOTAL = is a Hollerith constant: the 9 says the next 9 characters are literal text. Quoted strings arrived later, in FORTRAN 77.
7. Nine lines. The equivalent assembly is several dozen, and the assembly does not survive a change of computer.

■ 15.7.4 PLAIN – what is really happening inside

1. The FORTRAN compiler had to do the four hard jobs listed earlier in this chapter, on a machine with 4,096 or 8,192 words of memory.
2. First it read a statement and worked out the structure of the expression, deciding what binds to what.
3. Then it turned DO loops into counters, comparisons and jumps, and turned array subscripts into address arithmetic.
4. Then came the part that mattered: deciding which values to keep in the 704's three index registers rather than reloading them from memory.
5. It also did what we now call common subexpression elimination and loop invariant motion: compute a thing once, and move work that does not change out of the loop.
6. Those techniques were essentially invented here, for this compiler, because there was no choice.
7. The compiler ran in several separate passes, reading and rewriting the program from tape between passes, because it could not all fit in memory.

■ 15.7.5 TECHNICAL – the engineer's version

1. Backus's team at IBM included Irving Ziller, Harlan Herrick, Robert Nelson, Roy Nutt, Sheldon Best, Richard Goldberg, Lois Haibt, David Sayre, Peter Sheridan and Harold Stern.
2. The effort is commonly cited at about 18 person-years over roughly three years for the original IBM 704 compiler.
3. Target machine: the IBM 704, a 36-bit word machine with hardware floating point, 3 index registers, and typically 4,096 to 32,768 words of core.

Version	Year	Notable addition
FORTTRAN	1957	first delivery, IBM 704
FORTTRAN II	1958	user subroutines, functions
FORTTRAN IV	1962	machine independence push
FORTTRAN 66	1966	first ANSI standard
FORTTRAN 77	1978	IF-THEN-ELSE, strings
Fortran 90	1991	free form, array syntax
Fortran 2023	2023	current ISO standard

4. FORTRAN 66 was the first standardized programming language in history. That is a standard in the strict sense: a written specification adopted by a standards body.
5. Backus received the ACM Turing Award in 1977 for FORTRAN. He had earlier written Speedcoding for the IBM 701 in 1953, an interpreted system, which taught him how badly interpretation cost.
6. Fortran is not a museum piece. As of 2026 it remains standard in weather forecasting, computational fluid dynamics, nuclear engineering and climate modelling, and the LINPACK and LAPACK numerical libraries beneath a great deal of scientific software began in Fortran.

The honest version: FORTRAN was neither the first high-level language nor the first compiler. It was the first that people used in large numbers, which is a different and more important claim. Autocode and Laning-Zierler came first and were barely adopted.

■ 15.7.6 WORDS – remember these

1. FORTRAN — formula translation — IBM's 1957 scientific language, the first widely adopted high-level language.
2. Optimizer — the part that makes code fast — the compiler passes performing semantics-preserving transformations.
3. Implicit typing — the language guessing a variable's type from its name — FORTRAN's I-to-N integer rule, switchable off with IMPLICIT NONE.
4. Fixed-form source — code laid out in fixed columns — label 1-5, continuation 6, statement 7-72, from the punched card.
5. Loop invariant motion — moving unchanging work out of a loop — a classic optimization introduced in the 1957 FORTRAN compiler.

15.8 COBOL, LISP and ALGOL: three different futures

■ 15.8.1 PLAIN – in simple words

1. Between 1958 and 1960 three languages appeared that between them set the shape of nearly everything since.
2. COBOL was for business. It reads like English sentences and is built around records, files and exact decimal money.
3. LISP was for symbols and for thinking about thinking. It is built around lists and functions, and programs in it are themselves lists.
4. ALGOL was for describing algorithms clearly. It was never widely used to run real work, and almost every modern language is its descendant.
5. COBOL came from a committee organized by the United States Department of Defense in 1959, with Grace Hopper as a technical adviser.
6. LISP came from one man, John McCarthy, at MIT in 1958.
7. ALGOL came from a joint European and American committee, and produced the ALGOL 60 report in January 1960.

■ 15.8.2 PLAIN – a picture in your head

1. Think of three documents: a company's accounting ledger, a mathematician's notebook, and a legal statute.
2. COBOL is the ledger. Every field has a fixed size, every amount has exactly two decimal places, and it must balance.
3. LISP is the notebook. Everything is written in one simple form, and you are free to write notes about your own notes.
4. ALGOL is the statute: a precise written definition of what the words mean, published so anyone can implement it exactly.

Where this comparison breaks: ALGOL 60 was a real running language, not just a document. But its lasting effect really is definitional, because the ALGOL 60 report taught the world how to specify a language properly.

■ 15.8.3 PLAIN – a worked example

1. Here is COBOL. The shape is deliberately unlike everything else.

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. PAYTOTAL.  
DATA DIVISION.
```

```

WORKING-STORAGE SECTION.
01 WS-TOTAL      PIC 9(7)V99 VALUE ZERO.
01 WS-AMOUNT     PIC 9(5)V99 VALUE 125.50.
PROCEDURE DIVISION.
    ADD WS-AMOUNT TO WS-TOTAL.
    DISPLAY "TOTAL IS " WS-TOTAL.
    STOP RUN.

```

2. PIC 9(7)V99 means seven digits, then an assumed decimal point, then two digits. The V is not stored; it marks where the point sits.
3. That single feature is why COBOL still runs banks. It stores money as exact decimal digits, not binary floating point, so 0.10 plus 0.20 is exactly 0.30 and never 0.30000000000000004.
4. Here is LISP. The whole language is this one shape.

```

(defun sum-to (n)
  (if (= n 0)
      0
      (+ n (sum-to (- n 1)))))

(sum-to 5) ; gives 15

```

5. Every LISP expression is a list in brackets: the first item is the operation, the rest are its arguments. That is all the syntax there is.
6. Because a program is a list, a program can build and run another program with no special machinery. That property is **homoiconicity**.

■ 15.8.4 PLAIN – what is really happening inside

1. COBOL's committee met at the Pentagon on 28 and 29 May 1959 with 41 attendees, chaired by Charles Phillips. An earlier meeting had been convened by Mary K. Hawes on 8 April 1959.
2. The goal was one language running on machines from every manufacturer, so the government would stop paying to rewrite the same payroll six times.
3. Grace Hopper was a technical adviser, not a committee member. Her FLOW-MATIC was the single strongest influence on the design.
4. Jean Sammet, who was on the committee, put the record straight bluntly: Hopper was not the mother, creator or developer of COBOL.
5. LISP started differently. McCarthy wrote a mathematical notation in a 1960 paper and did not intend it as a real language.
6. Steve Russell read the paper, realized the eval function described in it was itself an interpreter, and hand-compiled it into IBM 704 machine code. LISP became real by accident.
7. Two 704 assembly macros gave LISP two permanent words: car for the first item and cdr for the rest.
8. ALGOL's contribution is structural: blocks marked begin and end, names visible only inside the block where they are declared, and procedures that can call themselves.
9. If your language has curly brackets, local variables and recursion, it is an ALGOL descendant. C, Java, JavaScript, Python, Go and Rust all are.

■ 15.8.5 TECHNICAL – the engineer’s version

1. COBOL: COMmon Business-Oriented Language. The COBOL 60 specification was approved by the CODASYL executive committee on 8 January 1960. The first COBOL program ran on an RCA 501 on 17 August 1960.
2. In December 1960 the same COBOL program ran on both an RCA and a UNIVAC machine. That cross-vendor portability demonstration was the point of the whole exercise.

Figure	Value	Source and date
Lines of COBOL in use	200 billion	Gartner estimate, 1997
Business programs run	about 80%	Gartner estimate, 1997
Card swipes touching it	about 95%	widely cited, 2020

3. Treat newer figures with care. Claims such as “COBOL handles 3 trillion dollars of transactions a day” circulate widely and trace back to vendor material rather than an audited study.
4. The safe statement is that COBOL still runs core banking, insurance, payroll and government systems in 2026, and nobody has a credible plan to remove most of it.
5. LISP: John McCarthy, MIT, from 1958. Key paper: “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”, 1960. LISP introduced automatic garbage collection, dynamic typing, higher-order functions, the read-eval-print loop, and code as data.
6. LISP is the second-oldest high-level language still in use, after Fortran. Living dialects include Common Lisp, Scheme, Racket and Clojure.
7. ALGOL 58, originally called IAL, came from a Zurich meeting in 1958. ALGOL 60 came from a Paris meeting in January 1960 with 13 authors: Bauer, Naur, Rutishauser, Samelson, Vauquois, van Wijngaarden and Woodger from Europe, and Backus, Green, Katz, McCarthy, Perlis and Wegstein from the United States. Peter Naur edited the report.
8. The report’s grammar notation, invented by Backus and refined by Naur, is **Backus-Naur Form**. It is still how language grammars are written in 2026.
9. Tony Hoare’s verdict on ALGOL 60 is the most quoted line in language design: a language so far ahead of its time that it was an improvement not only on its predecessors but on nearly all its successors.

■ 15.8.6 WORDS – remember these

1. COBOL — the English-like business language — CODASYL, 1959 to 1960, built on records, files and fixed-point decimal.
2. PICTURE clause — the description of a field’s exact shape — COBOL’s PIC notation defining digits, sign and implied decimal point.
3. LISP — the list language — McCarthy, 1958: s-expressions, garbage collection and code as data.
4. Homoiconicity — programs made of the same stuff as data — source represented directly in the language’s own data structures.
5. ALGOL — the algorithm description language — ALGOL 58 and 60, source of block structure, lexical scope and recursion.
6. Backus-Naur Form — the standard way of writing a grammar — a metasyntax for context-free grammars, from the ALGOL 60 report.

15.9 The moth, and what a bug really is

■ 15.9.1 PLAIN – in simple words

1. On 9 September 1947, operators of the Harvard Mark II found the machine giving wrong answers.
2. They traced the fault to relay number 70 in panel F, and found a moth caught between the contacts.
3. They removed the moth, taped it into the logbook, and wrote beside it: “First actual case of bug being found.”
4. That logbook page, moth still attached, is at the Smithsonian’s National Museum of American History.
5. It is a lovely story and it is almost always told wrongly.
6. The word “bug” for a fault did not start there. It was already old engineering slang.
7. Thomas Edison used it in writing in 1878, describing how little faults and difficulties, “as such little faults and difficulties are called”, show themselves and take months to clear.
8. The joke in the logbook only works because everyone already used the word. For once they had found an actual, literal bug.
9. Grace Hopper was associated with the Mark II team and told the story for decades. Accounts differ on whether she was in the room, and the entry is generally attributed to others on the team.

■ 15.9.2 PLAIN – a picture in your head

1. Imagine a workshop where people have called every annoying fault a “gremlin” for seventy years.
2. One day someone opens a machine and finds a small carved gremlin figure that a child dropped in.
3. They pin it to the wall and write “first actual gremlin found”. Everyone laughs, because the word was already worn smooth.
4. A century later people say “that is where the word gremlin comes from”, which reverses the joke completely.

Where this comparison breaks: unlike gremlins, insects really did cause failures in relay machines. Relay contacts were open, mechanical and warm, and buildings were not sealed. It was a genuine failure mode, not a one-off.

■ 15.9.3 PLAIN – a worked example

1. Now the harder question. How did you debug in 1949, when there was no screen and printing a line cost machine time you did not have?
2. Method one: the console lights. A row of lamps showed the accumulator or the program counter in binary, and you read the bits.
3. Method two: single step. A switch made the machine execute one instruction per button press, so you could watch a register change.
4. Method three: the post-mortem routine. A small program left in memory that, after your program stopped, printed a chosen block of memory.
5. Method four: the checking routine, or trace. A program that ran yours one instruction at a time and printed what each did. Stanley Gill described this for EDSAC in a 1951 paper on diagnosing mistakes in programmes.
6. Method five: sound. EDSAC and the Ferranti Mark 1 had a loudspeaker wired to a bit of a register. Programmers learned the sound of a healthy program and heard a stuck loop instantly.
7. Method six: desk checking. You read your own code on paper, pretending to be the machine, writing register values in a column.

8. All six survive today, renamed: registers view, single step, core dump, instruction trace, profiling by observation, and code review.

■ 15.9.4 PLAIN – what is really happening inside

1. A trace routine is really an interpreter. It fetches one of your instructions, prints it, performs it itself, and moves on.
2. A post-mortem dump is a different idea. It does not slow your program at all. It captures the wreckage afterwards.
3. That trade-off has never gone away. It is exactly the choice between running under a debugger and reading a crash dump.
4. Breakpoints were originally physical. You replaced an instruction in memory with a halt, ran until the machine stopped, read the lights, then put the original instruction back.
5. Modern debuggers do the identical trick. On x86 a breakpoint is the single byte CC, the INT 3 instruction, written over your code and restored when execution stops.

■ 15.9.5 TECHNICAL – the engineer’s version

1. Harvard Mark II: an electromechanical relay machine completed in 1947 for the United States Navy at Dahlgren, Virginia, built by Howard Aiken’s group.
2. The 9 September 1947 log entry sits at 15:45 in the logbook. The relay is recorded as Relay 70, Panel F.
3. Earlier uses of “bug”: Edison’s 1878 letter; the 1896 edition of Hawkins’ New Catechism of Electricity defines “bug” as a fault or trouble in an apparatus.
4. The term was standard in radio and telephony by the 1920s, and “debug” is documented in aviation in the 1940s, before the moth.
5. Terminology today, precisely: a **fault** is the defect in the code, an **error** is the wrong internal state it causes, and a **failure** is the externally visible wrong behaviour. That three-level split is standard in dependability engineering.
6. Tools that replaced the console lights: gdb and lldb for interactive debugging, strace and dtrace for system call tracing, perf for sampling, valgrind and AddressSanitizer for memory errors, and core dumps controlled by ulimit -c and /proc/sys/kernel/core_pattern on Linux.

The honest version: the popular claim “the word bug comes from a moth in a computer in 1947” is simply false, and the people who wrote the log entry knew it, which is exactly why they wrote “first actual case”. The story is true; the etymology attached to it is not.

■ 15.9.6 WORDS – remember these

1. Bug — a fault in a program — a defect in source or design; the word is engineering slang predating computing by at least 70 years.
2. Debugging — finding and removing faults — systematic fault localization, from observed failure back to defect.
3. Post-mortem dump — a printout of memory after a crash — a core dump, the captured process image at failure time.
4. Trace — a record of each step performed — an instruction-level execution log, historically produced by interpretation.
5. Breakpoint — a place where the machine stops for you — a trap instruction patched over the target instruction, restored on hit.
6. Fault, error, failure — the defect, the bad state, the visible wrongness — the standard three-level dependability vocabulary.

15.10 The rise of the operating system as software

■ 15.10.1 PLAIN – in simple words

1. In the early 1950s there was no operating system, because there was nothing for one to do.
2. One program ran at a time, and a human being was the operating system.
3. The machine sat idle during all of that, and the machine was the expensive part.
4. So people wrote a small program that stayed in memory permanently and did the operator's job: read the next job, run it, clean up, read the next.
5. That resident program was called a **monitor**, and it is the ancestor of every operating system.
6. The next problem was worse. A program waiting for a tape to rewind left the processor idle for seconds at a time.
7. The answer was to keep several programs in memory and switch to another whenever the current one had to wait.
8. Then came the obvious next question: if we can switch between programs, we can switch between people. Give everyone a terminal and take turns. That is **time sharing**.

■ 15.10.2 PLAIN – a picture in your head

1. Think of one very expensive oven in a village.
2. At first each family carries its dish in, lights the oven, cooks, and carries it out, and the oven cools between families.
3. Then the village hires a permanent baker who keeps the oven hot and takes dishes in a queue. That is the batch monitor.
4. Then the baker puts several dishes in at once, turning to whichever needs attention. That is multiprogramming.
5. Then the baker gives every family thirty seconds of attention in rotation, so each feels the oven is theirs. That is time sharing.

Where this comparison breaks: the oven's heat is genuinely shared, whereas a processor gives each program the whole machine for a short slice. The illusion of simultaneity comes from switching faster than a human can notice, not from dividing capacity.

■ 15.10.3 PLAIN – a worked example

1. Consider a 1957 job on an IBM 704 that computes for 2 seconds.
2. Without a monitor: 4 minutes of operator handling, 2 seconds of computing. Machine utilization is under 1 per cent.
3. With a batch monitor and a stack of jobs on one tape: handling drops to under a second per job, and utilization rises above 50 per cent.
4. With multiprogramming: while job A waits 30 milliseconds for a disk, job B uses those 30 milliseconds, and utilization approaches 90 per cent.
5. With time sharing: 30 users each get slices of about 100 milliseconds, and each feels alone on the machine as long as none is doing heavy work.
6. The economics justify the complexity. A machine costing tens of thousands of dollars a month sitting idle at 1 per cent is a catastrophe, and that is why operating systems exist at all.

■ 15.10.4 PLAIN – what is really happening inside

1. A monitor needs three abilities, and each one required new hardware.
2. It must regain control from a program that will not stop. That needs a **timer interrupt**: a clock that forces the processor into the monitor.

3. It must stop one program corrupting another. That needs **memory protection**: hardware that checks every address.
4. It must stop a program driving the tape drive directly. That needs **privileged mode**: some instructions work only for the monitor.
5. Those three inventions are the boundary between a program and an operating system. Without them, a monitor is only a convention programs may ignore.
6. The word “kernel” means exactly this: the part that runs in privileged mode and that everything else must go through.

■ 15.10.5 TECHNICAL – the engineer’s version

1. GM-NAA I/O, written in 1956 by Robert Patrick of General Motors Research and Owen Mock of North American Aviation for the IBM 704, is generally described as the first operating system.
2. It was followed by the FORTRAN Monitor System, then IBSYS on the IBM 7090 and 7094 family.
3. Job Control Language exists because a batch monitor must be told where one job ends and the next begins. IBM’s JCL statements still start with two slashes in columns 1 and 2, a punched-card artefact.
4. CTSS, the Compatible Time-Sharing System, was first demonstrated by Fernando Corbato at MIT in November 1961 on an IBM 709, with three user consoles.
5. It moved to an IBM 7090 in 1962 and a 7094 later, entered routine service in summer 1963, and could support up to 112 teleprinter terminals through an IBM 7750 controller.
6. CTSS was the first system with password login, and carried some of the first electronic mail and text formatting tools.
7. Multics, from 1965, was the joint MIT, General Electric and Bell Labs successor. It introduced segmented virtual memory, a hierarchical file system, dynamic linking and rings of protection.
8. Multics was late, huge and slow to arrive, and Bell Labs withdrew in 1969. That withdrawal directly caused UNIX, which is the next section.

System	Year	Idea it added
GM-NAA I/O	1956	resident batch monitor
IBSYS	1960	full job stream management
CTSS	1961	interactive time sharing
Multics	1965	virtual memory, protect rings

9. Chapter 18 takes this up properly: scheduling, virtual memory, system calls and modern kernels. Here we needed only the reason the operating system had to become software in the first place.

■ 15.10.6 WORDS – remember these

1. Monitor — the small permanent program that runs jobs — the resident supervisor of a batch system.
2. Batch — jobs run back to back with no user present — non-interactive scheduling from a job queue.
3. Multiprogramming — several programs in memory, one running — overlapping input and output with computation.
4. Time sharing — many people using one machine at once — preemptive scheduling with short quanta over interactive terminals.
5. Kernel — the privileged core of the operating system — code executing in supervisor mode, mediating hardware access.
6. Privileged mode — an instruction level only the system may use — supervisor or ring 0 execution enforced by the processor.

15.11 UNIX and C

■ 15.11.1 PLAIN – in simple words

1. In 1969 Bell Labs pulled out of Multics, and Ken Thompson had a large, ambitious operating system project taken away from him.
2. He had written a game called Space Travel and wanted somewhere to run it. He found a little-used PDP-7 in a corner.
3. With Dennis Ritchie and others, and with a month of free time while his wife was away, he wrote a tiny operating system for it.
4. It had a hierarchical file system, processes and a small set of tools. It was called Unics as a joke on Multics, and the spelling settled as UNIX.
5. UNIX was written in assembly at first, like everything else.
6. Then came the radical part. Thompson wrote a language called B, and Ritchie grew it into a language called C between 1971 and 1973.
7. In 1973 they rewrote the UNIX kernel itself in C.
8. Nobody did that. An operating system had to be assembly, because it touches hardware and must be fast.
9. But it worked, and the consequence was enormous: to move UNIX to a new computer you now wrote a C compiler for it, not a whole new operating system.
10. That is where portable software begins.

■ 15.11.2 PLAIN – a picture in your head

1. Imagine every city building its trains to a different rail gauge, so every train must be rebuilt from scratch to move city to city.
2. Now someone builds an adjustable axle. The carriage design stays the same; only the axle is refitted.
3. C is the adjustable axle. The compiler is refitted per machine, and the software above it moves across.
4. The carriage is not perfectly identical everywhere. Some of it still has to be adjusted by hand, and the parts touching the rails always do.

Where this comparison breaks: C's portability is not automatic, it is a discipline. C deliberately leaves things such as integer size and byte order unspecified, and code assuming one machine's answers is not portable, no matter that it is written in C.

■ 15.11.3 PLAIN – a worked example

1. Here is the same “add 1 to n” job from earlier, written in C, next to what a compiler produces for it.

```
int sum_to(int n) {
    int total = 0;
    for (int i = 1; i <= n; i++)
        total += i;
    return total;
}
```

2. Below is the x86-64 assembly from GCC 13.3.0 with optimization turned off, in Intel syntax. Six lines of C become nineteen instructions.

```
sum_to:
    push    rbp
```

```

    mov     rbp, rsp
    mov     DWORD PTR -20[rbp], edi    ; save n
    mov     DWORD PTR -8[rbp], 0      ; total = 0
    mov     DWORD PTR -4[rbp], 1      ; i = 1
    jmp     .L2
.L3:
    mov     eax, DWORD PTR -4[rbp]    ; load i
    add     DWORD PTR -8[rbp], eax    ; total += i
    add     DWORD PTR -4[rbp], 1      ; i++
.L2:
    mov     eax, DWORD PTR -4[rbp]    ; load i
    cmp     eax, DWORD PTR -20[rbp]   ; compare with n
    jle     .L3                      ; loop if i <= n
    mov     eax, DWORD PTR -8[rbp]    ; return total
    pop     rbp
    ret

```

3. Notice .L2 and .L3. Those are labels, exactly the idea from the 6502 assembly earlier in this chapter. The compiler invents them, so no human counts bytes.
4. Notice the loop is compiled with the test at the bottom and one jump into it. That shape is a choice the compiler made, not something you wrote.
5. Turn optimization on and the function changes completely: the compiler unrolls the loop and computes two values per pass. That is the compiler choosing, which an assembler never does.
6. You can see this on any Linux machine with `gcc -S -masm=intel -O0 file.c` and then `gcc -S -masm=intel -O2 file.c`.

■ 15.11.4 PLAIN – what is really happening inside

1. C was designed to sit exactly one small step above the machine, and no higher.
2. Its types map onto machine word sizes, its pointers are memory addresses, and its arrays are pointer arithmetic in disguise.
3. That closeness is why a C compiler is small and easy to write for a new processor, and why UNIX could be moved.
4. It is also why C is dangerous. There is no bounds checking, because the machine has none, and the language does not add what the machine lacks.
5. The 1973 rewrite made the UNIX kernel about a third larger and somewhat slower than the assembly version, and Thompson and Ritchie accepted that trade deliberately.
6. They got back something worth far more: they could read their own kernel, change it safely, and move it.
7. In 1978 Brian Kernighan and Dennis Ritchie published “The C Programming Language”. The book was so precise and so short that it became the definition of the language for over a decade, known simply as K and R.

■ 15.11.5 TECHNICAL – the engineer’s version

1. Timeline: Bell Labs left Multics in 1969; UNIX started on a PDP-7 in 1969; moved to a PDP-11/20 in 1970; C developed from B and BCPL through 1971 and 1972; Version 4 UNIX rewritten in C in 1973.
2. The paper by Ritchie and Thompson, presented at the ACM Symposium on Operating Systems Principles in 1973 and published in Communications of the ACM in July 1974, is the announcement that made UNIX famous.
3. Version 6 UNIX, released in 1975, was licensed cheaply to universities. John Lions’s 1976 commentary on its source code was for years the most photocopied document in computing.

4. UNIX's design rules became a culture: everything is a file, programs do one thing, programs are joined by pipes, and text is the universal interface.
5. C standards: K and R C from 1978, ANSI X3.159-1989 (C89), ISO/IEC 9899:1990 (C90), then C99, C11, C17, and C23 published in 2024.

Year	Event	People
1969	Bell Labs leaves Multics	Bell Labs
1969	UNIX begins on a PDP-7	Thompson, Ritchie
1972	C takes shape	Ritchie
1973	UNIX kernel rewritten in C	Thompson, Ritchie
1978	The C Programming Language	Kernighan, Ritchie

6. Thompson and Ritchie received the ACM Turing Award in 1983.
7. As of 2026 the direct descendants of this work include Linux, the BSDs, macOS and iOS (through NeXTSTEP and BSD), and Android's userspace model. The POSIX standards, IEEE 1003, are the written form of the UNIX interface.

The honest version: C is not portable in the sense people usually mean. The standard deliberately leaves behaviour unspecified, undefined or implementation-defined in hundreds of places. What C gives you is a portable language with a small compiler, not portable programs for free.

■ 15.11.6 WORDS – remember these

1. UNIX — the small operating system from Bell Labs, 1969 — a multi-user, multi-process system whose interface is standardized as POSIX.
2. C — the language UNIX was rewritten in, 1972 — a statically typed systems language with direct memory access and manual lifetime management.
3. Portability — software that moves between machines — source compatibility given a conforming compiler and defined behaviour.
4. The 1973 rewrite — the moment an operating system stopped being assembly — Version 4 UNIX in C, trading size and speed for maintainability.
5. K and R — the 1978 book that defined C — Kernighan and Ritchie, "The C Programming Language", the pre-standard reference.

15.12 How software was shared, and then sold

■ 15.12.1 PLAIN – in simple words

1. For the first fifteen years, software was not sold. It came free with the computer, because the computer was the product.
2. Users swapped programs directly. IBM's user group SHARE, founded in 1955, ran a library that members contributed to and drew from.
3. In 1969 IBM changed that. Facing antitrust pressure, it announced it would price software separately from hardware. That decision, effective in 1970, is where the software industry begins.
4. When home computers arrived in the mid-1970s, sharing came back, because the people involved were hobbyists with no money.
5. Magazines printed the full source code of programs, and you typed them in by hand, then hunted for your own typing mistakes.
6. The Homebrew Computer Club, first meeting on 5 March 1975 in a garage in Menlo Park, California, was the centre of this. Members copied paper tapes and passed them round freely.

7. In January 1976 a 20-year-old Bill Gates wrote an angry letter to that community, saying most of them were stealing his BASIC and that this would stop anyone writing good software.
8. He was making an argument nobody had needed to make before: that software by itself is a product you buy.
9. Later, others argued the opposite just as forcefully, and built an entire parallel world on it.

■ 15.12.2 PLAIN – a picture in your head

1. Think of recipes. For centuries they spread freely; cooks copied them, improved them and passed them on.
2. Then someone opens a restaurant and says the recipe is the business, not the meal, and copying it is theft.
3. Some cooks agree and buy licences. Others say a recipe you may not read or change is not a recipe at all, and start a movement to keep them open.
4. Both worlds now exist side by side, and most restaurants use some of each.

Where this comparison breaks: recipes are not protected by copyright in most places, whereas software source code clearly is. The legal ground under the argument is completely different, and that is why licences, not manners, ended up deciding it.

■ 15.12.3 PLAIN – a worked example

1. Here is what “sharing software” physically meant, in order of era.
2. 1955 to 1968: a magnetic tape posted to a user group, and a printed catalogue of routines you could request.
3. 1975 to 1979: a punched paper tape passed hand to hand at a club meeting, or a program listing printed in a magazine that you typed in yourself.
4. 1978 to 1990: a floppy disk, posted or copied at a club, and later a bulletin board system you dialled with a modem at 300 or 1,200 bits per second.
5. 1991 onwards: an FTP site, then the web, then a version control server.
6. 2008 onwards: a public git repository, which is where nearly all of it lives in 2026.
7. The pattern is constant. Each drop in the cost of copying produced a burst of sharing, and a matching argument about payment.

■ 15.12.4 PLAIN – what is really happening inside

1. Gates’s letter is worth understanding rather than cheering or booing.
2. The facts he cited: MITS was shipping about 1,000 Altair computers a month at the end of 1975, while paid copies of BASIC numbered in the low hundreds.
3. He said about 40,000 dollars of computer time went into writing it, and that the royalty worked out at roughly two dollars an hour for the work.
4. Critics answered that the computer time figure was inflated accounting, and that Microsoft’s BASIC had itself been developed on a university machine.
5. Both things can be true. The letter is still the clearest early statement that software has a development cost that must be paid for somehow.
6. In 1983 Richard Stallman made the opposite argument, prompted by a printer whose driver he was not allowed to fix.
7. His answer was not to ask nicely. It was a licence that used copyright law against itself: you may copy and change this, provided anything you distribute carries the same permission.
8. That trick is called **copyleft**, and it is the mechanism, not the slogan, that made free software durable.

■ 15.12.5 TECHNICAL – the engineer’s version

1. IBM’s unbundling announcement came on 23 June 1969 and took effect in January 1970. Before it, software and support were included in the hardware price; after it, they were priced separately.
2. Homebrew Computer Club: first meeting 5 March 1975, Gordon French’s garage, Menlo Park. The Altair 8800 had appeared on the cover of Popular Electronics dated January 1975.
3. A pre-release paper tape of Altair BASIC went missing at a MITS event in Palo Alto in June 1975; Dan Sokol duplicated it, and about 50 copies appeared at the next Homebrew meeting.
4. Gates wrote “An Open Letter to Hobbyists” in January 1976. It was published in the Homebrew Computer Club Newsletter dated 3 February 1976 and in MITS Computer Notes for February 1976, and reprinted widely.
5. Richard Stallman announced the GNU Project, a recursive joke standing for “GNU’s Not Unix”, in a Usenet message on 27 September 1983. Development began January 1984, the GNU Manifesto appeared in 1985, and the Free Software Foundation was founded in October 1985.

Licence	Year	Core idea
GPL version 1	1989	copyleft, source must follow
GPL version 2	1991	the Linux kernel licence
GPL version 3	2007	patents and locked hardware
MIT / BSD	1980s	permissive, no copyleft

6. The four freedoms of free software: run the program for any purpose, study and change it, redistribute copies, and distribute your modified versions. Access to source code is a precondition of the second and fourth.
7. Linus Torvalds posted his announcement to the comp.os.minix newsgroup on 25 August 1991, describing a free operating system as “just a hobby, won’t be big and professional like gnu”. Version 0.01 followed on 17 September 1991.
8. Linux moved to the GPL with version 0.12 in February 1992 and reached version 1.0 in March 1994. Combining the Linux kernel with the GNU tools produced the first complete free operating system.
9. The term “open source” was coined at a meeting in Palo Alto on 3 February 1998, following Netscape’s decision to release its browser source. The Open Source Initiative was founded that month.
10. Free software and open source are not synonyms, and the disagreement is real rather than pedantic. Free software is an ethical argument about user freedom. Open source is an argument about development quality and business practicality. They mostly cover the same code.

■ 15.12.6 WORDS – remember these

1. Unbundling — selling software separately from hardware — IBM’s 1969 announcement, the origin of the commercial software industry.
2. Copyleft — a licence that forces the freedom onward — a reciprocal licence requiring derivative distributions under the same terms.
3. GPL — the GNU General Public License — the reciprocal licence family, versions 1 (1989), 2 (1991) and 3 (2007).
4. Permissive licence — you may do almost anything — MIT and BSD style terms with attribution but no reciprocity.
5. Free software — free as in freedom, not price — the four freedoms defined by the Free Software Foundation.

6. Open source — publicly readable and modifiable code — the Open Source Initiative definition, from 1998.

15.13 A short history of the idea of a “programmer”

■ 15.13.1 PLAIN – in simple words

1. Before 1945, “computer” was a job title for a person, usually a woman, who did calculations with a desk machine.
2. The first people to program electronic machines were drawn from that pool and were called operators, not programmers.
3. The work was thought of as clerical, because the design of the machine was assumed to be the intellectual part.
4. That was badly wrong, and it took about fifteen years to correct.
5. Through the 1950s the word “programmer” settled in, and “coder” came to mean a junior who turned someone else’s plan into instructions.
6. By the mid-1960s, projects had grown so large that they routinely failed. Systems were years late, far over budget, and full of faults.
7. In 1968 a NATO conference gave that a name: the **software crisis**. It also made popular a deliberately provocative term for the cure: **software engineering**.
8. The provocation was the point. If bridges can be engineered with method and discipline, why not programs.
9. Almost sixty years later the argument is unfinished, and it is still the most useful question in the field.

■ 15.13.2 PLAIN – a picture in your head

1. Think of building. First there were people who put up a shed, and it stood up or it did not.
2. Then buildings got taller, and a shed-builder’s instincts stopped scaling. People fell.
3. So the trade split: architects, structural engineers, surveyors, inspectors, codes of practice and licences.
4. Software went through the first two stages and got stuck partway through the third.

Where this comparison breaks: a bridge’s load is physics and does not argue. A program’s requirements change every week, and it is expected to be modified constantly for decades. No civil engineer is asked to change the span of a finished bridge on a Thursday.

■ 15.13.3 PLAIN – a worked example

1. The example everyone in 1968 had in mind was IBM’s OS/360, the operating system for the System/360 family announced in 1964.
2. It ran to thousands of people and years of delay, and adding people made it later, not earlier.
3. Fred Brooks, who managed it, wrote up the lessons in “The Mythical Man-Month” in 1975.
4. His central observation: a task that takes 12 months for 1 person does not take 1 month for 12 people, because they must all talk to each other.
5. With n people there are n times $(n - 1)$ divided by 2 communication paths. For 12 people that is 66. For 50 people it is 1,225.
6. Brooks’s law, in his own compressed form: adding people to a late software project makes it later.
7. That is the sharpest single sentence about software management ever written, and it is still true in 2026.

■ 15.13.4 PLAIN – what is really happening inside

1. The 1968 conference proposed answers that are now everyday practice: specification before coding, modular design, reviews, testing as a distinct activity, and measurement.
2. Alongside it, a related movement argued that programs should be structured so a human can reason about them.
3. Edsger Dijkstra's letter "Go To Statement Considered Harmful", published in Communications of the ACM in March 1968, is the famous shot in that campaign.
4. His argument was not aesthetic. It was that with unrestricted jumps you cannot look at a line of code and know how you got there, so you cannot reason about the program at all.
5. What replaced the goto was exactly the block structure ALGOL 60 had introduced: sequence, choice and loops with one way in and one way out.
6. So the theory and the management story meet here. Structured programming was a response to a management crisis.

■ 15.13.5 TECHNICAL – the engineer's version

1. The earliest known use in print of "software" in the modern sense is by John Tukey, in "The Teaching of Concrete Mathematics", American Mathematical Monthly, January 1958.
2. The NATO Software Engineering Conference was held at Garmisch, Germany, from 7 to 11 October 1968, with roughly 50 participants. A second conference followed at Rome in October 1969.
3. Who coined "software engineering" is genuinely disputed. Margaret Hamilton, who led the Apollo onboard flight software at the MIT Instrumentation Laboratory, states she began using the term during Apollo in the mid-1960s.
4. Anthony Oettinger used it in an ACM president's letter in 1966, and the 1968 NATO conference made it general.
5. All three claims can be accurate at once, and the safe statement is that the 1968 conference established the term.
6. Dijkstra's 1972 Turing Award lecture summarized the crisis: with no machines there was no programming problem, with weak machines a mild one, and with powerful machines an equally gigantic one.
7. A 1968 study by Sackman, Erikson and Grant reported differences of an order of magnitude or more between individual programmers on the same task.
8. The headline "10x programmer" comes from that work. Its method has been criticized ever since, so the size of the effect is active debate rather than established fact.

Year	Term or event	Significance
1946	"operator"	programming seen as clerical
1958	"software" in print	Tukey, AMM, January 1958
1968	NATO Garmisch	"software engineering" named
1968	Go To Considered Harmful	structured programming
1975	The Mythical Man-Month	Brooks, on team scaling

The honest version: software engineering is still not engineering in the sense that civil engineering is. There is no universal licensure, no legally binding code of practice in most jurisdictions, and no agreed body of predictive knowledge saying how long a given program will take. Experts genuinely disagree on whether that is a failing to be fixed or a reflection of a different kind of work.

■ 15.13.6 WORDS – remember these

1. Software — the instructions, not the machine — programs and data as distinct from hardware; earliest known print use, Tukey, 1958.
2. Software crisis — projects late, costly and broken — the recognition, named in 1968, that project growth outran available method.
3. Software engineering — treating programming as a disciplined craft — the application of engineering method to software, named at NATO 1968.
4. Brooks's law — adding people to a late project makes it later — the communication overhead of $n(n-1)/2$ channels swamps added capacity.
5. Structured programming — code shaped so you can reason about it — single entry, single exit constructs replacing arbitrary jumps.

15.98 Common wrong ideas

1. Wrong: a program is a special kind of thing stored differently from data. Right: it is ordinary numbers in ordinary memory, treated as instructions only because the program counter points at them.
2. Wrong: Grace Hopper wrote the first compiler as we use the word today. Right: she coined the word and built A-0 in 1952, which selected, relocated and joined library routines. That is a linking loader, not a translator.
3. Wrong: nothing existed before FORTRAN. Right: Corrado Bohm's 1951 thesis described a translator on paper, Glennie's Autocode ran in 1952, and Laning and Zierler's algebraic system ran on Whirlwind by 1954.
4. Wrong: the word "bug" comes from the 1947 moth. Right: Edison used it in 1878, and the logbook entry says "first actual case" precisely because the word was already old slang.
5. Wrong: ENIAC took two weeks to reprogram, full stop. Right: planning took weeks, physical setup days and debugging days more, and the 1948 conversion changed the whole picture by cutting setup to hours.
6. Wrong: assembly language is always one line to one instruction. Right: macros, pseudo-instructions and assembler-chosen branch widths break that on most real assemblers.
7. Wrong: COBOL is obsolete. Right: it still runs core banking, insurance and government payroll in 2026, largely because its exact decimal arithmetic is correct for money in a way binary floating point is not.
8. Wrong: high-level languages won because programmers wanted comfort. Right: they won when FORTRAN's optimizer made the generated code fast enough that the speed objection collapsed.
9. Wrong: C is portable, so C programs run anywhere. Right: C is a portable language with a small compiler. Programs depending on integer size, byte order or undefined behaviour are not portable at all.
10. Wrong: free software means free of charge. Right: it means the four freedoms to run, study, share and modify. You may sell it, and the Free Software Foundation says so explicitly.

15.99 Chapter summary in 20 lines

1. A program at the bottom is a list of numbers in memory that a processor reads as instructions, one after another.
2. Seven bytes on a 6502 make a complete working program, and you can trace it by hand with nothing but an opcode table.
3. ENIAC in 1946 had no program storage at all. You wired it with cables and set 1,200-way switch banks, and a change took days.
4. Its six principal programmers, McNulty, Jennings, Snyder, Wescoff, Bilas and Lichterman, were classified as operators and taught themselves the machine.
5. The 1948 conversion gave ENIAC a stored instruction set and cut setup from days to hours at about a sixfold speed cost.
6. Programs then went in as raw numbers: front-panel switches, punched paper tape, and above all the 80-column punched card.
7. A card holds 12 rows by 80 columns; a digit is one punch, a letter is a zone punch plus a digit punch, and that is Hollerith encoding.
8. Batch processing meant handing a deck to an operator and waiting hours or a day for a printout, so programmers desk-checked everything first.
9. Assembly language replaced numbers with mnemonics and hand-counted addresses with labels, and an assembler recomputed the numbers every time.
10. EDSAC's Initial Orders by David Wheeler in 1949, the Wheeler Jump, the 87-routine library, and the 1951 Wilkes, Wheeler and Gill book built the idea of reusable software.
11. Assembly is tied to one machine, slow to write and barely checked, so people asked whether a program could write the machine code instead.
12. The objection was economic, not snobbish: machine time cost far more than programmer time, so a translator had to lose almost no speed.
13. Grace Hopper coined "compiler" and built A-0 in 1952. It gathered subroutines from a tape library, relocated them and joined them, which today we call linking and loading.
14. Bohm's 1951 thesis, Glennie's 1952 Autocode and the 1954 Laning-Zierler system all have claims to "first compiler" under different definitions.
15. FORTRAN, from Backus's team at IBM, ran from 1954 to delivery in April 1957, and won because its optimizer matched hand-written assembly closely enough for scientists to accept it.
16. COBOL came from the CODASYL committee in 1959 with FLOW-MATIC as its main ancestor, and its exact decimal arithmetic keeps it running finance in 2026.
17. LISP in 1958 gave us code as data, garbage collection and recursion, and ALGOL 60 gave nearly every later language its block structure.
18. The moth of 9 September 1947 is real and is in the Smithsonian, but the word "bug" was already 70 years old when it was taped in.
19. Batch monitors from 1956, then CTSS time sharing in 1961 and Multics, made the operating system a permanent program that manages other programs.
20. UNIX from 1969 and its 1973 rewrite in C made operating systems portable, and everything since, from Linux in 1991 to the machine in front of you, sits on that foundation.

How a Compiler Actually Works

16.0 What this chapter gives you

1. You will be able to name every stage between the text you type and the electricity that runs, in the right order, and say what each one does.
2. You will be able to take one five-word C function and follow it through preprocessing, tokens, tree, symbol table, intermediate code, optimization, assembly, object file, executable and running process.
3. You will be able to read real assembly output and explain every single line.
4. You will be able to say why compilers use an intermediate language in the middle instead of going straight to machine instructions.
5. You will be able to explain why `-O2` turned our program into three instructions, and why that is legal.
6. You will be able to explain what a linker does, why “undefined reference” happens, and why the order of libraries on the command line matters.
7. You will be able to explain what happens at the moment you press Enter on a program, including the dynamic linker, the GOT and the PLT.
8. You will be able to compare compiled, bytecode and interpreted execution with real numbers, and explain why JavaScript got fast.
9. You will be able to explain the bootstrap problem and Ken Thompson’s 1984 trusting-trust attack to somebody who has never programmed.
10. You will be able to run the commands yourself and see the same output.

Our one running example, for the whole chapter, is this file. We will call it `add.c`. Everything below is that file at a different moment of its life.

```
int add(int a, int b) { return a + b; }  
int main(void) { return add(2, 3); }
```

Every command shown was run on Ubuntu 24.04 on an x86-64 machine, with GCC 13.3.0 and Clang 18.1.3. The outputs pasted here are the real outputs.

16.1 The whole pipeline in one view

■ 16.1.1 PLAIN – in simple words

1. A computer chip does not understand the words you type. It understands numbers, and only a small fixed set of them.
2. A **compiler** is a program that reads your text and writes those numbers.

3. It does not do this in one jump. It does it in a chain of small steps.
4. Each step takes one shape of information and turns it into a simpler shape.
5. First the text is cleaned up and glued together with other files.
6. Then it is chopped into small pieces, like words in a sentence.
7. Then those pieces are arranged into a tree, the way a sentence has a subject and an object.
8. Then the meaning is checked: do these names exist, do the types match.
9. Then it is rewritten in a simple made-up language that is easy to improve.
10. Then it is improved: shortened, simplified, things that do nothing removed.
11. Then it is written as instructions for one particular chip.
12. Then those instructions become raw bytes in a file.
13. Then several such files are glued into one program.
14. Then the operating system loads it into memory and starts it.

■ 16.1.2 PLAIN – a picture in your head

1. Think of a factory line that turns a handwritten recipe into a packed meal.
2. Station one gathers all the recipe pages that the main page refers to.
3. Station two reads the pages aloud, one word at a time.
4. Station three works out the grammar: which words are the ingredients, which are the actions, what belongs inside what.
5. Station four checks it makes sense: you cannot fry a number.
6. Station five rewrites the whole thing as a plain numbered work order.
7. Station six removes wasted steps: no need to boil water you never use.
8. Station seven writes the work order in the exact words this kitchen's machines accept.
9. Station eight packs it in a box. Station nine puts several boxes in a crate.
10. Station ten unpacks the crate onto a table and starts cooking.

Where this comparison breaks: a factory line moves one item forward and never looks back. A real compiler often loops. The optimizer runs dozens of passes over the same code, some of them many times, and a later pass can undo an earlier one. Also, several stations can be the same piece of code in memory, not separate programs. The clean chain is a teaching model, not the shape of the source code inside GCC or Clang.

■ 16.1.3 PLAIN – a worked example

Here is the chain, drawn for `add.c`.

```

add.c    (the text you typed)
  |
  v
[ preprocessor ]  glue in #include, expand #define
  |
  v
[ lexer ]        text -> tokens: int, add, (, int, a, ...
  |
  v
[ parser ]        tokens -> a tree of the program
  |
  v
[ semantic ]      names, scopes, types, symbol table
  |
  v
[ IR builder ]    tree -> simple three-address code
  |

```

```

      v
[ optimizer ]    fold, propagate, inline, delete
      |
      v
[ code generator ] IR -> assembly text for one CPU
      |
      v
[ assembler ]    assembly -> object file add.o (bytes)
      |
      v
[ linker ]       add.o + C library -> one executable
      |
      v
[ loader ]       exec: map into memory, bind symbols
      |
      v
a running process

```

One line for each stage, for `add.c`:

1. Preprocessor: nothing to do here, there are no # lines. Output is the same two lines of text.
2. Lexer: produces 32 tokens plus an end marker.
3. Parser: produces a tree with two function declarations at the top.
4. Semantic analysis: records that `add` takes two `int` and returns `int`, and that the call in `main` matches.
5. IR: produces `_3 = a + b; return _3;` for `add`.
6. Optimizer at `-O2`: notices `add(2,3)` is always 5.
7. Code generator: writes `leal (%rdi,%rsi), %eax` for `add`.
8. Assembler: writes the bytes `8d 04 37` for that instruction.
9. Linker: fills in the address of `add` inside `main`'s call instruction.
10. Loader: maps the file into memory and jumps to the entry point.
11. The process returns 5. `echo $?` prints 5.

■ 16.1.4 PLAIN - what is really happening inside

1. Most of these “stages” are functions inside one program, passing data structures to each other, not separate processes.
2. But some really are separate programs. On Linux, `gcc` is only a driver: it runs other programs and passes files between them.
3. Running `gcc -v add.c` shows the driver's real children. The three that matter are `cc1`, `as` and `collect2`.
4. `cc1` is the actual C compiler. It reads `add.c` and writes assembly text to a temporary file.
5. `as` is the assembler. It reads that text and writes a temporary `.o` file.
6. `collect2` is a wrapper around `ld`, the linker. It writes the executable.
7. So on a normal Linux box, compiling one C file starts three extra programs.
8. Each stage narrows what is possible. After lexing, you can no longer ask about spaces. After parsing, you can no longer ask about brackets. After code generation, you can no longer ask about variable names.
9. That narrowing is the point. Every stage throws away information the next stage does not need, so the next stage can be simpler.

■ 16.1.5 TECHNICAL – the engineer’s version

1. The classical decomposition is **front end**, **middle end**, **back end**. The front end is language specific, the back end is target specific, and the middle end is neither.
2. The front end covers preprocessing, lexical analysis, syntax analysis and semantic analysis, and emits an intermediate representation (IR).
3. The middle end runs target-independent optimization passes on the IR.
4. The back end performs instruction selection, instruction scheduling and register allocation, and emits assembly or machine code directly.
5. The canonical reference is *Compilers: Principles, Techniques, and Tools* by Alfred Aho, Ravi Sethi and Jeffrey Ullman, 1986, known as the Dragon Book. The 2006 second edition adds Monica Lam.
6. The first true optimizing compiler was the IBM FORTRAN compiler for the IBM 704, delivered in April 1957 under John Backus. It took about 18 staff-years and its output had to beat hand-written assembly to be accepted.
7. Grace Hopper’s A-0 system of 1952 predates it but was closer to a linking loader than to a modern compiler.
8. GCC 1.0 was released by Richard Stallman in March 1987. LLVM began as Chris Lattner’s work at the University of Illinois, with LLVM 1.0 in 2003.

Observing the pipeline with real commands:

```
gcc -v add.c -o prog      # show cc1, as, collect2
gcc -E add.c             # stop after the preprocessor
clang -Xclang -dump-tokens -fsyntax-only add.c
clang -Xclang -ast-dump -fsyntax-only add.c
clang -S -emit-llvm add.c -o add.ll
gcc -S add.c -o add.s     # stop after code generation
gcc -c add.c -o add.o     # stop after the assembler
```

Stage	GCC name	Clang / LLVM name
Preprocess	cpp inside cc1	clang -E
Parse + types	cc1 front end	clang front end
Middle IR	GIMPLE, then RTL	LLVM IR
Optimize	tree and RTL passes	LLVM pass manager
Emit assembly	cc1 back end	LLVM target backend
Assemble	GNU as	integrated assembler
Link	ld via collect2	lld or ld

■ 16.1.6 WORDS – remember these

Compiler — a program that turns your text into machine numbers — a translator from a source language to a target language, usually with static checking. Pass — one walk over the program — a single traversal of the IR performing one analysis or transformation. Front end — the part that understands your language — lexing, parsing, semantic analysis, IR generation. Back end — the part that knows the chip — instruction selection, scheduling, register allocation, code emission. Driver — the program you actually type — gcc or clang, which orchestrates the real compiler, assembler and linker as child processes.

16.2 The preprocessor

■ 16.2.1 PLAIN – in simple words

1. Before the compiler looks at your program, another tool edits the text.
2. That tool is the **preprocessor**. It only understands lines starting #.
3. `#include "file.h"` means: delete this line and paste that whole file here.
4. `#define TWO 2` means: everywhere the word `TWO` appears later, write `2`.
5. `#if` and `#ifdef` mean: keep this block of text or throw it away.
6. That is all it does. It moves text around. It does not know what C is.
7. It does not know types. It does not know functions. It cannot count.
8. The compiler never sees your `#include` lines. It sees the result.

■ 16.2.2 PLAIN – a picture in your head

1. Imagine a school essay where you are told to write `see page 40` instead of copying a long definition.
2. Before the teacher marks it, a helper goes through and physically pastes `page 40` in place of every `see page 40`.
3. The helper also has a list of shorthands: wherever you wrote `WHO`, write `World Health Organization`.
4. The helper does not read for meaning. If you wrote `see page 40` inside a joke, `page 40` gets pasted into the joke.
5. The teacher then marks one long essay with no shortcuts left in it.

Where this comparison breaks: the helper in this story is careful. The C preprocessor is not. It will happily paste text that produces nonsense, and it will report the error at the pasted line, not at your line. Macro expansion also happens repeatedly until nothing changes, which no human helper would do.

■ 16.2.3 PLAIN – a worked example

Split our example into a header and a source file.

```
/* addh.h */
#ifndef ADD_H
#define ADD_H
#define TWO 2
#define THREE 3
int add(int a, int b);
#endif

/* main2.c */
#include "addh.h"
int main(void) { return add(TWO, THREE); }
```

Now run `gcc -E main2.c`. The real tail of the output is:

```
# 1 "main2.c"
# 1 "addh.h" 1

int add(int a, int b);
# 2 "main2.c" 2
int main(void) { return add(2, 3); }
```

1. The `#include` line has gone. The declaration from the header sits in its place.
2. `TWO` and `THREE` have become `2` and `3`.

3. The `#ifndef` guard lines have gone. They produced no text.
4. The lines beginning `#` with numbers are **line markers**. They tell the compiler “the next line really came from file X, line N”, so error messages point at your file and not at the merged text.
5. The blank lines are the preprocessor keeping line numbers lined up.

Now a real size measurement, on the same machine:

```
$ printf '#include <stdio.h>\nint main(void){return 0;}\n' > hello.c
$ wc -l < hello.c
2
$ gcc -E hello.c | wc -l
815
$ gcc -E hello.c | wc -c
21292
```

1. Two lines of yours became 815 lines and about 21 kB of text.
2. It pulled in 29 distinct files.
3. In C++ it is far worse. `#include <iostream>` with GCC 13.3 on this machine expanded to 36,584 lines.

■ 16.2.4 PLAIN - what is really happening inside

1. The preprocessor runs in phases defined by the C standard. The important ones, in order, are: join lines ending in a backslash, replace comments with a single space, then execute the `#` directives, then join adjacent string literals.
2. The **header guard** trick, `#ifndef ADD_H / #define ADD_H / #endif`, exists because a header may be included twice through different paths.
3. Without a guard, the second include pastes the same declarations again, which is an error for anything defined rather than merely declared.
4. With the guard, the second visit finds `ADD_H` already defined and skips the whole file, producing no text.
5. `#pragma once` does the same job in one line. It is not in the C or C++ standard, but every major compiler supports it. That makes it a convention, not a standard.
6. Conditional compilation, `#if defined(_WIN32)`, is how one source file serves several operating systems. The text for the other systems never reaches the compiler at all.
7. Macros are textual, so they have famous traps. `#define SQ(x) x*x` then `SQ(1+2)` becomes `1+2*1+2`, which is 5, not 9. Parentheses fix it: `#define SQ(x) ((x)*(x))`.

■ 16.2.5 TECHNICAL - the engineer's version

1. The C preprocessor is specified in the C standard, clause 6.10. It is a standard, not an implementation detail.
2. Translation phases 1 to 4 of the standard cover trigraph mapping, line splicing, comment removal and directive execution.
3. Predefined macros include `__FILE__`, `__LINE__`, `__DATE__`, `__TIME__`, `__STDC__` and `__STDC_VERSION__`. `__STDC_VERSION__` is 199901L for C99, 201112L for C11, 201710L for C17 and 202311L for C23.
4. `#include <x.h>` searches the system include path. `#include "x.h"` searches the current directory first, then the system path. `gcc -I dir` prepends a directory. `gcc -E -v` prints the search list actually used.
5. Macro expansion is not recursive on the macro currently being expanded, which is what stops `#define X X` from looping forever.

6. `#` in a function-like macro stringizes its argument. `##` pastes tokens together. Both are standard.
7. Preprocessing cost is real. Header expansion is the main reason C++ builds are slow, and is the motivation for precompiled headers and for C++20 modules, standardized in ISO C++20 and still only partially implemented in toolchains as of 2026.

Command	What it shows
<code>gcc -E f.c</code>	Preprocessed text
<code>gcc -E -P f.c</code>	Same, without line markers
<code>gcc -dM -E f.c</code>	Every macro defined
<code>gcc -H -c f.c</code>	Header include tree
<code>gcc -MMD -c f.c</code>	Write a .d dependency file

The honest version: saying “the preprocessor is a separate program” is a useful lie. It once was, as `/lib/cpp`. In GCC today it is a library linked into `cc1`, and Clang implements it as a token source feeding the lexer, so text is never fully materialized. You get the same answer either way, which is why the lie is safe.

■ 16.2.6 WORDS – remember these

Preprocessor — the text editor that runs before the compiler — the phase that executes `#` directives, per C standard clause 6.10. Macro — a shorthand replaced by text — an object-like or function-like replacement list expanded during translation phase 4. Header guard — a trick to stop a file being pasted twice — an `#ifndef` / `#define` / `#endif` idiom giving idempotent inclusion. Translation unit — one source file plus everything it pasted in — the input the compiler proper actually sees. Conditional compilation — keeping or dropping blocks of text — `#if`, `#ifdef`, `#elif`, `#else`, `#endif` evaluated on preprocessing tokens.

16.3 Lexing: characters into tokens

■ 16.3.1 PLAIN – in simple words

1. After the text is glued together, it is still just a row of characters.
2. The **lexer** reads that row left to right and groups characters into words.
3. Each word it produces is a **token**: a small labelled piece.
4. `int` becomes a keyword token. `add` becomes an identifier token. `2` becomes a number token. `+` becomes an operator token.
5. Spaces, tabs, newlines and comments produce no tokens at all. They only separate things.
6. The lexer does not care about order. `+++)` is fine by the lexer. It is the next stage that objects.
7. The lexer’s only real skill is to always take the longest match. In `a+++b` it produces `a`, `++`, `+`, `b`, not `a`, `+`, `+`, `+`, `b`.

■ 16.3.2 PLAIN – a picture in your head

1. Imagine a long strip of paper with no spaces: `THECATSATONTHEMAT`.
2. Your job is to cut it into words and label each one: noun, verb, article.
3. You have a list of legal words. You scan from the left, and at each point you take the longest legal word you can.
4. You never look ahead at the sentence’s meaning. You never ask if the sentence is sensible. You just cut and label.
5. You hand the next person a stack of labelled cards, in order.

Where this comparison breaks: English needs meaning to cut correctly. C is designed so that cutting never needs meaning. That was a deliberate language design decision, and languages that break it, such as C++ with its >> in nested templates, cause real trouble for their own compilers.

■ 16.3.3 PLAIN – a worked example

This is the real token stream for our file, from `clang -Xclang -dump-tokens -fsyntax-only add.c`, trimmed to the token and the text:

<code>int 'int'</code>	<code>identifier 'main'</code>
<code>identifier 'add'</code>	<code>l_paren '('</code>
<code>l_paren '('</code>	<code>void 'void'</code>
<code>int 'int'</code>	<code>r_paren ')'</code>
<code>identifier 'a'</code>	<code>l_brace '{'</code>
<code>comma ','</code>	<code>return 'return'</code>
<code>int 'int'</code>	<code>identifier 'add'</code>
<code>identifier 'b'</code>	<code>l_paren '('</code>
<code>r_paren ')'</code>	<code>numeric_constant '2'</code>
<code>l_brace '{'</code>	<code>comma ','</code>
<code>return 'return'</code>	<code>numeric_constant '3'</code>
<code>identifier 'a'</code>	<code>r_paren ')'</code>
<code>plus '+'</code>	<code>semi ';'</code>
<code>identifier 'b'</code>	<code>r_brace '}'</code>
<code>semi ';'</code>	<code>eof ''</code>
<code>r_brace '}'</code>	

1. Read the left column top to bottom, then the right column. That is the whole file: 32 tokens, then an end-of-file marker.
2. The keywords `int`, `void` and `return` get their own token kinds.
3. `add`, `main`, `a` and `b` are all `identifier`. The lexer does not know that `add` is a function and `a` is a parameter. It has no idea what they are.
4. `2` and `3` are `numeric_constant`. The text is kept, not the value. Working out that `2` means the number two happens later.
5. Every space and every newline vanished.
6. Clang also records the exact line and column of every token. That is why its error messages can point a caret at one character.

■ 16.3.4 PLAIN – what is really happening inside

1. A lexer is a machine with a small memory: a **finite automaton**.
2. It has a current state, it reads one character, and that character plus the state decides the next state. That is all the memory it has.
3. Start in state “nothing”. See a letter, go to state “reading an identifier”. Keep reading letters and digits. See something else, stop, and emit an identifier token.
4. Start in state “nothing”. See a digit, go to state “reading a number”. Keep reading digits. Stop at the first non-digit.
5. Because the memory is one state and nothing else, a lexer cannot count. It cannot check that brackets match. That job belongs to the parser.
6. After building an identifier, the lexer looks the text up in a fixed table of keywords. If `int` is in the table, it is a keyword. If `add` is not, it is an identifier. This is the standard trick.
7. Handling comments, string literals with escapes, and numeric suffixes such as `10UL` are all extra states in the same machine.

■ 16.3.5 TECHNICAL – the engineer’s version

1. Lexical structure is specified with **regular expressions**. Regular expressions and finite automata describe exactly the same class of languages, a result from Stephen Kleene’s work in 1951 and 1956.
2. The standard mechanical route is: regular expression -> nondeterministic finite automaton (NFA) by Thompson’s construction, 1968 -> deterministic finite automaton (DFA) by subset construction -> minimized DFA -> a table.
3. That construction is why generated lexers are fast. Each input character costs one table lookup, so lexing is linear in the input size.
4. **Lex** was written by Mike Lesk and Eric Schmidt at Bell Labs, described in a 1975 technical report. **Flex**, the free rewrite, was written by Vern Paxson starting in 1987 and is what most Unix systems ship today.
5. The rules in a `.l` file are pattern-action pairs. Flex resolves conflicts by two fixed rules: longest match wins, and among equal-length matches, the rule written earliest wins.
6. Production compilers usually hand-write the lexer instead. Clang’s `Lexer.cpp` and GCC’s `libcpp` are hand-written, because hand-written code handles error recovery, source locations and pre-processor interaction better.
7. The **maximal munch** rule is in the C standard: the next token is the longest character sequence that could form one. It is why C++11 needed a special case so `>>` can close two templates.

Token class	Example text	C standard term
Keyword	<code>int, return</code>	keyword
Identifier	<code>add, a</code>	identifier
Number	<code>2, 0x1f, 3.5f</code>	constant
String	<code>"three"</code>	string literal
Operator	<code>+, ==, -></code>	punctuator

■ 16.3.6 WORDS – remember these

Token — one labelled word of the program — the smallest unit the parser consumes, carrying a kind, spelling and source location. Lexer — the tool that cuts text into tokens — the scanner implementing the lexical grammar, usually as a DFA. Regular expression — a pattern for matching text shapes — a formula denoting a regular language, equivalent in power to a finite automaton. Finite automaton — a machine whose whole memory is which state it is in — a five-tuple of states, alphabet, transition function, start state and accepting states. Maximal munch — always take the longest legal word — the longest-match rule mandated by the C and C++ lexical grammars.

16.4 Parsing: tokens into a tree

■ 16.4.1 PLAIN – in simple words

1. A row of tokens has no shape. `return a + b ;` is a flat list.
2. The **parser** gives it shape by building a tree.
3. In the tree, `+` sits above `a` and `b`, because `+` is the thing being done and `a` and `b` are the things it is done to.
4. `return` sits above the `+`, because returning is done to the result of the addition.
5. The tree is called an **abstract syntax tree**, or AST.

6. It is called abstract because the brackets and semicolons disappear. Their only job was to tell the parser the shape, and now the shape is the tree.
7. A **syntax error** is the parser reaching a token that cannot legally follow what it has already seen.
8. That is all a syntax error is. It is not about meaning. It is about shape.

■ 16.4.2 PLAIN – a picture in your head

1. Think of the sentence “the cat sat on the mat”.
2. A flat list of six words tells you nothing about who did what.
3. A tree tells you: the action is “sat”, the doer is “the cat”, the place is “on the mat”.
4. Now a rule set: a sentence is a noun phrase followed by a verb phrase. A noun phrase is an article followed by a noun. Follow the rules and the tree assembles itself.
5. A syntax error is “the cat sat on the”: you have run out of sentence while a rule still needs a noun.

Where this comparison breaks: English is ambiguous, so the same sentence can have several correct trees. “I saw the man with the telescope” has two. Real programming language grammars are designed hard to avoid this, and where it creeps in, the language adds a written tie-breaking rule, such as C’s “an else belongs to the nearest unmatched if”.

■ 16.4.3 PLAIN – a worked example

The rules of a language are written in **Backus-Naur Form**. Here is a tiny slice, enough for our function:

```
function    ::= type identifier "(" params ")" block
block       ::= "{" statement* "}"
statement   ::= "return" expression ";"
expression  ::= expression "+" term | term
term        ::= identifier | number | call
call        ::= identifier "(" arglist ")"
```

Read `::=` as “is made of”, `|` as “or”, `*` as “zero or more”. Now the real AST for our file. This is a redrawing of the actual output of `clang -Xclang -ast-dump -fsyntax-only add.c`:

```
TranslationUnit
|
+- FunctionDecl add : int (int, int)
|   +- ParmVarDecl a : int
|   +- ParmVarDecl b : int
|   +- CompoundStmt
|       +- ReturnStmt
|           +- BinaryOperator '+' : int
|               +- DeclRefExpr a : int
|               +- DeclRefExpr b : int
|
+- FunctionDecl main : int (void)
    +- CompoundStmt
        +- ReturnStmt
            +- CallExpr : int
                +- DeclRefExpr add : int (int, int)
                +- IntegerLiteral 2 : int
                +- IntegerLiteral 3 : int
```

1. TranslationUnit is the whole file.
2. Under it are exactly two things, our two functions.
3. CompoundStmt is the `{ ... }` block. The braces themselves are gone.
4. BinaryOperator `'+'` has two children. That is the whole meaning of `a + b`.
5. CallExpr has three children: what to call, then each argument in order.

6. The semicolons are gone. The parentheses are gone. Nothing is lost, because the tree already says what they said.

And a real syntax error, from `gcc -c synerr.c`:

```
synerr.c:1:36: error: expected expression before ';' token
  1 | int add(int a, int b) { return a + ; }
    |                                     ^
```

The parser had read `a` and `+`, so its rule demanded another expression. It got `;`. No rule allows that, so it stopped and reported the column.

■ 16.4.4 PLAIN – what is really happening inside

1. There are two main families of parser: top-down and bottom-up.
2. **Top-down** starts from “this should be a function” and tries to prove it by matching smaller and smaller pieces.
3. **Recursive descent** is top-down written by hand. You write one function per grammar rule. `parseStatement` calls `parseExpression`, which calls `parseTerm`. The call stack of the compiler mirrors the tree being built.
4. **Bottom-up** does the opposite. It pushes tokens onto a stack, and whenever the top of the stack matches the right-hand side of a rule, it replaces those items with the left-hand side. That is called a reduce.
5. Bottom-up parsers are usually generated by a tool from a grammar file, because the tables are far too tedious to write by hand.
6. **Precedence** is the rule that `2 + 3 * 4` is 14, not 20. A hand-written parser gets it by layering: the `+` function calls the `*` function, so `*` binds tighter and sits deeper in the tree.
7. **Associativity** is the rule that `a - b - c` means `(a - b) - c`. It decides whether the tree leans left or right.

■ 16.4.5 TECHNICAL – the engineer’s version

1. Programming language syntax is described by a **context-free grammar** (CFG), level 2 in the Chomsky hierarchy from Noam Chomsky’s 1956 classification.
2. Backus-Naur Form came from John Backus in 1959 and Peter Naur’s editing of the ALGOL 60 report in 1960. Donald Knuth proposed the name in 1964. Extended BNF adds `*`, `+`, `?` and grouping.
3. **LL(k)** parsers read Left to right and build a Leftmost derivation, with `k` tokens of lookahead. Recursive descent is the hand-written form of **LL(1)**.
4. **LR(k)** parsers read Left to right and build a Rightmost derivation in reverse. **LALR(1)** is the practical variant used by Yacc and Bison. LR grammars are strictly more powerful than LL grammars.
5. **Yacc** was written by Stephen C. Johnson at Bell Labs, published as a 1975 technical report. **GNU Bison** is the free replacement, from Robert Corbett in the mid-1980s. **ANTLR**, by Terence Parr from 1989, generates adaptive **ALL(*)** parsers.
6. Real compilers have moved back to hand-written recursive descent. GCC replaced its Bison C++ parser in GCC 3.4, 2004, and its C parser in GCC 4.1, 2006. Clang was recursive descent from the start. The reason is error messages and recovery, not speed.
7. Two classic grammar problems. The **dangling else** is settled by a rule in the C standard. The C **typedef ambiguity** makes `(A)*b` a cast or a multiplication depending on whether `A` is a type.
 1. C parsers settle the second by feeding symbol table facts back into the parser. That breaks the clean stage separation, and is called the lexer hack.

8. Operator precedence in C has 15 levels. Precedence climbing, also called Pratt parsing after Vaughan Pratt's 1973 paper, handles all of them in one compact function instead of 15 nested ones.

Parser style	Reads	Typical tool
Recursive descent	LL(1), hand	none, written by hand
LALR(1) table	bottom-up	Yacc, Bison
GLR	ambiguous CFG	Bison in GLR mode
PEG / packrat	ordered choice	pegjs, Rust pest
ALL(*)	adaptive LL	ANTLR 4

■ 16.4.6 WORDS – remember these

Parser — the part that finds the shape of the program — the syntax analyser that builds a parse tree or AST from a token stream. Abstract syntax tree — a tree of what the code does — a tree of language constructs with punctuation removed, the front end's main data structure. Grammar — the written rules of the language's shape — a context-free grammar, usually given in BNF or EBNF. Recursive descent — one function per grammar rule — a hand-written predictive top-down parser, normally LL(1) with local backtracking. Syntax error — the shape is wrong — no production allows the current token given the parser state; reported with the source location of that token.

16.5 Semantic analysis: does it make sense

■ 16.5.1 PLAIN – in simple words

1. A sentence can have correct shape and still be nonsense. "The number ate the Tuesday" is grammatical and meaningless.
2. **Semantic analysis** is the stage that catches the meaningless ones.
3. It asks: does this name exist here? Have you used it before declaring it? Have you declared it twice?
4. It asks: do the types fit? Can you add these two things? Can you pass this thing where that thing is expected?
5. It builds a **symbol table**: a list of every name, what it is, what type it has, and where it is visible.
6. Most of the errors a working programmer sees come from this stage, not from the parser.

■ 16.5.2 PLAIN – a picture in your head

1. Imagine a hotel with a guest register at the front desk.
2. When somebody checks in, you write their name and room in the register.
3. When a letter arrives for "Mr Smith", you look the name up. If he is not in the register, the letter cannot be delivered.
4. The hotel has floors. Each floor has its own small register. When looking up a name you check the floor you are on, then the floor below, then the ground floor register.
5. When a floor closes for the night, its small register is thrown away, and those names stop meaning anything.

Where this comparison breaks: a hotel register is one flat list per floor. A real symbol table also records types, storage class, linkage, whether the thing is a function or a variable, and how the compiler will refer to it later. And hotel names are unique; C deliberately lets an inner name hide an outer one.

■ 16.5.3 PLAIN – a worked example

The symbol table for our file, after analysing it, holds these entries:

Name	Kind	Type	Scope
add	function	int (int, int)	file
a	parameter	int	body of add
b	parameter	int	body of add
main	function	int (void)	file

1. When the parser reaches `a + b` inside `add`, semantic analysis looks up `a`. It finds the parameter, whose type is `int`.
2. It looks up `b`. Also `int`.
3. It checks that `+` accepts two `int` values. It does. The result type of the whole expression is recorded as `int`.
4. It checks that the type being returned matches the function's declared return type. `int` and `int`. Fine.
5. In `main`, it looks up `add`, finds a function of type `int (int, int)`, checks the call has two arguments, and checks each argument's type.

Now break it on purpose. Change the call to `add(2, "three")`. Real output from `clang -c typeerr.c`:

```
typeerr.c:2:32: error: incompatible pointer to integer
      conversion passing 'char[6]' to parameter of type 'int'
    2 | int main(void) { return add(2, "three"); }
      |                               ^~~~~~
typeerr.c:1:20: note: passing argument to parameter 'b' here
    1 | int add(int a, int b) { return a + b; }
      |               ^
1 error generated.
```

1. The shape of this program is perfectly legal. The parser was happy.
2. Only the type checker complains, and it can quote two places: the call and the declaration. It could only do that because the symbol table remembered where `add` was declared.
3. GCC 13.3 makes this a warning by default and still compiles. Clang 18 makes it an error. Experts disagree on default strictness; GCC 14 turned several such warnings into errors in 2024. Use `-Werror` to be strict.

■ 16.5.4 PLAIN – what is really happening inside

1. The symbol table is usually a stack of hash tables, one per scope.
2. Entering a `{` pushes a new table. Leaving `}` pops it.
3. Looking up a name searches the top table, then the one below, and so on. The first hit wins. That is exactly why an inner variable hides an outer one.
4. Type checking walks the AST from the leaves upward. Each node computes its own type from its children's types.
5. `IntegerLiteral 2` says “I am `int`”. `DeclRefExpr a` asks the symbol table and says “I am `int`”. `BinaryOperator +` sees two `ints` and says “I am `int`”.
6. Where types nearly match, C inserts a silent conversion. The Clang AST for our file shows `ImplicitCastExpr <LValueToRValue>` around `a` and `b`. That node means “read the value out of the variable”. The compiler added it; you did not write it.

7. **Type inference** is the same walk in reverse: the compiler works out a type instead of checking a written one. In C++ `auto x = 2 + 3;` the type of `x` is deduced from the right-hand side.

■ 16.5.5 TECHNICAL – the engineer’s version

1. Semantic analysis covers name resolution, scope and lifetime rules, type checking, implicit conversion insertion, constant expression evaluation, and language-specific rules such as C’s requirement that a case label be a constant.
2. C has four scopes in the standard: file, block, function prototype and function (labels only). Linkage is separate from scope: `static` at file level means internal linkage, no `static` means external linkage.
3. Clang stores this in a Sema object with an `IdentifierResolver` and a chain of `DeclContexts`. GCC uses `binding_level` structures. Both amount to a scoped hash table.
4. Full **Hindley-Milner** inference, from J. Roger Hindley in 1969 and Robin Milner in 1978, infers types for a whole program without annotations. ML, OCaml and Haskell use it. Rust and C++ use weaker local inference, so errors stay near the mistake.
5. The dividing line matters for error quality. Missing semicolon is a parse error. Undeclared identifier, wrong argument count, wrong argument type, assigning to a `const`, duplicate definition and returning the wrong type are all semantic errors.
6. Borrow checking in Rust and lifetime analysis are semantic analysis too, done on a control-flow graph after the AST. They are the reason `rustc` is slower than a C front end.
7. Useful flags: `gcc -Wall -Wextra -Werror`, `gcc -fsyntax-only` to stop after this stage, `clang -Xclang -ast-dump` to see the typed tree.

Error	Stage that catches it
Missing ;	Parser
Unbalanced {	Parser
Undeclared variable	Semantic
Wrong argument type	Semantic
Wrong number of args	Semantic
Missing function body	Linker
Divide by zero at run time	Nobody, it crashes

■ 16.5.6 WORDS – remember these

Symbol table — the list of every name and what it means — a scoped mapping from identifier to declaration, type, storage class and linkage. Scope — the region where a name is visible — the block, file, prototype or function region defined by the language standard. Type checking — making sure the pieces fit — verifying every expression’s type against the operator or context that consumes it. Implicit conversion — a change the compiler adds silently — a cast node inserted into the AST, such as C’s integer promotions and lvalue-to-rvalue conversion. Name resolution — deciding which declaration a name refers to — lookup through the scope chain, obeying shadowing and linkage rules.

16.6 Intermediate representation

■ 16.6.1 PLAIN – in simple words

1. After the tree is checked, the compiler could write chip instructions straight away. Almost none do.

2. Instead they rewrite the program in a simple made-up language of their own. That is the **intermediate representation**, or IR.
3. The IR is deliberately boring. Every line does one tiny thing. No nesting, no shortcuts, no cleverness.
4. Boring is useful. It is much easier to spot a wasted step in a flat list of tiny operations than in a tree full of nested expressions.
5. It is also useful because the IR does not belong to any chip. The same IR can later be turned into instructions for many different chips.
6. And it does not belong to any language either. C, C++, Rust and Swift can all produce the same IR, and then share one optimizer and one code generator.

■ 16.6.2 PLAIN - a picture in your head

1. Think of an international airport with one central hub.
2. There are many airlines flying in from many countries, and many destinations flying out.
3. Without a hub you would need a direct route from every origin to every destination. Ten origins and ten destinations means a hundred routes.
4. With a hub, every origin flies to the hub, and the hub flies to every destination. Ten plus ten. Twenty routes instead of a hundred.
5. The IR is the hub. Languages fly in, chips fly out.

Where this comparison breaks: a passenger arrives at the hub unchanged. A program does not. Most of the real work, and most of the compiler's total running time, happens while the program is sitting in the hub being rewritten.

■ 16.6.3 PLAIN - a worked example

Here is our add function in GCC's IR, called GIMPLE, from `gcc -O1 -fdump-tree-gimple=/dev/stdout -S add.c`:

```
int add (int a, int b)
{
    int D.2746;
    D.2746 = a + b;
    return D.2746;
}

int main ()
{
    int D.2748;
    D.2748 = add (2, 3);
    return D.2748;
}
```

1. Notice the temporary name D.2746. The compiler invented it.
2. Every statement has at most one operator and at most three names: one destination and two sources. That is why this style is called **three-address code**.
3. Now the same thing in Static Single Assignment form, from `gcc -O1 -fdump-tree-ssa=/dev/stdout -S add.c`:

```
int add (int a, int b)
{
    int _3;
    <bb 2> :
    _3 = a_1(D) + b_2(D);
```

```

    return _3;
}

```

4. Every name now has a version number. `a_1`, `b_2`, `_3`.
5. `<bb 2>` is a **basic block**: a run of instructions with no branch in the middle. Control enters at the top and leaves at the bottom.
6. And here is our whole file in LLVM IR, real output from `clang -O0 -S -emit-llvm add.c`, with the attribute lines removed:

```

define dso_local i32 @add(i32 noundef %0, i32 noundef %1) {
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 %0, ptr %3, align 4
    store i32 %1, ptr %4, align 4
    %5 = load i32, ptr %3, align 4
    %6 = load i32, ptr %4, align 4
    %7 = add nsw i32 %5, %6
    ret i32 %7
}

define dso_local i32 @main() {
    %1 = alloca i32, align 4
    store i32 0, ptr %1, align 4
    %2 = call i32 @add(i32 noundef 2, i32 noundef 3)
    ret i32 %2
}

```

7. `i32` means a 32-bit integer. `%0` and `%1` are the two parameters.
8. `alloca` reserves space on the stack. At `-O0` the compiler dutifully stores both parameters to the stack and loads them straight back. That is pointless, and the optimizer will delete it.
9. `add nsw` means add with “no signed wrap”: the compiler is allowed to assume signed overflow never happens. Hold on to that; it comes back in 16.7.

■ 16.6.4 PLAIN – what is really happening inside

1. **Static Single Assignment**, or SSA, means every name is written exactly once in the whole function.
2. If your source assigns `x` three times, the IR has `x_1`, `x_2` and `x_3`.
3. Why bother? Because now, if you see a use of `x_2`, there is exactly one place it could have come from. No searching. No guessing.
4. That single property makes constant propagation, dead code removal and value numbering enormously simpler and faster.
5. Branches create a problem: after an `if`, which version of `x` is live? SSA solves it with a **phi node**, written `PHI`, which says “take the version from whichever block we came from”.
6. Here is a real phi from a loop, for `(i = 0; i < n; i++) s += i;`, dumped by GCC with `-fdump-tree-ssa`:

```

<bb 4> :
# s_1 = PHI <s_3(2), s_8(3)>
# i_2 = PHI <i_4(2), i_9(3)>
if (i_2 < n_5(D))
    goto <bb 3>;
else
    goto <bb 5>;

```

7. Read $s_1 = \text{Phi} \langle s_3(2), s_8(3) \rangle$ as: if we arrived from block 2, s_1 is s_3 ; if from block 3, it is s_8 .
8. Phi nodes are not real instructions. Late in the back end they are removed by inserting copies into the predecessor blocks.

■ 16.6.5 TECHNICAL – the engineer’s version

1. SSA was formalized by Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman and Kenneth Zadeck in *Efficiently computing static single assignment form and the control dependence graph*, ACM TOPLAS, October 1991.
2. Placing phi nodes minimally requires **dominance frontiers**, computed by the Lengauer-Tarjan dominator algorithm of 1979 or Cooper, Harvey and Kennedy’s simpler 2001 method.
3. GCC uses two IRs in sequence. GIMPLE is a three-address, SSA-capable, tree-ish IR added in GCC 4.0, 2005. RTL, Register Transfer Language, is the older low-level IR used by the back end.
4. LLVM uses a single strongly typed SSA IR with three equivalent forms: an in-memory form, a human-readable .ll text form, and a compact .bc bitcode form. llvm-as and llvm-dis convert between text and bitcode.
5. LLVM IR is not a portable virtual machine. It embeds the target data layout and pointer sizes, visible in our dump as `target triple = "x86_64-pc-linux-gnu"`. Shipping .bc and expecting it to run anywhere is a common misunderstanding.
6. The front-end / middle-end / back-end split is what makes the ecosystem work. Clang, Rust, Swift, Julia, Zig and flang all emit LLVM IR. LLVM 18 registers over 30 target backends, from x86 and AArch64 to RISC-V, WebAssembly, NVPTX and AVR.
7. Other IR designs exist. Java bytecode and CPython bytecode are stack machines, not three-address. WebAssembly is a structured stack machine. Cranelift, used by Wasmtime, uses its own SSA IR built for compile speed rather than peak output quality.

IR	Owner	Style
GIMPLE	GCC	3-address, SSA
RTL	GCC back end	low-level, registers
LLVM IR	LLVM	typed SSA
Java bytecode	JVM	stack machine
CPython bytecode	CPython	stack machine
Wasm	W3C	structured stack

■ 16.6.6 WORDS – remember these

Intermediate representation — the compiler’s own simple language — a program form between source and machine code, target and language neutral in the middle end. Three-address code — one operation, one result, two inputs per line — a linearized IR form of the shape $t1 = a \text{ op } b$. SSA — every name written exactly once — static single assignment form, with phi functions merging values at control-flow joins. Basic block — a straight run with no branches inside — a maximal instruction sequence with a single entry and a single exit. Phi node — “take whichever value we came in with” — a pseudo-instruction at a join point selecting a value by predecessor block.

16.7 Optimization

■ 16.7.1 PLAIN – in simple words

1. The optimizer rewrites your program so it does less work but produces the same visible result.
2. It is allowed to change anything you cannot observe. It is not allowed to change what you can observe.
3. If the answer is always the same number, compute it now and store the number.
4. If a value never changes, replace the name with the value.
5. If a result is never used, delete the code that made it.
6. If the same sum is computed twice, compute it once.
7. If a function is tiny, paste its body into the caller instead of calling it.
8. Do all of these repeatedly, because each one exposes new chances for the others.

■ 16.7.2 PLAIN – a picture in your head

1. Imagine editing a set of instructions for making tea for a guest.
2. Step 3 says “boil water”. Step 9 says “boil water” again. Delete step 9 and reuse the first pot. That is common subexpression elimination.
3. Step 5 says “count the sugar packets in the drawer” every time round the stirring loop, but nobody adds packets. Move it outside the loop. That is loop-invariant code motion.
4. Step 7 says “fetch a lemon” and no later step uses the lemon. Delete step 7. That is dead code elimination.
5. Step 2 says “look up the boiling point of water in the book”. You know it is 100. Write 100. That is constant folding.
6. Step 8 says “consult the separate one-line card called stir”. Just write “stir” here. That is inlining.

Where this comparison breaks: you may delete “fetch a lemon” only if fetching lemons has no other effect. In a real program, a “useless” call might print something, write a file or set a flag another thread reads. Deciding whether a step is observable is the hard part, and it is most of what an optimizer does.

■ 16.7.3 PLAIN – a worked example

This is the heart of the chapter. Compile our unchanged file two ways.

`gcc -O0 -S add.c`, the whole `add` function, with directives removed:

```
add:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    ret

main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $3, %esi
    movl    $2, %edi
    call    add
```

```

    popq    %rbp
    ret

```

gcc -O2 -S add.c, the same file:

```

add:
    endbr64
    leal    (%rdi,%rsi), %eax
    ret

main:
    endbr64
    movl    $5, %eax
    ret

```

1. Seventeen instructions became five.
2. Why is add now one instruction? At -O0 the compiler stores both arguments to the stack and reloads them, because -O0 keeps every variable in memory for the debugger. At -O2 that is deleted.
3. `leal (%rdi,%rsi), %eax` computes `rdi + rsi` and puts it in `eax`. `lea` means load effective address: it does the address arithmetic but does not touch memory. It is used here as a free three-operand add.
4. Why is main now `movl $5, %eax`? Three optimizations in sequence.
 1. **Inlining**: add is tiny, so its body is pasted into main, giving `return 2 + 3;`.
 2. **Constant folding**: `2 + 3` is computed at compile time, giving `return 5;`.
 3. **Dead code elimination**: nothing else in main is needed, so the frame setup, the call, and the stack work all disappear.
5. add still exists as a separate function because it has external linkage: another file might call it. Mark it static and the whole function vanishes from the output.
6. The program still returns 5. `./a.out; echo $?` prints 5. That is the only thing that had to stay true.

Other optimizations, each verified on this machine with GCC 13.3 at -O2:

Source	Output at -O2
<code>3 * 4 + 10 / 2</code>	<code>movl \$17, %eax</code>
<code>x * 8</code>	<code>leal 0(,%rdi,8), %eax</code>
<code>int u = a*999; return a+1;</code>	<code>leal 1(%rdi), %eax</code>
<code>a*b+1 and a*b+2 summed</code>	one <code>imull</code> , then <code>leal</code>

1. Row one is constant folding: the whole expression is a compile-time constant.
2. Row two is **strength reduction**: multiply by 8 became a scaled address computation, which is cheaper than a general multiply.
3. Row three is dead code elimination: the multiply by 999 is gone.
4. Row four is common subexpression elimination: `a*b` was written twice and is computed once.

■ 16.7.4 PLAIN – what is really happening inside

1. **Loop unrolling** copies a loop body several times so the loop test runs less often. Our recursive `tail(n, acc)` function was turned by GCC into a loop and then unrolled by a factor of two.
2. **Tail call elimination** is what made that possible. When the last thing a function does is call something and return its result, the call can reuse the current stack frame. A self tail call then becomes a jump, that is, a loop.

3. Even across separate files, GCC at -O2 turned `call add; ret` into a plain `jmp add`. That is a tail call: `add`'s own `ret` returns straight to `main`'s caller.
4. **Vectorization** does several elements at once. For `for (i=0;i<n;i++) out[i] = in[i] * 2.0f;` GCC at -O3 emitted `movups` and `addps`. `addps` is "add packed single", four 32-bit floats in one instruction, using the 128-bit SSE registers.
5. **Register allocation** decides which values live in registers. x86-64 has 16 general purpose registers, a few reserved, so roughly 12 to 14 are usable. Values that do not fit must be **spilled** to the stack.
6. Chaitin's method models this as **graph colouring**. Each value is a node, and two nodes are joined if they are live at the same moment. Colouring with k colours, k being the register count, is exactly a valid allocation.
7. Graph colouring is NP-complete in general, so compilers use heuristics: repeatedly remove any node with fewer than k neighbours, and if none exists, pick a node to spill and try again.

■ 16.7.5 TECHNICAL – the engineer's version

1. The legal limit on all of this is the **as-if rule**. An implementation may do anything, provided observable behaviour matches the abstract machine. The C standard defines that as volatile accesses, input and output, and the state at termination.
2. GCC optimization levels, as documented for GCC 13: -O0 none, -O1 about 47 passes, -O2 most non-size-increasing passes, -O3 adds vectorization and heavier inlining, -Os optimizes for size, -Ofast is -O3 plus -ffast-math, -Og keeps debugging usable.
3. **Floating point is not associative**. IEEE 754 arithmetic rounds each operation, so $(a+b)+c$ and $a+(b+c)$ can differ. Real run on this machine with $a = 1e20f$, $b = -1e20f$, $c = 1.0f$:

```
(a+b)+c = 1
a+(b+c) = 0
```

4. That is why compilers must not reassociate floating point by default. -ffast-math grants permission to reassociate, to assume no NaN or infinity, and to flush denormals. On this machine, adding `0.1f` ten times prints `1.00000012`, not `1`.
5. **Undefined behaviour** lets the compiler assume the impossible never happens. Signed integer overflow is undefined in C. Real GCC 13.3 output at -O2:

```
int check(int x) { return x + 1 > x; }
check:
    movl    $1, %eax          # always true, folded
    ret

unsigned check_u(unsigned x) { return x + 1 > x; }
check_u:
    xorl    %eax, %eax        # real comparison kept
    cmpl    $-1, %edi
    setne   %al
    ret
```

6. The signed version was folded to a constant 1. The unsigned version was not, because unsigned overflow is defined to wrap, so $x + 1 > x$ really is false when x is `UINT_MAX`.
7. Compiling the signed version with -fwrapv restores the comparison, against 2147483647. -fwrapv is a GCC and Clang extension defining signed overflow as wrapping. It is an implementation choice, not the C standard.
8. Undefined behaviour also erases null checks after a dereference, deletes infinite loops with no side effects, and assumes strict aliasing. Tools: -fsanitize=undefined, -fsanitize=address, and Clang's -Rpass=inline to see what was inlined.

Optimization	What it removes
Constant folding	Compile-time arithmetic
Constant propagation	Reads of known values
Dead code elimination	Unused computations
Common subexpr elim	Repeated identical work
Strength reduction	Expensive operators
Loop-invariant motion	Repeated loop work
Inlining	Call overhead
Tail call elimination	Stack frame growth

■ 16.7.6 WORDS – remember these

Constant folding — do the sums now, not later — evaluating constant expressions at compile time. Inlining — paste the function body in — replacing a call with the callee’s body, enabling further optimization at the call site. Dead code elimination — delete what nobody uses — removing instructions whose results are not live and have no side effects. Spilling — running out of registers and using memory — storing a live value to the stack frame because the allocator could not colour it. As-if rule — you may change anything nobody can see — the standard’s permission to transform a program while preserving observable behaviour. Undefined behaviour — the standard gives no rules for this — a construct for which the standard imposes no requirements, letting the optimizer assume it never occurs.

16.8 Code generation

■ 16.8.1 PLAIN – in simple words

1. Now the compiler must write instructions for one actual chip.
2. Three jobs happen here, tangled together.
3. **Instruction selection**: pick which real instructions do what the IR said. The IR said “add”. The chip may offer add, lea or inc.
4. **Instruction scheduling**: pick the order. Some instructions wait for results; put unrelated work in the gap.
5. **Register allocation**: pick which of the chip’s few fast slots holds which value, and put the rest in memory.
6. There is also a rule book that everybody must obey: where arguments go, where the answer comes back, who is allowed to break which register.
7. That rule book is the **calling convention**. It is what lets a function compiled today call a library compiled ten years ago.

■ 16.8.2 PLAIN – a picture in your head

1. Think of handing work to a colleague through a hatch with numbered trays.
2. Everyone in the building has agreed: the first thing you are passing goes in tray 1, the second in tray 2, and the answer comes back in tray 0.
3. Some trays are yours to scribble on; the colleague may empty them. Other trays they must hand back exactly as they found them.
4. If you both follow the agreement, neither of you needs to know anything about how the other works inside.
5. If you disagree about which tray is which, everything breaks in ways that look like random corruption.

Where this comparison breaks: real conventions also cover stack alignment, variable argument lists, structures too big for a register, floating point in separate registers, and where the return address lives. And there is no single agreement: Linux, Windows and ARM each chose different trays.

■ 16.8.3 PLAIN - a worked example

Here is our `-O0 add` function again, and this time every line is annotated.

```
add:
    endbr64                # landing pad for indirect jumps (CET)
    pushq   %rbp           # save caller's frame pointer
    movq    %rsp, %rbp     # this function's frame starts here
    movl    %edi, -4(%rbp)  # store arg 1 (a) into the frame
    movl    %esi, -8(%rbp)  # store arg 2 (b) into the frame
    movl    -4(%rbp), %edx  # load a back into edx
    movl    -8(%rbp), %eax  # load b back into eax
    addl    %edx, %eax     # eax = a + b
    popq    %rbp           # restore caller's frame pointer
    ret                  # jump back to the return address
```

1. `%edi` and `%esi` are the low 32 bits of `rdi` and `rsi`. On Linux those are the first two integer argument registers.
2. `%eax` holds the return value. The result of the addition is already there, so no extra move is needed.
3. The two stores and two loads are pure `-O0` overhead, as explained in 16.7.
4. `pushq %rbp` and `movq %rsp, %rbp` are the **prologue**. `popq %rbp` and `ret` are the **epilogue**.
5. `endbr64` is not part of the calling convention. It is Intel Control-flow Enforcement Technology, enabled by default on Ubuntu with `-fcf-protection=full`. It marks a legal target for an indirect jump.

And main:

```
main:
    endbr64
    pushq   %rbp           # prologue
    movq    %rsp, %rbp
    movl    $3, %esi       # second argument = 3
    movl    $2, %edi       # first argument = 2
    call    add            # push return address, jump to add
    popq    %rbp           # epilogue; eax already holds 5
    ret                  # return to the C runtime
```

1. Arguments are loaded second-first here. That order is the compiler's choice, not a rule. Only which register holds which argument is fixed.
2. `call` pushes the address of the next instruction onto the stack and jumps.
3. Nothing copies the result. `add` left it in `eax` and `main` returns it from `eax`.

The stack while `add` is running:

```
higher addresses
+-----+
| main's frame          |
+-----+
| return address into main | <- pushed by call
+-----+
| saved rbp              | <- pushed by prologue
+-----+ <- rbp now points here
| -4(%rbp) copy of a     |
```

```

| -8(%rbp)  copy of b      |
+-----+ <- rsp
| 128-byte red zone      | usable without moving rsp
+-----+
lower addresses

```

■ 16.8.4 PLAIN – what is really happening inside

1. Instruction selection is usually done by **tree pattern matching**. The IR is covered with the cheapest set of instruction patterns that fits.
2. The cost model matters. `imul` by 8 costs about 3 cycles on many x86 cores; `lea` with a scale of 8 costs 1. That is why GCC chose `lea`.
3. Instruction scheduling matters most on machines that issue several instructions per cycle. Modern out-of-order x86 chips reorder anyway, so scheduling is worth less there than on an in-order core.
4. The same source compiled for AArch64 with `clang --target=aarch64-linux-gnu -O2` gives:

```

add:
    add    w0, w1, w0
    ret
main:
    mov    w0, #5
    ret

```

5. Same structure, different names. `w0` and `w1` are the low 32 bits of `x0` and `x1`, the first two AArch64 argument registers, and `x0` is also the return register.
6. The same source targeting Windows with `clang --target=x86_64-pc-windows-msvc -O0` stores from `%ecx` and `%edx` instead of `%edi` and `%esi`, because Windows chose different argument registers.
7. One source file, three sets of rules, all correct.

■ 16.8.5 TECHNICAL – the engineer’s version

1. The three conventions side by side. All three are written specifications, not conventions in the loose sense: System V AMD64 ABI, Microsoft x64 calling convention, and Arm’s AAPCS64.

Role	SysV AMD64	Win x64	AArch64
Int arg 1	<code>rdi</code>	<code>rcx</code>	<code>x0</code>
Int arg 2	<code>rsi</code>	<code>rdx</code>	<code>x1</code>
Int arg 3	<code>rdx</code>	<code>r8</code>	<code>x2</code>
Int arg 4	<code>rcx</code>	<code>r9</code>	<code>x3</code>
Int args 5–6	<code>r8, r9</code>	stack	<code>x4, x5</code>
Float args	<code>xmm0–xmm7</code>	<code>xmm0–xmm3</code>	<code>v0–v7</code>
Return	<code>rax (rdx:rax)</code>	<code>rax</code>	<code>x0 (x1 too)</code>
Shadow space	none	32 bytes	none
Red zone	128 bytes	none	none
Frame pointer	<code>rbp</code>	<code>rbp</code>	<code>x29</code>
Return address	on stack	on stack	<code>x30 (lr)</code>

2. **Callee-saved** registers on System V AMD64 are `rbx`, `rbp`, `r12`, `r13`, `r14`, `r15`. A function that uses them must restore them. Everything else is **caller-saved**: if you need it after a call, save it yourself.
3. Stack alignment: System V AMD64 requires `rsp + 8` to be 16-byte aligned at a call, so `rsp` is 16-byte aligned on entry after the return address is pushed. AAPCS64 requires `sp` 16-byte aligned at all public interfaces.

4. The **red zone** is 128 bytes below `rsp` that a leaf function may use without adjusting `rsp`. Signal handlers must not clobber it. Kernel code is compiled with `-mno-red-zone` for exactly this reason.
5. Structures larger than 16 bytes are passed in memory on System V AMD64; smaller ones are classified field by field into INTEGER and SSE classes. The classification algorithm in the ABI document is one of the fiddliest parts of the whole convention.
6. Register allocation in production: GCC uses IRA, an integrated regional allocator, plus LRA for reload, replacing the old reload pass in GCC 4.8,
2013. LLVM's default is a greedy allocator based on priority and live range splitting, not classical Chaitin colouring. Linear scan allocation, from Massimiliano Poletto and Vivek Sarkar in 1999, is preferred by JITs because it is far faster to run.
7. Useful commands: `gcc -S -masm=intel` for Intel syntax, `objdump -d --no-show-raw-insn, perf annotate` to see which instruction is hot, and `llvm-mca` to model instruction throughput on a named CPU.

■ 16.8.6 WORDS - remember these

Calling convention — the shared rules about who puts what where — the ABI specification of argument registers, return registers, stack alignment and register preservation. Prologue — the few instructions that set up a function — saving the frame pointer and reserving stack space on entry. Epilogue — the few instructions that clean up — restoring saved registers and returning. Stack frame — one function's private scratch area — the region between the frame pointer and the stack pointer holding locals, spills and saved registers. Callee-saved register — one you must hand back unchanged — a register the ABI requires the called function to preserve across the call. Instruction selection — choosing which real instructions to use — covering the IR with target patterns under a cost model.

16.9 The assembler

■ 16.9.1 PLAIN - in simple words

1. The compiler's output is still text. `addl %edx, %eax` is nine characters.
2. The **assembler** turns each such line into the bytes the chip actually reads.
3. It is a much simpler program than the compiler. Mostly it is a lookup table plus arithmetic on addresses.
4. It also collects your program into named piles called **sections**.
5. Code goes in one pile. Numbers you gave starting values go in another. Numbers that start at zero go in a third that takes no file space at all.
6. It records every name you defined and every name you used but did not define.
7. Where an address is not yet known, it writes zeros and leaves a note. Those notes are called **relocations**.
8. Its output is an **object file**, ending `.o` on Unix and `.obj` on Windows.

■ 16.9.2 PLAIN - a picture in your head

1. Think of typesetting a book chapter that refers to other chapters.
2. You can set every word of your own chapter into metal type immediately.
3. But "see chapter 9, page ..." cannot be set, because you do not know the page number yet. Other people are still setting chapter 9.
4. So you leave a gap of the right size and write a note in the margin: "fill this gap with the start page of chapter 9".
5. You hand the printer your typeset pages plus your list of margin notes.

6. Later, somebody assembling the whole book fills every gap.

Where this comparison breaks: some relocations are not simple substitutions. They store a difference between two addresses, or an offset from the current position, or an index into a table filled at program start. The note in the margin has a type, and there are dozens of types.

■ 16.9.3 PLAIN – a worked example

Run `gcc -c add.c -o add.o`, then `objdump -d add.o`:

```
0000000000000000 <add>:
  0: f3 0f 1e fa    endbr64
  4: 55             push    %rbp
  5: 48 89 e5       mov     %rsp,%rbp
  8: 89 7d fc       mov     %edi,-0x4(%rbp)
 b: 89 75 f8       mov     %esi,-0x8(%rbp)
 e: 8b 55 fc       mov     -0x4(%rbp),%edx
11: 8b 45 f8       mov     -0x8(%rbp),%eax
14: 01 d0         add     %edx,%eax
16: 5d             pop     %rbp
17: c3             ret

0000000000000018 <main>:
18: f3 0f 1e fa    endbr64
1c: 55             push    %rbp
1d: 48 89 e5       mov     %rsp,%rbp
20: be 03 00 00 00 mov     $0x3,%esi
25: bf 02 00 00 00 mov     $0x2,%edi
2a: e8 00 00 00 00 call    2f <main+0x17>
2f: 5d             pop     %rbp
30: c3             ret
```

1. The left column is the offset inside the section. The middle column is the actual bytes. The right column is the text form.
2. `01 d0` is two bytes. That is the entire `add %edx,%eax` instruction.
3. `c3` is one byte: `ret`.
4. Look at offset `2a`. The `call` is `e8` followed by `00 00 00 00`. The assembler did not know where `add` would end up, so it wrote four zero bytes.
5. `objdump` guesses the target as `2f`, which is simply “the next instruction”, because the offset is zero. That is not the real target. It is a hole.
6. Now the note in the margin, from `readelf -r add.o`:

```
Relocation section '.rel.text':
  Offset          Type           Sym. Name + Addend
  00000000000002b  R_X86_64_PLT32    add - 4
```

7. Read it as: at byte offset `0x2b`, which is the four zero bytes, write the distance from here to `add`, minus 4.
8. `nm add.o` shows the symbols: `0000000000000000 T add` and `0000000000000018 T main`. `T` means defined in the text section.

■ 16.9.4 PLAIN – what is really happening inside

1. The assembler makes two passes. Pass one measures every instruction and works out where each label lands. Pass two writes the bytes, using the label addresses from pass one.
2. Two passes are needed because of forward references: a jump to a label further down the file cannot be encoded until that label's position is known.
3. Sections keep unlike things apart so the operating system can protect them differently:
 1. `.text` is machine code. It ends up readable and executable, not writable.
 2. `.rodata` is constants, such as string literals. Readable, not writable.
 3. `.data` is variables with a non-zero starting value. Readable and writable, and its contents occupy space in the file.
 4. `.bss` is variables starting at zero. Readable and writable, and it takes no file space at all: only a size is recorded.
4. `readelf -S add.o` on our file shows `.text` at 0x31 bytes and both `.data` and `.bss` at zero bytes, because our program has no global variables.
5. That is why a program declaring a 10 MB zeroed array does not have a 10 MB executable. The `.bss` entry says “10 MB of zeros” in a few bytes.

■ 16.9.5 TECHNICAL – the engineer's version

1. The object file format on Linux and most Unix systems is **ELF**, Executable and Linkable Format, introduced with UNIX System V Release 4 around 1988 and adopted by Linux in the mid-1990s, replacing a.out. macOS uses Mach-O, from NeXTSTEP. Windows uses PE/COFF.
2. Our `add.o` contains 13 section headers. The important ones are `.text`, `.rela.text`, `.data`, `.bss`, `.symtab`, `.strtab`, `.shstrtab` and `.eh_frame`.
3. `.symtab` is the symbol table, `.strtab` holds the symbol name strings, and `.shstrtab` holds the section name strings. Names are stored once and referenced by offset.
4. `.eh_frame` holds unwind information generated from the `.cfi_*` directives in the assembly. It is what lets a debugger produce a backtrace and what lets C++ exceptions unwind the stack. It is generated even for C.
5. Common x86-64 relocation types: `R_X86_64_64` absolute 64-bit, `R_X86_64_PC32` 32-bit program-counter relative, `R_X86_64_PLT32` call through the procedure linkage table, `R_X86_64_GOTPCREL` load through the global offset table. `R_X86_64_RELATIVE` is applied at load time for position-independent executables.
6. Symbol binding is `LOCAL`, `GLOBAL` or `WEAK`. `nm` prints `T` for a global text symbol, `t` for local, `U` for undefined, `B` for `.bss`, `D` for `.data`, `W` for weak.
7. Our whole object file is 169 bytes of text, data and bss combined, per `size add.o`. The linked executable reports 1302 bytes of text, because the C runtime start-up code has been added.

Command	What it shows
<code>objdump -d f.o</code>	Disassembled code
<code>objdump -h f.o</code>	Section headers, sizes
<code>readelf -S f.o</code>	Full section table
<code>readelf -r f.o</code>	Relocation entries
<code>readelf -s f.o</code>	Symbol table
<code>nm f.o</code>	Symbols, short form
<code>size f.o</code>	text, data, bss totals

■ 16.9.6 WORDS – remember these

Assembler — turns instruction text into bytes — a two-pass translator from assembly mnemonics to encoded machine instructions plus metadata. Object file — a half-finished program with holes — a relocatable ELF, Mach-O or COFF file containing sections, symbols and relocations. Section — one named pile of similar bytes — a named region such as `.text`, `.data`, `.bss` or `.rodata` with its own permissions. Relocation — a note saying “fill this hole later” — a record naming an offset, a symbol and a formula for computing the value to patch in. Symbol — a name the outside world can see — a named entry binding an identifier to a section and offset, with a binding and a visibility.

16.10 The linker

■ 16.10.1 PLAIN – in simple words

1. Real programs are many files. Each is compiled separately into an object file full of holes.
2. The **linker** puts them all together and fills every hole.
3. It stacks all the `.text` sections into one `.text`, all the `.data` into one `.data`, and so on.
4. Now every function has a final address, so every hole can be filled with a real number.
5. Then it checks that every name that was used somewhere is defined somewhere.
6. If a name is used and never defined, it stops and says **undefined reference**. That is the most famous error message in C.
7. It also pulls in code from **libraries**: bundles of ready-made object files.

■ 16.10.2 PLAIN – a picture in your head

1. Think of assembling one book from chapters written by different authors.
2. Each author numbered their pages from 1. You must renumber so the whole book runs from 1 to 400.
3. Every “see page 12” inside chapter 3 must be adjusted by however much chapter 3 moved.
4. Every “see the chapter on rivers” must be turned into a real page number by looking up which chapter that is.
5. If somebody wrote “see the chapter on volcanoes” and no such chapter exists, the book cannot be finished. That is undefined reference.
6. A **static library** is a shelf of spare chapters. You copy in only the ones somebody referred to.
7. A **dynamic library** is a separate book that stays separate. Your book says “look this up in the other book”, and the reader must own that book.

Where this comparison breaks: the linker does more than renumber. It merges duplicate definitions of inline functions and templates, discards unused sections, can reorder functions for cache locality, and with link-time optimization can re-run the whole optimizer across chapter boundaries.

■ 16.10.3 PLAIN – a worked example

Split our example across two files: `main2.c` calling `add`, and `addonly.c` defining it. Compile `main2.c` alone and try to link:

```
$ gcc main2.c -o prog2
/usr/bin/ld: /tmp/cckot5dw.o: in function `main':
main2.c:(.text+0x13): undefined reference to `add'
collect2: error: ld returned 1 exit status
```

1. The compiler was perfectly happy: the header declared `add`, so the call type checked.
2. The linker was not, because no object file defines `add`.

3. Now link both: `gcc main2.o addonly.o -o prog2`. It works, and returns 5.

Look at what relocation actually did. In `add.o` the call was `e8 00 00 00 00`. In the linked executable, `objdump -d`:

```
00000000000001129 <add>:
    1129: f3 0f 1e fa    endbr64
    ...
00000000000001141 <main>:
    1141: f3 0f 1e fa    endbr64
    ...
    1153: e8 d1 ff ff ff    call    1129 <add>
```

4. The four zero bytes became `d1 ff ff ff`, which as a signed 32-bit little-endian number is -47.
5. The instruction after the call starts at `0x1158`. And `0x1158 - 47 = 0x1129`, which is exactly where `add` ended up.
6. That is the whole of relocation, in one number.

Now libraries. Real commands and real sizes on this machine:

```
$ ar rcs libadd.a addonly.o          # static library
$ gcc main2.c -L. -ladd -o prog_static

$ gcc -fPIC -shared addonly.c -o libadd.so # shared library
$ gcc main2.c -L. -ladd -o prog_dyn -Wl,-rpath,'$ORIGIN'
```

File	Size in bytes
libadd.a	1372
libadd.so	15112
prog_static	15840
prog_dyn	15944

Both programs return 5. In `prog_static` the `add` code is inside the executable. In `prog_dyn` it is not, and `ldd prog_dyn` lists `libadd.so => /tmp/kb/./libadd.so`.

■ 16.10.4 PLAIN – what is really happening inside

1. A static library `.a` file is just an archive of object files, made by `ar`. `ar -t libadd.a` prints `addonly.o`.
2. The linker treats an archive specially: it does not include everything. It scans the archive and pulls in only the members that resolve a symbol still undefined at that moment.
3. That is why **link order matters**. The linker walks the command line left to right, keeping a set of currently undefined symbols.
4. Real proof on this machine. `gcc -L. -ladd main2.c` fails with “undefined reference to `add`”, while `gcc main2.c -L. -ladd` succeeds. Same files, different order.
5. The reason: when the library was scanned, nothing needed `add` yet, so no member was pulled in. `main2.c` came later and asked for a symbol nobody was still looking for. Put objects first and libraries last.
6. In **C++** the situation is harder, because C++ allows several functions with the same name. The linker only handles plain names, so the compiler encodes the full signature into the name. That is **name mangling**.
7. Real output from `nm` on a C++ file, alongside `nm -C` which demangles:

```

_Z3addii      -> add(int, int)
_Z3adddd      -> add(double, double)
_ZN2kb1T3addEii -> kb::T::add(int, int)
add_c         -> add_c

```

8. Read `_Z3addii` as: mangled name, 3-letter name `add`, arguments `i, i`. The last one is extern `"C"`, which turns mangling off, which is exactly how C and C++ call each other.

■ 16.10.5 TECHNICAL – the engineer’s version

1. Linker phases: read inputs, resolve symbols, assign section addresses via a layout script, apply relocations, write output, optionally strip.
2. Library naming, all conventions rather than standards: `.a` static and `.so` shared on Linux, `.lib` static and `.dll` dynamic on Windows, `.a` static and `.dylib` dynamic on macOS. Windows also uses an import `.lib` that describes a `.dll` without containing its code.
3. **Symbol visibility** controls what a shared library exports. GCC and Clang accept `-fvisibility=hidden` plus `__attribute__((visibility("default")))` on the few symbols you mean to export. Smaller export tables mean faster loading and better optimization. Windows uses `__declspec(dllexport)` and `__declspec(dllimport)`.
4. Shared library code must be position independent, built with `-fPIC`, because it can be mapped at different addresses in different processes. Most Linux distributions have built executables as position independent (PIE) by default since around 2017.
5. **Link-time optimization** keeps the IR in the object file instead of only machine code, so the optimizer can run across file boundaries at link time. Real measurement on this machine, with `add` in a separate file from `main`:
 1. `gcc -O2 main2.c addonly.c` gave `main` a tail call: `mov $3,%esi; mov $2,%edi; jmp 1150 <add>`.
 2. `gcc -O2 -flto main2.c addonly.c` gave `main` exactly `mov $0x5,%eax; ret`.
6. That is the same inline-then-fold sequence from 16.7, now working across translation units, which plain `-O2` cannot do. GCC’s LTO arrived in GCC 4.5,
2010. LLVM offers full LTO and ThinLTO, the latter published by Teresa Johnson and colleagues in 2017 for much better build parallelism.
7. Linker choices as of 2026: GNU `ld`, the original; `gold`, 2008, ELF only and dropped from recent `binutils`; LLVM `lld`, the default in many toolchains; and `mold` by Rui Ueyama, 1.0 in 2022, the fastest widely used linker for large C++ builds.
8. Weak symbols let a definition be overridden. `__attribute__((weak))` is how a library provides a default that a program may replace.

Symptom	Usual cause
undefined reference to X	Object or library missing
undefined reference in C++	Missing extern <code>"C"</code> , mangling
multiple definition of X	Definition in a header
library before object fails	Wrong command-line order
runs, then “cannot open .so”	Library not on the run path

■ 16.10.6 WORDS – remember these

Linker — the tool that glues object files into a program — the program that merges sections, resolves symbols and applies relocations. Static library — a shelf of object files you copy from — an `ar` archive whose members are pulled in only to satisfy undefined symbols. Shared library — a separate file loaded at run time — a `.so`, `.dll` or `.dylib` mapped into the process by the dynamic linker. Name mangling — encoding the full signature into the symbol name — the compiler scheme that gives overloaded and namespaced C++ entities distinct linker names. Link-time optimization — optimizing across file boundaries — keeping IR in object files so the optimizer runs at link time. Undefined reference — somebody used a name nobody defined — an unresolved symbol remaining after all inputs have been scanned.

16.11 The loader and dynamic linking

■ 16.11.1 PLAIN – in simple words

1. A finished executable is a file on disk. It is not yet a running program.
2. When you type its name and press Enter, the operating system creates a process and puts the file's contents into that process's memory.
3. The part of the operating system that does this is the **loader**.
4. It does not copy the whole file. It maps it, meaning it says “this range of memory corresponds to this part of this file”, and lets the pages arrive when they are first touched.
5. If the program uses shared libraries, they are not inside it, so somebody must find them, load them too, and connect the calls.
6. That job belongs to a small program called the **dynamic linker**, which the loader starts first.
7. Only after all that does your main actually run.

■ 16.11.2 PLAIN – a picture in your head

1. Think of a play. The script is the executable file. The performance is the process.
2. The stage manager, the loader, sets out the scenery from the script.
3. The script says “at this point, the orchestra plays the theme”. The orchestra is a shared library and is not in your script.
4. So before the curtain, an assistant goes and finds the orchestra, checks they have the right music, and writes their seat numbers into a small card at the side of the stage.
5. Every time the script says “orchestra”, an actor glances at the card to see where to look.
6. If the assistant is lazy, the card starts blank, and the first time the actor glances at it, the assistant is fetched to fill in that one entry.

Where this comparison breaks: the card, the global offset table, is per process, not per library, and it is normally made read-only after start-up for security. And the assistant is itself a shared library, loaded by a special path that does not need an assistant.

■ 16.11.3 PLAIN – a worked example

`readelf -l` on our executable shows what the loader is told to do:

```
Elf file type is DYN (Position-Independent Executable)
Entry point 0x1040
  Type      VirtAddr    FileSiz  MemSiz  Flg  Align
  INTERP    0x00000318    0x00001c 0x00001c R    0x1
    [Requesting interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD      0x00000000    0x0005f0 0x0005f0 R    0x1000
```

```
LOAD 0x00001000 0x000169 0x000169 R E 0x1000
LOAD 0x00002000 0x0000ec 0x0000ec R 0x1000
LOAD 0x00003df0 0x000220 0x000228 RW 0x1000
```

1. Four LOAD entries, four memory regions with different permissions.
2. The second is R E, readable and executable: that is your code.
3. The last is RW and its MemSiz (0x228) is larger than its FileSiz (0x220). The extra 8 bytes are .bss, zero-filled at load and stored in no file bytes at all.
4. INTERP names the dynamic linker. The kernel loads that, not your program.

Now watch it happen. `strace -e trace=execve,openat,mmap ./prog_dyn`, trimmed:

```
execve("./prog_dyn", ["/prog_dyn"], ...) = 0
openat("../glibc-hwcap/x86-64-v4/libadd.so") = -1 ENOENT
openat("../glibc-hwcap/x86-64-v3/libadd.so") = -1 ENOENT
openat("/tmp/kb/libadd.so", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL, 16400, PROT_READ, ...) = 0x7f88..
mmap(0x..1c, 4096, PROT_READ|PROT_EXEC, ...)
mmap(0x..1d, 4096, PROT_READ, ...)
mmap(0x..1e, 8192, PROT_READ|PROT_WRITE, ...)
openat("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
openat("/lib/x86_64-linux-gnu/libc.so.6", ...) = 3
+++ exited with 5 +++
```

5. It searched several CPU-feature-specific directories first, then found the library, then mapped it in four pieces with different permissions.
6. It consulted `/etc/ld.so.cache`, a prebuilt index of every library on the system, before touching the real directories, because scanning directories for every library on every program start would be slow.

■ 16.11.4 PLAIN – what is really happening inside

1. Calls into a shared library do not jump directly. They jump to a small local stub. Real disassembly of main in `prog_dyn`:

```
115b: e8 f0 fe ff ff  call 1050 <add@plt>

00000000000001050 <add@plt>:
1050: f3 0f 1e fa      endbr64
1054: ff 25 76 2f 00 00 jmp *0x2f76(%rip) # 3fd0
```

2. main calls a stub called `add@plt`. That stub jumps to whatever address is stored at `0x3fd0`.
3. The table of such addresses is the **global offset table**, the GOT. The table of stubs is the **procedure linkage table**, the PLT.
4. `readelf -r prog_dyn` shows the note that fills slot `0x3fd0`:

```
Relocation section '.rela.plt' contains 1 entry:
Offset      Type           Sym. Name
000000003fd0 R_X86_64_JUMP_SLO add + 0
```

5. **Lazy binding**: by default, the GOT slot initially points back into the PLT, which calls the dynamic linker to resolve the symbol, patch the slot, and jump on. Every later call goes straight through.
6. Setting `LD_BIND_NOW=1` resolves everything up front. On this machine, `LD_DEBUG=statistics LD_BIND_NOW=1 ./prog_dyn` reported 100 relocations and about 155,000 cycles of total dynamic loader start-up time.

7. Lazy binding trades a slightly slower first call for a faster start. Hardened builds turn it off, because a writable GOT is an attack target; `-z now` plus `-z relro` makes the GOT read-only after start-up.

■ 16.11.5 TECHNICAL – the engineer’s version

1. Search order for a shared library on glibc: `DT_RPATH` if there is no `DT_RUNPATH`, then `LD_LIBRARY_PATH`, then `DT_RUNPATH`, then `/etc/ld.so.cache`, then the default directories `/lib` and `/usr/lib` with their architecture subdirectories.
2. `DT_RPATH` is deprecated in favour of `DT_RUNPATH`, because `RUNPATH` is overridable by `LD_LIBRARY_PATH` and `RPATH` is not. Our example used `-Wl,-rpath,'$ORIGIN'` and `readelf -d` shows `RUNPATH Library runpath: [$ORIGIN]`. `$ORIGIN` expands to the directory of the executable, which is how self-contained application bundles work.
3. Setting `LD_LIBRARY_PATH` globally in a shell profile is a well-known source of confusing bugs, because it affects every program you start. Prefer `RUNPATH` baked into the binary.
4. `LD_PRELOAD` loads a library before all others, letting you interpose on any function. It is the mechanism behind many profilers and sanitizer shims. For safety, it is ignored for set-user-ID programs.
5. Versioned sonames are how Linux avoids most DLL hell. A library records a `SONAME` such as `libc.so.6`, and glibc additionally versions individual symbols, so `__libc_start_main@GLIBC_2.34` and an older version can coexist in one file. Our binary’s relocation table shows exactly that symbol.
6. **Dependency hell** is when two of your dependencies need incompatible versions of a third. **DLL hell** was the Windows form, where an installer overwrote a shared system32 DLL and broke other programs. Windows answered with side-by-side assemblies from Windows XP, 2001.
7. Static linking avoids all of it and costs disk space and the loss of shared library security updates. Go links statically by default; Rust links its own standard library statically but the system C library dynamically unless you choose a musl target.
8. This chapter has treated ELF, Mach-O and PE only as far as needed to explain linking and loading. Chapter 20 covers the file formats themselves in detail.

Tool	Purpose
<code>ldd prog</code>	List shared library deps
<code>readelf -d prog</code>	NEEDED, SONAME, RUNPATH
<code>LD_DEBUG=all ./prog</code>	Full dynamic linker trace
<code>LD_DEBUG=bindings</code>	Which symbol bound where
<code>ldconfig -p</code>	Contents of <code>ld.so.cache</code>
<code>otool -L prog</code>	Same as <code>ldd</code> , on macOS

■ 16.11.6 WORDS – remember these

Loader — the part of the OS that turns a file into a process — the `execve` path that maps `PT_LOAD` segments and transfers control to the entry point. Dynamic linker — the helper that finds and connects libraries — `ld.so`, named in the `PT_INTERP` segment, which maps dependencies and applies relocations. GOT — the little table of “where is it really” — the global offset table, an array of addresses patched at load or first use. PLT — the little stubs that read that table — the procedure linkage table, one trampoline per imported function. Lazy binding — only look it up the first time it is called — resolving PLT entries on first call rather than at start-up; disabled by `-z now`. `RUNPATH` — a search path baked into the program — the `DT_RUNPATH` dynamic entry, searched after `LD_LIBRARY_PATH`.

16.12 Interpreters and virtual machines

■ 16.12.1 PLAIN – in simple words

1. Compiling is not the only way to run a program.
2. An **interpreter** reads your program and does what it says, immediately, without ever producing machine code.
3. The simplest kind walks the syntax tree from 16.4. At a + node it evaluates the left child, evaluates the right child, and adds.
4. That is easy to write and slow to run, because it re-inspects the tree every single time round a loop.
5. So most real “interpreted” languages do something in between. They compile your program once into a simple made-up instruction set, then run those.
6. Those instructions are **bytecode**, and the program that runs them is a **virtual machine**.
7. Bytecode is not for any real chip. It is designed to be easy to produce and quick to run in a loop.

■ 16.12.2 PLAIN – a picture in your head

1. Imagine a recipe in a foreign language and a cook who does not read it.
2. Option one: a translator stands beside the cook and translates each line aloud, every time, including every time round a repeated step. That is a tree interpreter.
3. Option two: the translator writes out the whole recipe in simple numbered steps first, and the cook then follows the numbered steps. Translation happens once. That is bytecode.
4. Option three: the translator rewrites the recipe into this exact kitchen’s own house notation, tuned for these exact pans, before anyone starts. That is ahead-of-time compilation.

Where this comparison breaks: option two’s numbered steps are still not what the kitchen’s machines understand. Somebody, the virtual machine, is still reading each numbered step and deciding what to do. That reading is the cost that a JIT, covered in 16.13, removes.

■ 16.12.3 PLAIN – a worked example

Our example in Java, compiled with `javac` and disassembled with `javap -c`:

```
static int add(int, int);
  Code:
    0: iload_0
    1: iload_1
    2: iadd
    3: ireturn

public static void main(java.lang.String[]);
  Code:
    0: iconst_2
    1: iconst_3
    2: invokestatic #7    // Method add:(II)I
    5: invokestatic #13   // Method System.exit:(I)V
    8: return
```

1. This is a **stack machine**. `iload_0` pushes local variable 0. `iload_1` pushes local variable 1. `iadd` pops two and pushes their sum. `ireturn` pops and returns.
2. Every opcode is one byte, which is where the name bytecode comes from.
3. No registers are named. That is deliberate: the JVM does not know what chip it is on.
4. The first eight bytes of the `.class` file, from `od -An -tx1 -N 8`, are `ca fe ba be 00 00 00 41`. The first four are the magic number `0xCAFEBAFE`, chosen at Sun in the early 1990s. `0x41` is 65: Java 21.

And the same idea in CPython 3.11, from the `dis` module:

```
def add(a, b): return a + b
    LOAD_FAST      0 (a)
    LOAD_FAST      1 (b)
    BINARY_OP      0 (+)
    RETURN_VALUE

def main(): return add(2, 3)
    LOAD_GLOBAL    1 (NULL + add)
    LOAD_CONST     1 (2)
    LOAD_CONST     2 (3)
    PRECALL        2
    CALL           2
    RETURN_VALUE
```

5. Same shape: push, push, operate, return.
6. Python caches this in `__pycache__/m.cpython-311.pyc`, whose first four bytes here are `a7 0d 0d 0a`: a version magic, then a carriage return and newline chosen so that a text-mode file transfer corrupts the magic and the file is rejected rather than misread.
7. The huge difference is what the opcodes mean. `iadd` in Java adds two 32-bit integers, full stop. `BINARY_OP +` in Python must inspect both objects' types at run time and find the right method. That is most of the speed gap.

■ 16.12.4 PLAIN – what is really happening inside

1. A bytecode virtual machine is a loop: fetch the next opcode, jump to the code for it, do it, repeat.
2. The jump is usually a computed jump into a table of handler addresses. The cost per opcode is that jump plus the work.
3. The jump is hard for the CPU's branch predictor, because the next opcode is essentially unpredictable. That mispredict cost is the main tax of interpretation.
4. **Threaded code** reduces it. Instead of one jump at the top of the loop, each handler ends with its own jump to the next handler, giving the predictor more context. CPython uses computed gotos on GCC and Clang for this reason.
5. Bytecode is portable because it names no registers, no addresses and no instruction encodings. One `.class` file runs on x86, AArch64 and anything else with a JVM.
6. It also starts fast, because there is nothing to compile at start-up if the bytecode is already cached, which is exactly what `.pyc` files are for.

■ 16.12.5 TECHNICAL – the engineer's version

1. Java was released by Sun Microsystems in January 1996. The JVM specification defines the class file format, the instruction set, verification rules and the memory model. It is a written standard.
2. The JVM has about 200 defined opcodes out of 256, leaving room reserved for internal use. Class file major versions map to releases: 52 is Java 8, 55 is Java 11, 61 is Java 17, 65 is Java 21.
3. Bytecode verification happens at class load. The verifier proves type safety and stack discipline before any code runs. That is a real security property, and it is why JVM bytecode is not simply machine code in disguise.
4. CPython compiles to bytecode at import and caches it. `__pycache__` and the `.pyc` naming scheme were introduced in Python 3.2 by PEP 3147, 2010. The bytecode format is explicitly unstable between minor releases, which is why the file name includes `cpython-311`.

5. Python 3.11, October 2022, added a specializing adaptive interpreter, PEP 659, which rewrites hot opcodes into type-specialized versions at run time. The CPython team reported roughly 1.25 times faster on their benchmark suite versus 3.10.
6. Rough figures for an arithmetic loop, order of magnitude only, because the real ratio depends heavily on the workload. Treat these as approximate.
 1. A tree-walking interpreter: roughly 100 to 1000 times slower than C.
 2. A bytecode VM with no JIT: roughly 10 to 100 times slower.
 3. A good JIT: within about 1 to 3 times.

Trait	Compiled AOT	Bytecode VM	Tree walker
Start-up	Fastest	Medium	Fast
Peak speed	Fastest	Near AOT, if JIT	Slowest
Portability	Rebuild per CPU	One artifact	One artifact
Debug info	Needs symbols	Built in	Built in
Examples	C, Rust, Go	Java, C#, Python	Early Ruby, bash

■ 16.12.6 WORDS – remember these

Interpreter — a program that carries out your program directly — an evaluator that executes source or IR without emitting native code. Bytecode — compact instructions for a made-up machine — a serialized instruction set for a virtual machine, typically one byte per opcode. Virtual machine — the program that runs bytecode — an abstract machine specification plus its implementation, such as the JVM or CPython's `ceval` loop. Stack machine — a machine with no named registers — an architecture where operands are pushed and popped on an operand stack. Dispatch — choosing which handler runs next — the fetch-decode-jump step of an interpreter loop, often implemented with computed gotos.

16.13 Just-in-time compilation

■ 16.13.1 PLAIN – in simple words

1. A **just-in-time** compiler, or JIT, compiles your program while it is already running.
2. It does not compile everything. Most code runs once or twice and is not worth the effort.
3. So the system counts. Every time a function is entered or a loop goes round, a counter goes up.
4. When a counter crosses a threshold, that code is **hot**, and the JIT compiles it to real machine instructions.
5. The next time it is reached, the fast version runs instead.
6. A JIT knows something an ordinary compiler cannot: what actually happened. It knows which types really turned up, which branch really got taken, and which function really got called.
7. So it compiles a specialized version based on those observations, and adds a cheap check that the observation still holds.
8. If the check ever fails, it throws that version away and goes back to the slow safe path. That is called **deoptimization**.

■ 16.13.2 PLAIN – a picture in your head

1. Imagine a new receptionist at a large office. On day one they look up every visitor's destination in the directory.
2. After a week they notice that nearly everyone who arrives asks for the third floor, so they start saying "third floor" before checking.

3. But they keep a quick glance at the visitor's badge, in case somebody different turns up.
4. If a badge does not match, they stop guessing, go back to the directory, and look it up properly for that visitor.
5. Guessing is a huge win only because it is nearly always right and because being wrong is safe.

Where this comparison breaks: the receptionist keeps their memory. A real JIT must be able to abandon an optimized version mid-execution, reconstruct the exact state the slow interpreter would have had at that point, and resume. Getting that reconstruction correct is one of the hardest parts of JIT engineering.

■ 16.13.3 PLAIN – a worked example

Take our add, but in a dynamic language, where $a + b$ could mean anything.

1. First call: the interpreter runs `BINARY_OP +`. It looks at the type of `a`, looks at the type of `b`, finds both are whole numbers, and adds them.
2. The system records what it saw: two whole numbers, at this exact spot.
3. Calls two to about a thousand: the same thing, and the recording is confirmed each time.
4. The counter crosses the threshold. The JIT compiles this function assuming both arguments are whole numbers.
5. It emits something close to our `-O2` output: a check, then one add instruction, then a return.
6. The check is a **guard**: “is `a` really a whole number, is `b`?” A few instructions, usually predicted correctly, essentially free.
7. Call 100,000: someone passes a string. The guard fails. The compiled version is abandoned for that call, the state is rebuilt, and the interpreter takes over from that exact point.
8. If it keeps failing, the compiled version is discarded and recompiled with the new information.

Real tiering in the HotSpot JVM, which is a written design, not a guess:

Tier	What runs	Roughly when
0	Interpreter	Always at first
1-3	C1 compiler	After ~1,500 invocations
4	C2 compiler	After ~10,000

The exact thresholds are controlled by `-XX:CompileThreshold` and related flags and vary between JVM versions; treat the numbers as typical defaults rather than fixed values.

■ 16.13.4 PLAIN – what is really happening inside

1. Compilation happens on a background thread, so the program keeps running on the old version while the new one is prepared.
2. **On-stack replacement**, or OSR, handles the awkward case of a loop that is already running and will not be entered again. The system swaps the running frame over to compiled code partway through.
3. **Inline caching**, invented for Smalltalk by L. Peter Deutsch and Allan Schiffman in 1984, remembers at each call site which method was called last time, so the lookup is skipped.
4. A **polymorphic inline cache**, from Urs Hölzle, Craig Chambers and David Ungar's work on Self in 1991, remembers several possibilities at one site.
5. **Hidden classes**, also from Self, give dynamically shaped objects a fixed internal layout, so a field access can become a single load at a fixed offset instead of a dictionary lookup.

- Those three ideas together are the reason JavaScript became fast. They were research results from the 1980s and 1990s that were finally applied to the web in the 2008 to 2012 period.

■ 16.13.5 TECHNICAL – the engineer’s version

- HotSpot came to Sun through the 1997 acquisition of Animorphic and shipped in 1999. It introduced tiered compilation with the C1 client compiler and the C2 server compiler, plus deoptimization back to the interpreter.
- Google’s V8 shipped with Chrome in September 2008, developed in Aarhus under Lars Bak, who had worked on Self and HotSpot. The lineage is direct.
- V8’s pipeline as of 2026 is Ignition, a bytecode interpreter, then Sparkplug, a baseline compiler added in 2021, then Maglev, a mid-tier compiler added in 2023, then TurboFan. Ignition plus TurboFan replaced Crankshaft in Chrome 59, 2017.
- Mozilla’s SpiderMonkey, the original JavaScript engine written by Brendan Eich in 1995, went through TraceMonkey in 2008, and now uses Baseline plus IonMonkey with WarpBuilder.
- Deoptimization needs a precise mapping from optimized machine state back to interpreter state at every safepoint. This is stored as side tables and is a large part of why JIT compilers are hard to write correctly.
- The trade against ahead-of-time compilation:
 - JIT wins on peak speed for dynamic languages, because it can specialize on observed types and inline through virtual calls speculatively.
 - AOT wins on start-up time, on memory, on predictable latency, and on platforms that forbid writable-and-executable memory, such as iOS.
 - JIT costs memory for the compiler, the profiling data and the code cache.
 - JIT hurts tail latency, because compilation and deoptimization happen at unpredictable moments. This matters for trading systems and for games.
- Hybrid approaches now dominate. Java has AppCDS class data sharing and GraalVM native-image ahead-of-time compilation, first released 2019. Android moved from Dalvik’s pure JIT to ahead-of-time ART in Android 5.0, 2014, then to a profile-guided mix in Android 7.0, 2016.
- Established fact: JIT reliably beats plain interpretation for long-running dynamic code. Active research: cheap start-up, predictable latency, and sharing profiles across runs. Marketing claim: any flat statement that a JIT language is “as fast as C”.

■ 16.13.6 WORDS – remember these

JIT — compiling while the program runs — dynamic translation of hot code to native instructions, guided by run-time profiles. Hot path — the code that runs most — a method or loop whose execution counter has crossed a compilation threshold. Tiered compilation — several compilers of increasing quality — a pipeline from interpreter to baseline to optimizing compiler, per method. Deoptimization — abandoning fast code when a guess turns out wrong — reconstructing interpreter state at a safepoint and resuming in the slower tier. Inline cache — remembering what was called here last time — a per-call-site cache of receiver type to target method, monomorphic or polymorphic. On-stack replacement — swapping code under a running loop — transferring an active frame from interpreted to compiled code, or back.

16.14 Modern build reality

■ 16.14.1 PLAIN – in simple words

1. In real projects, nobody types gcc by hand.
2. A **build system** works out what needs rebuilding and runs the commands.
3. It does this by comparing times: if the source is newer than the output, the output is stale and must be rebuilt.
4. Anything that depended on that output is stale too, and so on up the chain.
5. Rebuilding only what changed is an **incremental build**, and it is the difference between a two-second edit-run cycle and a twenty-minute one.
6. A **cross-compiler** runs on one kind of machine and produces code for another. That is how phone apps are built on laptops.
7. A **transpiler** compiles from one high-level language to another, rather than to machine code.
8. A **toolchain** is the whole matched set: compiler, assembler, linker, standard library and headers for one target.

■ 16.14.2 PLAIN – a picture in your head

1. Think of a kitchen preparing a large set menu.
2. Some dishes depend on a stock that takes an hour. Some depend on that stock's sauce. Some depend on nothing.
3. A good kitchen keeps a chart of what depends on what.
4. If the stock is remade, everything downstream of it must be remade. If only a garnish changed, only the garnish is redone.
5. A busy kitchen also keeps a fridge of finished components labelled with exactly which ingredients went into them. If someone orders the same component again with identical ingredients, it comes out of the fridge. That is a build cache.

Where this comparison breaks: a kitchen can smell that the stock has gone off. A build system has only timestamps and hashes. If a timestamp lies, for example because clocks differ between machines or a file was restored from backup, the build system will confidently reuse something wrong. This is a real and common class of bug.

■ 16.14.3 PLAIN – a worked example

Here is our two-file version, driven by make.

```
prog: main2.o addonly.o
    gcc main2.o addonly.o -o prog
main2.o: main2.c addh.h
    gcc -c main2.c -o main2.o
addonly.o: addonly.c
    gcc -c addonly.c -o addonly.o
```

Real session on this machine:

```
$ make
gcc -c main2.c -o main2.o
gcc -c addonly.c -o addonly.o
gcc main2.o addonly.o -o prog

$ make
make: 'prog' is up to date.

$ touch addonly.c && make
gcc -c addonly.c -o addonly.o
gcc main2.o addonly.o -o prog
```



```
$ touch addh.h && make
gcc -c main2.c -o main2.o
gcc main2.o addonly.o -o prog
```

1. The first build compiles both files.
2. The second build does nothing, because nothing is newer than its output.
3. Touching `addonly.c` rebuilds only that object, then relinks.
4. Touching the header rebuilds `main2.o`, because the rule lists `addh.h` as a dependency. That listing is the whole trick, and forgetting it is the classic cause of “it did not pick up my header change”. Generate it automatically with `gcc -MMD`.

■ 16.14.4 PLAIN – what is really happening inside

1. Make builds a graph of files and rules, sorts it so dependencies come first, and runs any rule whose output is older than any of its inputs.
2. Modern build systems replace timestamps with **content hashes**, which is more reliable: the same input bytes plus the same command always give the same output.
3. That is what makes a **build cache** possible. Hash the compiler version, the flags and the preprocessed source; if that hash was seen before, reuse the stored object file. `ccache` does exactly this locally, and Bazel and Gradle do it across a whole team.
4. **Cross-compilation** works because the pipeline in 16.1 is already split. The front end does not care about the target. Only the back end, the assembler, the linker and the libraries do.
5. Our chapter already did it. `clang --target=aarch64-linux-gnu -O2 -S add.c` produced ARM instructions on an x86 machine, with no special install, because LLVM ships every backend in one binary.
6. To produce a runnable ARM binary you also need ARM headers and libraries. That is the difference between a compiler and a toolchain, and it is why cross-toolchain setup is more annoying than cross-compiling itself.

■ 16.14.5 TECHNICAL – the engineer’s version

1. `make` was written by Stuart Feldman at Bell Labs in 1976, reportedly after losing a morning to a program that had failed to relink. Its tab-indentation rule has never been fixed for compatibility reasons.
2. `CMake` was created at Kitware around 2000 for the Insight Toolkit, funded by the US National Library of Medicine. It is a **build generator**: it does not build, it writes Makefiles, Ninja files or Visual Studio projects.
3. `Ninja`, written by Evan Martin for the Chrome build in the early 2010s, is deliberately not human-authored. Its input is machine-generated, its dependency handling is fast, and it is the usual back end for `CMake` and for Meson on large projects.
4. `Gradle` targets the JVM world, uses a Groovy or Kotlin domain-specific language, supports incremental and cached tasks, and reached 1.0 in 2012. `Bazel` is the open-sourced version of Google’s `Blaze`, released in 2015, and is built around hermetic, hash-keyed, remotely cacheable actions.
5. `ccache`, from Andrew Tridgell in the early 2000s, and `distcc`, from Martin Pool around the same time, remain the cheapest large wins for C and C++ projects that cannot adopt Bazel.
6. **Transpilers**: `TypeScript`, announced by Microsoft in October 2012 under Anders Hejlsberg, compiles to JavaScript. `Babel`, started as `6to5` by Sebastian McKenzie in 2014, compiles newer JavaScript to older JavaScript. The very first C++ implementation, Bjarne Stroustrup’s `Cfront` from 1983, was a transpiler to C.

7. **WebAssembly** is a portable compilation target with a formal specification. Its Core Specification 1.0 became a W3C Recommendation in December 2019.
 1. Our example built with `clang --target=wasm32 -O2 -nostdlib -c add.c` gave a 333-byte module.
 2. Its first eight bytes are `00 61 73 6d 01 00 00 00`, that is `\0asm` followed by version 1.
8. Why build times matter: they set the length of the edit-compile-test loop, which sets how often a developer tries an idea. As a scale marker, a full Chromium build is commonly reported in hours on one workstation and minutes with a large distributed cache.

Tool	Year	Kind
make	1976	Build executor
Cfront	1983	C++ to C transpiler
CMake	2000	Build generator
ccache	early 2000s	Compiler cache
Ninja	early 2010s	Build executor
Gradle 1.0	2012	Build system, JVM
TypeScript	2012	Transpiler
Bazel	2015	Build system, cached
Wasm 1.0	2019	Portable target

■ 16.14.6 WORDS – remember these

Build system — the thing that decides what to recompile — a dependency graph executor keyed on timestamps or content hashes. Incremental build — rebuild only what changed — recomputing the minimal set of stale targets from the dependency graph. Build cache — reuse a result somebody already computed — a content-addressed store keyed on inputs, tool version and flags. Cross-compilation — building on one machine for another — using a toolchain whose target triple differs from the host triple. Transpiler — a compiler whose output is another language — a source-to-source translator, such as TypeScript to JavaScript. Toolchain — the whole matched set of tools — compiler, assembler, linker, headers and runtime libraries for one target.

16.15 The bootstrap problem and trusting trust

■ 16.15.1 PLAIN – in simple words

1. A C compiler is a program. Programs must be compiled. So what compiled the first C compiler?
2. This is the **bootstrap problem**, and there are only three honest answers.
3. One: write the first version in something else, such as assembly or another existing language.
4. Two: write a small, limited version by hand, then use it to compile a bigger version, then use that to compile a bigger one still.
5. Three: use a compiler on a different machine to produce code for yours. That is cross-compilation from 16.14.
6. A compiler written in the language it compiles is **self-hosting**. Almost every serious compiler ends up this way.
7. Self-hosting is a strong test: the compiler is its own largest and most demanding user.

■ 16.15.2 PLAIN – a picture in your head

1. Imagine you want a workshop that can build any tool, including its own tools.
2. You cannot start there. You start with a rough hammer made by hand.
3. With the rough hammer you make a better hammer. With the better hammer you make a decent lathe. With the lathe you make precise tools.
4. Eventually the workshop can make every tool in it, including exact copies of its own machines.
5. The first rough hammer is then thrown away, and nobody alive has seen it.

Where this comparison breaks: a hammer's shape is visible. A compiler can carry an invisible instruction inside it that is not written in any source file anybody keeps, and that instruction can copy itself into every tool the workshop makes. That is the whole point of the next block.

■ 16.15.3 PLAIN – a worked example

1. Start with our own file. `gcc add.c` worked because a `gcc` binary already existed on this machine. That binary is itself written in C and C++, so some earlier compiler must have built it. Follow that chain back far enough and you reach the question of where the first one came from.
2. Around 1972 to 1973, Dennis Ritchie's C compiler at Bell Labs was written in an earlier language, first B and then an intermediate step usually called NB or "new B", and grew into C by stages, each version compiling the next.
3. In 1973 the Unix kernel was rewritten in C for Version 4 Unix, which is the moment portable operating systems become possible.
4. Corrado Böhm's 1951 PhD thesis at ETH Zurich described the first compiler for a language written in that same language, so the idea is older than C by two decades.
5. Tim Hart and Mike Levin wrote a Lisp compiler in Lisp at MIT in 1962, then ran it through the Lisp interpreter to compile itself.
6. Modern examples: the Rust compiler became self-hosted in April 2011. Go removed the last of its C compiler in Go 1.5, August 2015, and has been written in Go since.
7. Today's route for a brand-new C compiler is easier. Compile version one with GCC, compile version one with itself, compile once more, and check the last two outputs are byte-identical. That is a three-stage bootstrap, and GCC does it in its own build.

■ 16.15.4 PLAIN – what is really happening inside

1. Now Ken Thompson's argument, in plain words. He gave it in his 1984 Turing Award lecture, *Reflections on Trusting Trust*, printed in Communications of the ACM in August 1984. He and Dennis Ritchie shared the 1983 Turing Award.
2. Step one. Suppose you modify a C compiler so that when it notices it is compiling the login program, it silently adds a back door accepting a secret password.
3. That is easy but obvious: the extra code is right there in the compiler's source, and any reviewer would see it.
4. Step two. So add a second rule: when the compiler notices it is compiling a C compiler, it silently inserts both rules into the output.
5. Now compile the compiler with the modified compiler. The new binary contains both rules.
6. Step three. Remove all of it from the source. Delete every line. The source is clean and reviewable and contains nothing suspicious.
7. But the compiled compiler still carries both rules, because it was built by a compiler that inserts them. Compile the clean source with that binary, and the new binary has them again.
8. The attack now lives only in the binary, reproducing itself forever, with no trace in any source file anybody reads.

9. Thompson's conclusion: you cannot trust code you did not totally write yourself, and "totally" reaches all the way down through the compiler, the assembler, the linker, the loader, the operating system and the hardware.
10. He also noted that no amount of source-level inspection protects you, because the source is genuinely clean.

■ 16.15.5 TECHNICAL – the engineer's version

1. The paper's technique is a quine-like self-reproducing program combined with two pattern-matching triggers. Thompson stated that he had actually built a working version, though it was never released.
2. The practical defence is **diverse double-compiling**, from David A. Wheeler's 2005 paper and 2009 PhD dissertation. Compile the suspect compiler's source with a second, independently written compiler, then use both binaries to compile that source again. Identical final binaries mean neither carries the attack.
3. This only works if the compiler is deterministic: the same source and flags must always give byte-identical output. That requirement is precisely the goal of the **Reproducible Builds** project, which began within Debian around 2013.
4. Sources of non-determinism that have to be removed: embedded build timestamps, build paths in debug information, file ordering from directory reads, random hash seeds and thread scheduling. GCC and Clang both accept `SOURCE_DATE_EPOCH` and `-ffile-prefix-map` to help.
5. As of 2026, Debian reports that the large majority of its source packages build reproducibly, and several distributions publish rebuild verification. Exact percentages move month to month, so treat any single figure as dated.
6. The related project **Bootstrappable Builds** attacks the other end: shrinking the trusted seed binary to something a human can audit. Its chain starts from a few hundred bytes of hand-checkable machine code and climbs through stage0, M2-Planet, TinyCC and GCC.
7. Established fact: the attack works and is well understood. Active research: verifying the whole chain from hardware upward, including proved compilers such as CompCert, machine-checked in Coq since 2008. Marketing claim: any product said to have solved supply-chain trust.
8. The honest version: reproducible builds and diverse double-compiling reduce the trusted base, they do not eliminate it. You still trust the CPU microcode, the firmware and the fabrication process. Thompson's point survives; we have only made the untrusted part smaller.

■ 16.15.6 WORDS – remember these

Bootstrapping — getting a compiler off the ground with no compiler — building a language's compiler through successive stages from an existing toolchain. Self-hosting — a compiler written in its own language — a compiler that can compile its own source, usually verified by a multi-stage build. Three-stage bootstrap — build it three times and compare — stage 2 and stage 3 outputs must be byte-identical, a standard part of the GCC build. Trusting trust — the compiler can lie and the source will not show it — Thompson's 1984 self-reproducing compiler back door. Reproducible build — same source in, same bytes out, every time — a build whose output is a deterministic function of its declared inputs. Diverse double-compiling — check one compiler using a different one — Wheeler's method for detecting a trusting-trust attack in a deterministic compiler.

16.98 Common wrong ideas

1. Wrong: the compiler translates your code line by line into machine code. Right: it goes through the whole chain of 16.1, and the optimizer may delete, merge, reorder and duplicate your lines. Our two-line file became three instructions with no call in it.
2. Wrong: `#include` imports a module the way Python's `import` does. Right: it pastes the file's text at that point. Two lines of C with `#include <stdio.h>` became 815 lines of text on this machine.
3. Wrong: a syntax error means the compiler does not understand what you meant. Right: a syntax error means no grammar rule allows the token it just saw. Meaning is checked later, in semantic analysis, and produces different errors.
4. Wrong: undefined reference is a compiler error. Right: it is a linker error. The compiler was satisfied by the declaration. The linker could not find a definition. That is why the fix is usually an extra object file or library, not a code change.
5. Wrong: `-O2` makes your program do the same steps, faster. Right: it makes the program produce the same observable result by doing different steps. Our `main` never calls `add` at all after `-O2`.
6. Wrong: undefined behaviour means the program does something unpredictable at that point. Right: it means the compiler may assume it never happens, which can change code far away from the mistake. `x + 1 > x` folded to a constant 1 at `-O2`.
7. Wrong: floating point differences between `-O0` and `-O2` are compiler bugs. Right: they can be legitimate, especially with `-ffast-math`, because IEEE 754 addition is not associative. On this machine `(a+b)+c` gave 1 and `a+(b+c)` gave 0 for the same three values.
8. Wrong: interpreted languages are slow because interpreting is slow. Right: mostly they are slow because operations must decide their meaning at run time. Java bytecode is interpreted too and is far faster, because `iadd` already knows it is adding two 32-bit integers.
9. Wrong: a JIT is always faster than ahead-of-time compilation. Right: it wins on peak speed for dynamic code and loses on start-up time, memory use and latency predictability. That is why iOS forbids it and why GraalVM native-image exists.
10. Wrong: reading a compiler's source proves it has no back door. Right: Thompson showed in 1984 that a compiler binary can carry an attack that appears in no source file. Diverse double-compiling, not reading, is the defence.

16.99 Chapter summary in 20 lines

1. A compiler is a chain of stages, each turning one shape of information into a simpler one, from text to running process.
2. The preprocessor is a text editor: it pastes files in and substitutes macros, and it knows nothing about C.
3. The lexer cuts characters into labelled tokens using a finite automaton, and always takes the longest legal match.
4. The parser arranges tokens into an abstract syntax tree, guided by a context-free grammar written in Backus-Naur Form.
5. A syntax error means no grammar rule allows the current token. It is about shape, not meaning.
6. Semantic analysis builds a scoped symbol table, resolves names, checks types and inserts implicit conversions.
7. Compilers then rewrite the program into an intermediate representation: flat three-address code, usually in static single assignment form.
8. SSA gives every value exactly one definition, with phi nodes at joins, which makes most optimizations simple and fast.
9. The front-end, middle-end, back-end split lets many languages share one optimizer and one set of processor backends.
10. The optimizer may do anything that preserves observable behaviour: fold, propagate, delete, inline, unroll, vectorize, and eliminate tail calls.
11. Our `int main(void) { return add(2,3); }` became `movl $5, %eax; ret` through inlining, then constant folding, then dead code elimination.
12. Undefined behaviour lets the optimizer assume the impossible never happens, which is why `x + 1 > x` folds to 1 for signed integers but not unsigned.
13. Code generation performs instruction selection, scheduling and register allocation, and must obey a calling convention.
14. System V AMD64 passes the first two integers in `rdi` and `rsi`, Windows x64 uses `rcx` and `rdx`, AArch64 uses `x0` and `x1`. All return in the first of those.
15. The assembler turns instruction text into bytes, groups them into sections, records symbols, and leaves relocations where addresses are unknown.
16. The linker merges sections, resolves symbols and patches relocations. Our `e8 00 00 00 00` became `e8 d1 ff ff ff`, a relative jump of minus 47 bytes.
17. Library order on the command line matters, because archives are scanned once against the symbols still undefined at that moment.
18. At exec time the kernel maps segments and starts the dynamic linker, which loads shared libraries and fills the GOT, lazily by default.
19. Bytecode virtual machines trade speed for portability, and a JIT wins it back by compiling hot paths using types observed at run time, with deoptimization as the safety net.
20. Compilers bootstrap themselves, and Ken Thompson showed in 1984 that a compiler binary can hide a back door that no source file reveals, which is why reproducible and diverse builds matter.

Programming Languages and How to Think in Them

17.0 What this chapter gives you

1. You will be able to say what a programming language actually is, and why there are thousands of them instead of one.
2. You will be able to read a small program in six different languages and see that they are all saying the same thing.
3. You will be able to explain what a type is, why static and dynamic typing argue with each other, and what the billion dollar mistake was.
4. You will be able to describe the three ways a program can manage memory, and name the exact failure each one prevents and each one allows.
5. You will be able to name the main programming paradigms and write the same task in two of them.
6. You will be able to pick the right data structure for a job and state its cost in Big-O notation without looking it up.
7. You will be able to explain why concurrency is genuinely hard, and show a race condition line by line.
8. You will be able to write a test, read someone else's code, and judge a dependency before you install it.
9. You will be able to choose a language for a real project and defend the choice with reasons instead of taste.

17.1 What a programming language is

■ 17.1.1 PLAIN – in simple words

1. A programming language is a way of writing down instructions that both a person and a machine can work with.
2. It has a **grammar**: rules about which arrangements of symbols are legal.
3. It has a **meaning**: rules about what each legal arrangement does.
4. Those two things are separate. A sentence can be perfectly formed and still say something silly or wrong.
5. The grammar is called **syntax**. The meaning is called **semantics**.
6. A language is designed for humans first. Machines only understand numbers, and something else turns your text into those numbers.

7. That something else is a compiler or an interpreter. Chapter 16 covered how a compiler works. Here we care about the language itself.
8. A language is not the same as its tools. C is a written standard. GCC and Clang are two programs that implement it.

■ 17.1.2 PLAIN – a picture in your head

1. Think of a knitting pattern.
2. A pattern has a strict notation: `k2`, `p2`, `rep to end`. Every symbol means one exact action.
3. The notation has a grammar. `k2` is legal. `k` on its own with no number is not, because the pattern must say how many.
4. The notation also has a meaning. `k2` means knit two stitches. Follow it and a real object appears.
5. Two knitters in different countries use different pattern notations. Both make the same jumper. The garment is the program, the notation is the language.
6. A pattern can be written correctly and still produce a jumper with three sleeves. The notation was obeyed. The intention was wrong.

Where this comparison breaks: a knitting pattern is followed by a patient human who silently fixes obvious mistakes. A computer fixes nothing. It does exactly what you wrote, at billions of steps per second, including the wrong thing. A pattern also has no way to say “repeat until the wool runs out and then decide what to do next”. Real languages do, and that ability to make decisions is what separates a program from a recipe.

■ 17.1.3 PLAIN – a worked example

1. Here are three lines of Python. Each is broken in a different way.

```
print("hello"  
x = 10 / 0  
average = total / count
```

2. Line 1 is a **syntax error**. The closing bracket is missing. The grammar was violated. Python refuses to run the file at all.
3. Line 2 is legal grammar but impossible meaning. Division by zero. Python starts running, reaches this line, and stops with an error.
4. Line 3 has perfect grammar and perfect meaning. It will run happily. But if `count` is the number of rows in the file and `total` is the sum of a different column, the answer is nonsense and nothing complains.
5. That third case is the one that costs money. No tool caught it because nothing was broken. The program did exactly what you said.
6. This is the whole of programming in one example: getting the syntax right is easy and the machine helps you. Getting the meaning right is hard and you are mostly on your own.

■ 17.1.4 PLAIN – what is really happening inside

1. When you save a source file you have saved plain text. Bytes. Nothing more.
2. A tool reads that text left to right and chops it into **tokens**: names, numbers, symbols, keywords.
3. It then checks whether those tokens form a legal shape, using the grammar. This is parsing, and it builds a tree of the program’s structure.
4. If the shape is illegal you get a syntax error, with a line number.
5. If the shape is legal, the tool works out what it means, and produces either machine instructions or a set of actions it will perform itself.

6. Because a language is a written definition and not a program, several different tools can implement it. Python has CPython, PyPy and others.
7. This is why “the language is slow” is usually the wrong sentence. The implementation is slow. The language is a document.
8. Why thousands of languages exist: every language is a set of trade-offs. Fast to write against fast to run. Safe against flexible. Small against expressive. Nobody has found one point that wins everywhere.
9. A new language is also cheap to start and expensive to finish. Thousands are started. About twenty are widely used at any moment.

■ 17.1.5 TECHNICAL – the engineer’s version

1. Syntax is normally specified as a context-free grammar in Backus-Naur Form. John Backus described the notation in 1959 for ALGOL 58, and Peter Naur refined it in the ALGOL 60 report of 1960, which is why it carries both names.
2. Noam Chomsky published the hierarchy of formal grammars in 1956. Type 3 is regular (what a lexer handles), type 2 is context-free (what a parser handles), type 1 is context-sensitive, type 0 is unrestricted.
3. Real languages are not purely context-free. C requires a symbol table to decide whether `A * B`; is a multiplication or a pointer declaration. This is called the lexer hack, and it is an implementation detail, not a standard.
4. Semantics has three classical formal styles: operational (Gordon Plotkin’s structural operational semantics, 1981), denotational (Dana Scott and Christopher Strachey, around 1970), and axiomatic (Tony Hoare, in the October 1969 paper “An Axiomatic Basis for Computer Programming”).
5. In practice most language specifications define semantics in careful English prose, not in formal mathematics. Standard ML and WebAssembly are among the very few with a fully formal specification.
6. Undefined behaviour is a specification concept, not a bug: the standard declines to say what happens, so the compiler may assume it never occurs. Signed integer overflow in C is the classic case.

Language	Governing document	Latest as of Aug 2026
C	ISO/IEC 9899	C23, published 2024
C++	ISO/IEC 14882	C++26, March 2026
JavaScript	ECMA-262	ECMAScript 2025
SQL	ISO/IEC 9075	SQL:2023
Python	PEPs, no ISO standard	3.14, Oct 2025

7. There is no census of languages. The Online Historical Encyclopaedia of Programming Languages lists roughly 9,000 entries, which is an approximate figure and includes long-dead research languages.
8. Tools that show you the grammar in action: `python -m ast`, `gcc -fdump-tree-` original, `clang -Xclang -ast-dump`, and `node --print-ast` style flags in various runtimes.

■ 17.1.6 WORDS – remember these

1. Syntax — the spelling and shape rules — the context-free grammar of the language.
2. Semantics — what it means when it runs — the operational or denotational definition of program behaviour.
3. Token — one word or symbol — the smallest unit produced by lexical analysis.

4. Parsing — checking the shape — building an abstract syntax tree from tokens according to the grammar.
5. Specification — the official rulebook — the ISO, ECMA or community document that defines the language, separate from any implementation.
6. Undefined behaviour — the rules do not say — a construct the standard leaves unconstrained, allowing any compiler output at all.

17.2 The building blocks every language has

■ 17.2.1 PLAIN - in simple words

1. Almost every language is built from the same ten or so ideas.
2. A **variable** is a named box that holds a value you can change.
3. A **literal** is a value written down directly, like 42 or "cat".
4. An **operator** is a symbol that combines values, like + or <.
5. An **expression** is anything that produces a value: 2 + 3, name, total > 100.
6. A **statement** is a complete instruction that does something: assign, print, return.
7. A **conditional** chooses between paths: if this, do that, otherwise do the other.
8. A **loop** repeats a block, either a fixed number of times or until some test stops being true.
9. A **function** is a named piece of program you can call from elsewhere, optionally handing it values and getting one back.
10. **Scope** is the region of the program where a name is visible.
11. A **comment** is text the machine ignores completely, written for humans.

■ 17.2.2 PLAIN - a picture in your head

1. Think of a kitchen with labelled jars on a shelf.
2. A jar is a variable. The label is the name, the contents are the value.
3. A literal is an ingredient you bring in fresh, not from a jar.
4. An operator is a kitchen action: mix these two, compare these two.
5. A recipe step that produces something you can hold is an expression. A step that just says "turn on the oven" is a statement.
6. A conditional is "if the dough is sticky, add flour".
7. A loop is "stir for ten minutes".
8. A function is a sub-recipe on its own card: "make the sauce". You can use it from any recipe without rewriting it.
9. Scope is which shelf you can reach from where you are standing. Jars in the locked pantry are out of scope.

Where this comparison breaks: a cook can ignore a step or improvise. A program cannot. Also, a real recipe never calls itself, and functions often do, which is called recursion and has no clean kitchen equivalent.

■ 17.2.3 PLAIN - a worked example

1. Here is the same tiny program in six languages. It adds the numbers 1 to 10 and prints 55.
2. Read them one after another. Notice that the punctuation changes and nothing else does.

```
#include <stdio.h>

int main(void) {
    int total = 0;
```

```

    for (int i = 1; i <= 10; i++) {
        total = total + i;
    }
    printf("%d\n", total);
    return 0;
}

total = 0
for i in range(1, 11):
    total = total + i
print(total)

let total = 0;
for (let i = 1; i <= 10; i++) {
    total = total + i;
}
console.log(total);

public class Sum {
    public static void main(String[] args) {
        int total = 0;
        for (int i = 1; i <= 10; i++) {
            total = total + i;
        }
        System.out.println(total);
    }
}

package main

import "fmt"

func main() {
    total := 0
    for i := 1; i <= 10; i++ {
        total = total + i
    }
    fmt.Println(total)
}

fn main() {
    let mut total = 0;
    for i in 1..=10 {
        total = total + i;
    }
    println!("{}", total);
}

```

3. Every one of them declares a variable called `total`, sets it to zero, loops `i` from 1 to 10, adds `i` to `total`, and prints the result.
4. The differences are: whether you write the type (`int`) or not, whether lines end in a semicolon, whether blocks use braces or indentation, and how much ceremony wraps the main body.
5. Java has the most ceremony because every piece of code must live inside a class. Python has the least because the file itself is the program.
6. Rust needs `mut` because in Rust a variable cannot be changed unless you say so explicitly.
7. Go uses `:=` to mean “make a new variable and work out its type for me”.
8. If you can read one of these, you can read all six with about an hour of effort. That is the real point of this section.

■ 17.2.4 PLAIN – what is really happening inside

1. A variable is a name the compiler maps to a place: a CPU register, a slot on the stack, or an address in the heap.
2. In `total = total + i`, the machine loads two values, adds them in a register, and stores the result back. Three steps, one line of your text.
3. `i++` is one instruction on most processors. `total = total + i` is usually one instruction too, once the values are in registers.
4. A loop is a conditional jump. At the bottom of the block, compare `i` with 10, and if the test passes, jump back to the top.
5. A function call pushes the return address and arguments somewhere the callee can find them, jumps to the function's code, and later jumps back.
6. Scope is a compile-time idea, not a runtime one. The compiler simply refuses to resolve a name that is not visible, and produces an error.
7. Comments never reach the machine. The lexer throws them away before the parser ever sees them.
8. Here is what the loop becomes in spirit:

```

    total <- 0
    i <- 1
top:
    if i > 10 goto done
    total <- total + i
    i <- i + 1
    goto top
done:
    print total

```

9. Every one of the six languages above compiles or interprets down to something with this shape. The high-level `for` is a convenience.

■ 17.2.5 TECHNICAL – the engineer's version

1. Expressions have precedence and associativity, fixed by the grammar. In C, `a + b * c` parses as `a + (b * c)` because `*` binds tighter. `a - b - c` parses as `(a - b) - c` because `-` is left-associative.
2. Statement versus expression is a real language design axis. In Rust and in most functional languages, `if` is an expression and returns a value. In C and Java it is a statement and returns nothing, which is why those languages need a separate ternary operator `?:`.
3. Scope rules split into lexical (static) and dynamic. Almost every modern language uses lexical scope: a name resolves by where it is written. Emacs Lisp and Bash use dynamic scope in places, where a name resolves by who called you.
4. Variable lifetime is separate from scope. A C `static` local has function scope but program lifetime.
5. Parameter passing conventions: call by value (C, Java for primitives, Go), call by reference (C++ with `&`, C# with `ref`), and call by sharing (Python, JavaScript, Java for objects, where the reference is copied but the object is not).
6. This is the source of a common confusion. In Python, reassigning a parameter inside a function does not affect the caller, but calling `list.append` on it does. Both are consistent with call by sharing.

Construct	C	Python	Rust
Block delimiter	braces	indentation	braces
Statement end	semicolon	newline	semicolon

Construct	C	Python	Rust
Declare variable	<code>int x = 1;</code>	<code>x = 1</code>	<code>let x = 1;</code>
Mutable by default	yes	yes	no

7. Tools: `python -m dis` shows the bytecode a Python function becomes. godbolt.org style compiler explorers show the assembly for C, C++, Rust and Go. `javap -c` disassembles a Java class file.

■ 17.2.6 WORDS – remember these

1. Variable — a named box for a value — a binding from an identifier to a storage location with a scope and lifetime.
2. Literal — a value written directly — a constant token whose value is fixed at compile time.
3. Expression — something that produces a value — a grammar production that evaluates to a value of some type.
4. Statement — an instruction that acts — a grammar production executed for its effect rather than its value.
5. Scope — where a name can be seen — the lexical region in which an identifier binding is visible.
6. Call by sharing — the object is shared, the variable is not — the argument reference is copied, so mutation is visible and rebinding is not.

17.3 Types

■ 17.3.1 PLAIN – in simple words

1. A **type** is the answer to “what kind of thing is this value”.
2. 42 is a whole number. "42" is a piece of text. 4.2 is a decimal number. They look similar and behave completely differently.
3. Types exist to stop you doing meaningless things, like subtracting a name from a date.
4. They also tell the machine how many bytes to set aside and which instructions to use. Adding two whole numbers uses different hardware from adding two decimals.
5. Some languages check types before the program runs. That is **static typing**.
6. Some languages check types while the program runs. That is **dynamic typing**.
7. Some languages refuse to mix types silently. That is **strong typing**.
8. Some languages quietly convert one type into another to make an expression work. That is **weak typing**.
9. Static and strong are different questions. A language can be any of the four combinations.

■ 17.3.2 PLAIN – a picture in your head

1. Think of an airport with two kinds of security.
2. Static typing is the check at the gate before boarding. Nobody without a valid ticket gets on the plane at all.
3. Dynamic typing is the check by the cabin crew mid-flight. Most people are fine, but if someone is in the wrong seat you find out at 30,000 feet.
4. Strong typing is a strict officer who says a bus ticket is not a plane ticket, full stop.
5. Weak typing is an officer who looks at your bus ticket, decides it is probably fine, and lets you through.
6. Static checking catches problems earlier and cheaper. Dynamic checking lets you board faster and change plans later.

Where this comparison breaks: static checks are not free and they are not complete. A type checker proves that the shapes fit together. It does not prove your program is correct. A program that computes the wrong average with perfectly matched types passes every static check ever written.

■ 17.3.3 PLAIN – a worked example

1. Watch the same expression in three languages.

Expression: `1 + "2"`

Python -> `TypeError`, refuses to run this line

JavaScript -> `"12"` (number turned into text)

PHP -> `3` (text turned into number, PHP 8+)

2. Python is dynamically typed and strongly typed. It waits until runtime to check, and then refuses to guess.
3. JavaScript is dynamically typed and weakly typed. It converts the number to text and joins them.
4. PHP is dynamically typed and weakly typed in the other direction for `+`, because `+` in PHP is only arithmetic and `.` is the joining operator.
5. Now the static side. In Java, `int x = "hello";` will not compile. The error arrives in under a second, at your desk, before anyone else sees the code.
6. In Python, `x = "hello"` followed a hundred lines later by `x + 1` will run fine until it reaches that line, possibly in production at 3am.
7. **Type inference** means the compiler works out the type so you do not have to write it. In Rust, `let x = 5;` gives an `i32` without you saying so. This is still static typing. The type is fixed, it just was not typed by hand.

■ 17.3.4 PLAIN – what is really happening inside

1. A type is mostly a compile-time idea. After compilation, memory holds bytes and nothing else.
2. The type told the compiler which machine instruction to emit. `ADD` for integers, `ADDSD` for double-precision decimals on x86.
3. In a dynamic language, every value carries a tag at runtime saying what it is. That tag costs memory and a check on every operation.
4. That is why a Python integer needs about 28 bytes while a C integer needs 4. The extra bytes hold the type pointer, a reference count and the digits.
5. **Composite types** build bigger shapes from smaller ones: a struct or record groups named fields, an array groups values of one type, a tuple groups a fixed number of possibly different types.
6. **Generics** let you write one piece of code that works for many types. A list of integers and a list of strings share one implementation. Without generics you either copy the code or throw away the type information.
7. **Null** is a special value meaning “there is nothing here”. It is allowed where a real value is expected, which is exactly the problem.
8. The fix is an **option type**: a value that is explicitly either “something” or “nothing”, and the language will not let you use it without checking which.

■ 17.3.5 TECHNICAL – the engineer’s version

1. Sizes. The C standard guarantees only minimum widths, so the exact sizes below are the near-universal LP64 arrangement used by Linux and macOS on x86-64 and ARM64. Windows uses LLP64, where `long` is 4 bytes.

Type	Size	Range or precision
int8, signed char	1 byte	-128 to 127
uint8, unsigned char	1 byte	0 to 255
int16, short	2 bytes	-32,768 to 32,767
int32, int	4 bytes	-2,147,483,648 up
int64, long (LP64)	8 bytes	about -9.2e18 to 9.2e18
float, binary32	4 bytes	about 7 decimal digits
double, binary64	8 bytes	about 15 to 17 digits
bool	1 byte	true or false
Rust char	4 bytes	one Unicode scalar value

- Floating point follows IEEE 754, first published in 1985 and revised in 2008 and 2019. binary32 has 24 bits of significand, binary64 has 53. This is why $0.1 + 0.2$ is 0.30000000000000004 in every language that uses binary64, including Python, JavaScript and Java.
- JavaScript numbers are all binary64. Integers are exact only up to $2^{53} - 1$, which is 9,007,199,254,740,991, exposed as `Number.MAX_SAFE_INTEGER`. BigInt was added in ECMAScript 2020 for larger values.
- Type system taxonomy: static or dynamic (when checking happens), strong or weak (how much implicit conversion is allowed), nominal or structural (do two types match by name or by shape). Go interfaces and TypeScript are structural. Java classes are nominal.
- Type inference in ML-family languages uses Hindley-Milner, published by Roger Hindley in 1969 and independently by Robin Milner in 1978. Rust and Swift use local inference, which is weaker but gives better error messages.
- Generics are implemented either by monomorphization (Rust, C++ templates, C# value types), which duplicates code per type and is fast, or by erasure (Java, which added generics in Java 5 in 2004), which keeps one copy and loses the type at runtime.
- Null. Tony Hoare introduced the null reference in 1965 while designing the type system of ALGOL W. At a software conference in 2009 he called it “my billion-dollar mistake”, saying it had led to “innumerable errors, vulnerabilities and system crashes”.
- Option types are the fix: `Option<T>` in Rust, `Optional<T>` in Java 8 and Swift, `Maybe a` in Haskell, `T?` with null-safety checks in Kotlin, and `strictNullChecks` in TypeScript since version 2.0 in 2016.

```
fn find(name: &str) -> Option<u32> {
    if name == "ada" { Some(1815) } else { None }
}

match find("ada") {
    Some(year) => println!("born {}", year),
    None => println!("not found"),
}
```

- The compiler will not let you reach `year` without handling `None`. That is the entire trick: the checking is not optional and not forgettable.
- Tools: `mypy` and `pyright` add static checking to Python. `tsc` does it for JavaScript via TypeScript. `sorbet` does it for Ruby.

■ 17.3.6 WORDS – remember these

- Type — what kind of value it is — a set of values plus the operations legal on them.
- Static typing — checked before running — type checking performed at compile time by the language’s type system.

3. Dynamic typing — checked while running — types carried on values and verified at each operation.
4. Strong typing — no silent conversions — the language rejects operations between incompatible types rather than coercing.
5. Type inference — the compiler guesses correctly — deducing types from context without explicit annotations, still statically.
6. Generics — one implementation for many types — parametric polymorphism, realized by monomorphization or by erasure.
7. Option type — a value that may be absent, safely — a sum type with a present and an absent case, forcing the caller to handle both.

17.4 Memory management

■ 17.4.1 PLAIN - in simple words

1. A running program needs space to keep things. That space is memory.
2. Some space is easy. A number you use inside one function lives on the **stack**, and disappears by itself when the function ends.
3. Some space is hard. Anything whose size you only learn while running, or that must outlive the function that made it, lives on the **heap**.
4. Heap space has to be asked for, and later given back.
5. There are exactly three answers to “who gives it back”.
6. Answer one: you do, by hand. This is C and old C++. It is fast and it is easy to get wrong.
7. Answer two: the language does, automatically, by watching your values while the program runs. This is Java, C#, Go, Python, JavaScript and most others.
8. Answer three: the compiler works it out before the program runs, from rules you followed while writing. This is Rust.
9. Every one of the three is a real trade. None of them is free.

■ 17.4.2 PLAIN - a picture in your head

1. Think of a library with a room of borrowed books.
2. Manual memory management is a library with no due dates. You borrow a book and you must remember to return it. If you forget, the room fills up. If you return the same book twice, the desk gets confused. If you return a book and then keep reading it, you are reading a book that has been given to someone else.
3. Reference counting is a library where each book has a tally on the cover. Add one when someone borrows it, subtract one when they finish. At zero, it goes back on the shelf immediately.
4. Tracing garbage collection is a librarian who periodically walks the whole building, notes every book anyone is actually holding, and reshelves the rest. While she walks, nobody may move.
5. Rust’s ownership is a library rule that each book has exactly one owner, and the owner must return it when they leave the room. The rule is checked at the door, before you enter.

Where this comparison breaks: books are visible and countable, and a librarian can see who is holding one. Real memory has no such property. The collector can only follow pointers from a known starting set, and anything not reachable that way is assumed dead, even if you meant to keep it.

■ 17.4.3 PLAIN – a worked example

1. Here is manual allocation in C, done correctly.

```
#include <stdlib.h>

int *make_array(int n) {
    int *a = malloc(n * sizeof(int));
    if (a == NULL) return NULL;
    for (int i = 0; i < n; i++) a[i] = i;
    return a;
}

/* caller must eventually call free(a) */
```

2. malloc asks for a block of bytes and hands back its address. free gives it back. Nothing else does.
3. Now the four ways this goes wrong.
4. **Leak**: you never call free. The block stays reserved for the life of the process. Do it in a loop and memory grows until the process is killed.
5. **Double free**: you call free(a) twice. The allocator's internal list is corrupted. This is a classic route to a security exploit.
6. **Use after free**: you call free(a) and then read a[0]. The bytes may still look right for a while, which makes this bug appear and disappear at random.
7. **Dangling pointer**: you return the address of a local variable, which lived on the stack and vanished the moment the function ended.

```
int *bad(void) {
    int x = 42;
    return &x; /* x dies here; the pointer is dangling */
}
```

8. All four compile without error in plain C. Three of the four are silent during testing and loud in production.

■ 17.4.4 PLAIN – what is really happening inside

1. Reference counting. Each object carries a small number. Every time a new name points at it, add one. Every time a name stops pointing at it, subtract one. At zero, free it at once.
2. That is simple, immediate and predictable. It has two costs. Every assignment does extra arithmetic, and in a multi-threaded program that arithmetic must be atomic, which is slow.
3. It also cannot free a cycle. If A points at B and B points at A, both counts stay at one forever even though nothing outside can reach them.
4. Tracing garbage collection solves the cycle problem by asking a different question: not “how many people point at this” but “can I still reach this”.
5. Mark and sweep works in two phases. Start from the **roots** (global variables, stack slots, registers). Follow every pointer and mark everything you can reach. Then walk the whole heap and free everything unmarked.
6. Most objects die very young. Measured across many programs, the large majority of allocations become garbage almost immediately. This observation is called the generational hypothesis.
7. So a **generational collector** splits the heap. New objects go in a small young area, collected often and quickly. Objects that survive a few collections are promoted to an old area, collected rarely.
8. A **GC pause** is the time the program is stopped so the collector can work without objects moving under its feet. Modern collectors do most of the work concurrently and stop the world only briefly.

9. Rust's answer removes the question. Every value has exactly one **owner**. When the owner goes out of scope, the value is freed. You can lend a value out as a **borrow**, and the compiler checks that no borrow outlives the owner.
10. There is no runtime cost at all, because all of this is checked before the program runs. The price is that the compiler rejects programs a human can see are fine, and you must restructure them.

```
fn main() {
    let s = String::from("hello");
    let r = &s;           // borrow, does not take ownership
    println!("{}", s, r); // both usable
}
```

// s dropped here, memory freed

■ 17.4.5 TECHNICAL – the engineer's version

1. malloc and free are C standard library functions, not system calls. They sit on top of brk, sbrk or mmap. glibc uses ptmalloc2; alternatives include jemalloc, tcmalloc and mimalloc, chosen for different fragmentation and threading behaviour.
2. Fragmentation is the memory management failure nobody warns you about: the allocator holds plenty of free bytes, but no single contiguous run large enough for the next request.
3. Reference counting in CPython is the primary mechanism, with a separate cycle detector for the case counting cannot handle. Swift uses ARC, automatic reference counting, inserted by the compiler at compile time, and requires the programmer to mark cycles with weak or unowned.
4. Tracing collectors in production, with real characteristics:

Collector	Runtime	Typical pause target
G1 (default since Java 9)	JVM	tens of milliseconds
ZGC (production, Java 15)	JVM	under 1 ms
Shenandoah	JVM	under 10 ms
Go's collector	Go	under 1 ms since Go 1.8

5. Go's collector is concurrent, tri-colour and non-moving. The Go 1.5 release in August 2015 was the point where sub-10 ms pauses became normal, and Go 1.8 in 2017 removed the stop-the-world stack rescanning that had been the main remaining source of long pauses.
6. ZGC and Shenandoah are concurrent compacting collectors using load or store barriers, so pause time is largely independent of heap size. This is the headline claim, and it holds for pause length, not for total throughput cost.
7. Rust's rules, stated exactly: each value has one owner; there may be either any number of immutable borrows or exactly one mutable borrow at a time, not both; every borrow's lifetime must be contained within the owner's. The checker that enforces this is the borrow checker, rewritten as NLL (non-lexical lifetimes) in the Rust 2018 edition.
8. Escape hatches exist and are honest: Rc<T> and Arc<T> add reference counting when single ownership does not fit, and unsafe blocks let you do what C does, in clearly marked regions.

Approach	Speed	Safety	Predictability
Manual (C)	fastest possible	none, all bugs live	fully predictable
Ref counting	steady small cost	leaks cycles	very predictable
Tracing GC	fast alloc, pauses	memory-safe	unpredictable
Ownership (Rust)	same as manual	memory-safe	fully predictable

9. The honest version: “Rust has no runtime cost” is nearly true but not entirely. Bounds checks on slice indexing remain at runtime unless the optimizer can prove them redundant, and Rc/Arc cost the same as any other reference count.
10. Microsoft reported in 2019 that around 70 percent of the security vulnerabilities it assigned CVEs to were memory safety issues. The Chromium project has published a very similar figure. This is the single strongest argument for the second and third approaches.
11. Tools: `valgrind --leak-check=full`, `AddressSanitizer (-fsanitize=address)`, `heaptrack`, `jcmd GC.heap_info`, `GODEBUG=gctrace=1`, and `tracemalloc` in Python.

■ 17.4.6 WORDS – remember these

1. Stack — automatic space that cleans itself — a contiguous region managed by push and pop, with frame lifetime tied to call depth.
2. Heap — space you ask for and give back — a dynamically managed region served by an allocator with explicit or automatic reclamation.
3. Memory leak — space never given back — allocated memory that remains reachable-but-unused or unreachable-but-unfreed for the process lifetime.
4. Use after free — reading a returned book — dereferencing a pointer to memory already released, a leading cause of exploitable vulnerabilities.
5. Reference counting — keep a tally per object — incrementing and decrementing a per-object counter, freeing at zero, unable to reclaim cycles.
6. Mark and sweep — find what is reachable, free the rest — a tracing algorithm that marks from roots then sweeps unmarked objects.
7. GC pause — the moment everything stops — a stop-the-world phase during which mutator threads are suspended for collector work.
8. Ownership — one owner, checked at compile time — Rust’s affine type discipline enforced by the borrow checker with zero runtime cost.

17.5 Paradigms

■ 17.5.1 PLAIN – in simple words

1. A **paradigm** is a style of organizing a program. It is a habit of thought, not a feature.
2. **Imperative** means you write a sequence of commands that change state. Do this, then this, then this.
3. **Procedural** is imperative plus the idea of grouping commands into named procedures you can call.
4. **Object-oriented** groups data together with the operations on that data into objects.
5. **Functional** treats computation as evaluating functions, avoids changing things in place, and passes functions around as ordinary values.
6. **Declarative** means you describe what you want, not how to get it. SQL and HTML are declarative.
7. **Event-driven** means the program sits waiting and reacts when something happens: a click, a message, a timer.
8. Most real languages support several of these. The paradigm is a choice you make per program, and sometimes per file.

■ 17.5.2 PLAIN – a picture in your head

1. Imagine you want a cup of tea and there are five ways to get one.
2. Imperative: you write out every step yourself. Fill kettle, switch on, wait, pour, add bag, wait three minutes, remove bag.

3. Procedural: you write “boil water”, “brew”, “serve” as three named jobs, then call them in order.
4. Object-oriented: you build a Kettle that knows how to boil itself and a Cup that knows how to be filled. You tell the kettle to boil, not the water.
5. Functional: you write `serve(brew(boil(water)))`. Each step takes something and returns a new something, and nothing is modified in place.
6. Declarative: you write “I want tea, medium strength, with milk” and something else works out the steps.
7. Event-driven: you sit down, and when the kettle clicks off, you react.

Where this comparison breaks: real programs mix all of these in one file, and the boundaries are much blurrier than five clean styles suggest. Also the declarative version only exists because somebody wrote the imperative version underneath it. Declarative never removes the work, it moves it.

■ 17.5.3 PLAIN - a worked example

1. The task: given a list of orders, find the total value of the ones over 100.
2. First in an object-oriented style, in Python.

```
class OrderBook:
    def __init__(self):
        self.orders = []

    def add(self, value):
        self.orders.append(value)

    def total_over(self, limit):
        total = 0
        for v in self.orders:
            if v > limit:
                total += v
        return total

book = OrderBook()
for v in [50, 120, 300, 80]:
    book.add(v)
print(book.total_over(100))    # 420
```

3. Now the same task in a functional style.

```
from functools import reduce

orders = [50, 120, 300, 80]

big = filter(lambda v: v > 100, orders)
total = reduce(lambda a, b: a + b, big, 0)
print(total)    # 420
```

4. Both print 420. The difference is where the state lives.
5. In the first, `self.orders` is a thing that exists, holds state, and is changed by `add`. Behaviour is attached to that thing.
6. In the second there is no thing. There is a list going in and a number coming out, through two functions that each produce a new value.
7. The functional version is easier to test, because there is nothing to set up. The object version is easier to extend when you later need customer names, dates and discounts hanging off each order.

8. Neither is better. They fail differently.

■ 17.5.4 PLAIN - what is really happening inside

1. Object-oriented programming rests on three ideas.
2. **Encapsulation**: the data inside an object is private, and you touch it only through the object's own methods. That way the object can guarantee its own rules stay true.
3. **Inheritance**: a new class can be defined as "an existing class, plus these changes". A SavingsAccount is an Account with an interest rate.
4. **Polymorphism**: many different objects can respond to the same message in their own way. Call `area()` on a Circle and a Square and each does the right thing without the caller knowing which it holds.
5. The honest criticism of inheritance: deep inheritance chains are one of the most reliable ways to make a codebase hard to change.
6. When class E inherits from D inherits from C inherits from B inherits from A, reading one method means reading five files, and changing A silently changes the behaviour of dozens of classes you have never opened.
7. This is why the modern advice, common since the 1994 book Design Patterns by Gamma, Helm, Johnson and Vlissides, is "favour composition over inheritance": hold an object rather than inherit from it.
8. Go took this seriously and has no inheritance at all, only embedding and interfaces. Rust has traits and no class inheritance. Both are widely used, which shows inheritance is not required for large software.
9. Functional programming rests on different ideas.
10. A **pure function** returns the same answer for the same inputs and changes nothing outside itself. It cannot write a file, print, or read the clock.
11. **Immutability** means values are never changed in place. To "change" a list you produce a new list. Nothing you already handed to someone else can shift under them.
12. A **higher-order function** takes a function as an argument or returns one. `map`, `filter` and `reduce` are the three you meet constantly.
13. `map` applies a function to every item and gives a new sequence. `filter` keeps the items passing a test. `reduce` folds a sequence down to one value by combining two at a time.

■ 17.5.5 TECHNICAL - the engineer's version

1. Timeline. Fortran, 1957, John Backus at IBM: imperative. Lisp, 1958, John McCarthy at MIT: the first functional language. Simula 67, 1967, Ole-Johan Dahl and Kristen Nygaard in Norway: the first object-oriented language, and the source of the word "class". Smalltalk, 1972 at Xerox PARC, Alan Kay and colleagues: the language that defined object-oriented as a whole worldview.
2. Alan Kay, who coined the term "object-oriented", said later that he meant messaging between independent objects rather than classes and inheritance, and that C++ and Java were not what he had in mind. Experts genuinely disagree about which reading is the useful one.
3. SOLID, named by Michael Feathers around 2004 from principles Robert Martin collected, is the standard object-oriented design checklist: single responsibility, open-closed, Liskov substitution, interface segregation, dependency inversion. Barbara Liskov stated the substitution principle in a 1987 keynote.
4. Substitution in practice: if Square inherits from Rectangle and Rectangle has `setWidth` and `setHeight`, a Square cannot honour both independently. This is the classic demonstration that "is a" in English is not "is a" in a type system.

5. Functional core concepts with names: referential transparency (an expression can be replaced by its value without changing behaviour), persistent data structures (updates share most of the old structure, so a “copy” is cheap), currying, closures, and algebraic data types.
6. Persistent data structures are what make immutability affordable. Clojure’s vectors use a 32-way branching trie, so an update copies about five small nodes rather than the whole vector.
7. Functional ideas have won in practice even where the paradigm has not. `map` and `filter` are in every mainstream language. Java 8 added lambdas and streams in March 2014. C++11 added lambdas in 2011. Python has had `lambda` since 1994.

Paradigm	Core unit	Example language
Procedural	the procedure	C, 1972
Object-oriented	the object	Smalltalk, 1972
Functional	the pure function	Haskell, 1990
Declarative	the description	SQL, 1974
Logic	the rule	Prolog, 1972

8. Event-driven is a control-flow architecture rather than a paradigm in the same sense. Its engine is a loop that pulls from a queue and dispatches to handlers, which we take apart in section 17.8.

■ 17.5.6 WORDS – remember these

1. Paradigm — a style of organizing code — a coherent model of computation and program structure.
2. Encapsulation — keep the insides private — restricting direct access to state so invariants are enforced by the type’s own operations.
3. Inheritance — a class built on another — subtyping with implementation reuse from a parent class.
4. Polymorphism — one call, many behaviours — dispatch of an operation based on the runtime or static type of its operand.
5. Pure function — same input, same output, no side effects — referentially transparent, with no observable interaction outside its return value.
6. Immutability — never change, always replace — values that cannot be modified after construction, usually with structural sharing for efficiency.
7. Higher-order function — a function taking or returning a function — a function whose domain or codomain includes function types.

17.6 Data structures, properly

■ 17.6.1 PLAIN – in simple words

1. A **data structure** is a way of arranging values in memory so that certain operations are cheap.
2. Every structure is good at some things and bad at others. There is no best one.
3. An **array** is a fixed row of slots, all the same size, side by side. You can jump straight to slot 500.
4. A **dynamic array** is an array that grows when it fills up, by making a bigger one and copying.
5. A **linked list** is a chain: each item holds a value and the address of the next item. Items can be anywhere in memory.
6. A **stack** is last in, first out. Like a pile of plates.
7. A **queue** is first in, first out. Like a queue at a counter.
8. A **deque** is a queue you can push to and pop from at both ends.
9. A **hash table** maps a key to a value with near-instant lookup.

10. A **set** is a collection with no duplicates and fast “is this in here”.
11. A **binary search tree** keeps items in order so you can find, insert and delete in a small number of steps.
12. A **heap** keeps the smallest (or largest) item instantly available.
13. A **graph** is nodes joined by edges. Roads, friendships, dependencies.
14. A **trie** stores strings by shared prefix, so all words starting “car” sit under one path.

■ 17.6.2 PLAIN – a picture in your head

1. Think of ways to store books in a house.
2. An array is a shelf of identical slots, numbered. You want book 40, you walk straight to slot 40. But inserting a book in the middle means shifting every book after it.
3. A linked list is a treasure hunt. Each book has a note saying where the next one is. Inserting is trivial, you just rewrite two notes. Finding book 40 means following 39 notes.
4. A hash table is a set of drawers labelled A to Z, where a rule turns the title into a drawer number. You compute the drawer and open it. If two books land in the same drawer, you keep a small pile there.
5. A binary search tree is a house where each room says “smaller titles that way, larger titles that way”. Twenty questions gets you to any of a million books.
6. A heap is a pile where the lightest book is always on top, and nothing else is sorted at all.

Where this comparison breaks: real memory has a property no shelf has, called locality. Reading one byte pulls its neighbours into cache for free. So an array is far faster than its step count suggests, and a linked list far slower, even when the theory says they are equal. Measured on modern hardware, scanning an array can be several times faster than walking a linked list of the same length.

■ 17.6.3 PLAIN – a worked example

1. Take a hash table holding names against phone numbers.
2. You insert “ada” -> 5551234.
3. The hash function turns the text “ada” into a number. Suppose it gives 3,481,092,776.
4. The table has 8 buckets. Take the hash modulo 8: $3,481,092,776 \bmod 8 = 0$.
5. So “ada” goes into bucket 0, storing both the key and the value.
6. Now insert “bob” -> 5559876. Suppose $\text{hash}(\text{“bob”}) \bmod 8$ is also 0.
7. That is a **collision**. Two different keys, one bucket.
8. Handling it by **chaining**: bucket 0 holds a tiny list, and now that list has two entries. Looking up “bob” finds bucket 0, then compares keys within it.
9. Handling it by **open addressing**: if bucket 0 is taken, try bucket 1, then 2, until an empty one is found. Lookup follows the same walk.

buckets, 8 slots, chaining:

```
0 -> ["ada" 5551234] -> ["bob" 5559876]
1 -> empty
2 -> empty
3 -> ["cy" 5550000]
...
7 -> empty
```

10. As the table fills, collisions get more common and lookups slow down.
11. So the table watches its **load factor**: entries divided by buckets. When that crosses a threshold, typically 0.7 to 0.9, it allocates a bigger table and rehashes everything into it.

12. That one resize is expensive, but it happens rarely, so the average cost of an insert stays constant. This is called amortized constant time.

■ 17.6.4 PLAIN – what is really happening inside

1. An array's superpower is arithmetic. The address of item i is $\text{base} + i * \text{item\_size}$. One multiply, one add, done. That is why access is constant time regardless of size.
2. A dynamic array grows by a factor, not by a fixed amount. Typically it doubles, or grows by 1.5 times. Growing by one each time would make appending n items cost n -squared work in total.
3. Because it doubles, appending n items costs about $2n$ copies overall, so each append averages constant time even though some individual appends are expensive.
4. A linked list's superpower is that inserting needs no shifting. Its weakness is that every item costs an extra pointer of memory and lives at an unpredictable address, which defeats the CPU cache.
5. A stack and a queue are usually not separate structures at all. They are restricted interfaces on top of an array or a linked list.
6. A binary search tree can degenerate. Insert 1, 2, 3, 4, 5 in order into a plain tree and you get a straight line: a linked list with extra steps, and lookups become linear.
7. A **balanced tree** fixes this by rotating itself after inserts to keep the height near the logarithm of the item count. AVL trees (1962) and red-black trees (1978) are the two you meet.
8. A heap is stored as a plain array with an implicit shape rule: the children of index i sit at $2i+1$ and $2i+2$. No pointers at all.
9. A trie stores one character per edge. All words sharing a prefix share a path, so lookup costs the length of the word and not the size of the dictionary.

■ 17.6.5 TECHNICAL – the engineer's version

1. Complexities below are average case unless marked. n is the number of elements, k the key length.

Structure	Access	Search	Insert
Array	$O(1)$	$O(n)$	$O(n)$
Dynamic array	$O(1)$	$O(n)$	$O(1)$ amortized end
Linked list	$O(n)$	$O(n)$	$O(1)$ at a known node
Stack, queue, deque	$O(1)$ ends	$O(n)$	$O(1)$
Hash table	n/a	$O(1)$ avg, $O(n)$ worst	$O(1)$ avg
Balanced BST	$O(\log n)$	$O(\log n)$	$O(\log n)$
Binary heap	$O(1)$ min	$O(n)$	$O(\log n)$
Trie	n/a	$O(k)$	$O(k)$

Structure	Delete	Ordered	Extra space
Array	$O(n)$	no	none
Dynamic array	$O(n)$	no	up to 100 percent
Linked list	$O(1)$ at node	no	1 pointer each
Hash table	$O(1)$ avg	no	buckets plus slack
Balanced BST	$O(\log n)$	yes	2 pointers each
Binary heap	$O(\log n)$	partial	none
Trie	$O(k)$	yes, by prefix	high

2. Real implementations. C++ `std::vector` is a dynamic array, `std::map` is a red-black tree, `std::unordered_map` is a hash table with chaining. Java `ArrayList` is a dynamic array, `HashMap`

uses chaining and converts a bucket to a red-black tree once it holds 8 or more entries, a change made in Java 8 in 2014 to blunt hash-collision denial-of-service attacks.

3. Python dict has been insertion-ordered since CPython 3.6 (an implementation detail then) and by language guarantee since 3.7 in 2018. It uses open addressing with a compact index array.
4. Go maps are hash tables with 8-slot buckets and randomized iteration order, deliberately, so that nobody writes code depending on the order.
5. Hash functions in production: SipHash-1-3 in Python and Rust's default hasher, chosen for resistance to hash-flooding attacks rather than raw speed; FNV-1a and xxHash where speed matters more; wyhash in several newer maps.
6. Python's `hash()` for strings is randomized per process by default since Python 3.3 in 2012, after the 2011 hash-collision denial-of-service disclosure. Set `PYTHONHASHSEED` to make it repeatable.
7. Cache behaviour dominates the constant factors. A cache line is 64 bytes on x86-64 and on Apple silicon. An array of 4-byte integers gets 16 per line for free. A linked list of nodes scattered across the heap gets one useful item per cache miss, and a main-memory miss costs on the order of 60 to 100 nanoseconds while an L1 hit costs about 1 nanosecond.
8. Graph representations: adjacency list uses $O(V + E)$ space and is right for sparse graphs; adjacency matrix uses $O(V^2)$ and gives $O(1)$ edge lookup, right only for dense graphs.
9. Tools: `sys.getsizeof` in Python, `jol` for Java object layout, `perf stat -e cache-misses` on Linux, and `heaptrack` for allocation profiles.

■ 17.6.6 WORDS - remember these

1. Array — numbered slots side by side — contiguous storage with $O(1)$ indexed access by address arithmetic.
2. Dynamic array — an array that grows — geometric reallocation giving amortized $O(1)$ append.
3. Linked list — a chain of items — nodes with pointers, $O(1)$ splice, poor cache locality.
4. Hash table — compute the drawer, open it — key-to-bucket mapping with collision resolution by chaining or open addressing.
5. Collision — two keys, one bucket — distinct keys hashing to the same index, unavoidable by the pigeonhole principle.
6. Load factor — how full it is — entries divided by buckets, the trigger for resize, typically 0.7 to 0.9.
7. Balanced tree — a tree that stays short — a BST with rebalancing keeping height $O(\log n)$, such as AVL or red-black.
8. Amortized — expensive rarely, cheap usually — the average cost per operation over a worst-case sequence, not a probabilistic average.

17.7 Algorithms and Big-O

■ 17.7.1 PLAIN - in simple words

1. An **algorithm** is a finite recipe for turning an input into an answer.
2. The same job usually has many algorithms, and they differ enormously in how the work grows as the input grows.
3. **Big-O notation** is a short way of saying how the work grows.
4. It ignores constants and small terms on purpose. It answers “what happens when the input gets ten times bigger”, not “how many milliseconds”.
5. $O(1)$ means the work does not depend on the size at all.
6. $O(n)$ means ten times the data takes ten times as long.

7. $O(n^2)$ means ten times the data takes a hundred times as long.
8. $O(\log n)$ means doubling the data adds one step. This is the good one.
9. $O(2^n)$ means adding one item doubles the time. This is the one that kills you.
10. Big-O is about scale, not speed. For 10 items, anything works. For 10 million, the choice decides whether the program finishes today.

■ 17.7.2 PLAIN – a picture in your head

1. You are looking for a name in a phone book of one million entries.
2. Reading every page from the front is **linear search**, $O(n)$. Worst case, a million looks.
3. Opening the middle, deciding which half, and repeating is **binary search**, $O(\log n)$. Twenty looks, because 2^{20} is just over a million.
4. Twenty against a million. That is the whole reason algorithms matter.
5. Now imagine comparing every entry with every other entry to find duplicates. That is $O(n^2)$, which is a million million comparisons. At a billion comparisons a second, about 11 days.
6. Sorting first and then scanning once is $O(n \log n)$, which is about 20 million steps, or a fiftieth of a second.

Where this comparison breaks: binary search only works because the phone book is already sorted. Big-O hides the cost of getting there. It also hides constants, and constants matter in the real world: an $O(n \log n)$ algorithm with a huge constant can lose to an $O(n^2)$ one for every input you will ever actually see. This is exactly why library sorts switch to insertion sort for small chunks.

■ 17.7.3 PLAIN – a worked example

1. Assume a rough one hundred million simple operations per second, which is a reasonable figure for interpreted code and pessimistic for compiled code. These figures are approximate and meant for ranking, not for prediction.

Complexity	Name	n done in about 1 s
$O(1)$	constant	any size
$O(\log n)$	logarithmic	effectively unlimited
$O(n)$	linear	100,000,000
$O(n \log n)$	linearithmic	about 4,000,000
$O(n^2)$	quadratic	about 10,000
$O(n^3)$	cubic	about 460
$O(2^n)$	exponential	about 26
$O(n!)$	factorial	about 11

2. Read the bottom two rows again. An exponential algorithm handles 26 items in a second. Twenty-six. Not 26 thousand.
3. Now dynamic programming, with the classic example. Fibonacci numbers, where each is the sum of the two before it.

```
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

4. This is correct and it is $O(2^n)$. `fib(40)` makes about 331 million calls, because `fib(38)` is computed twice, `fib(37)` three times, and so on.

5. The insight of **dynamic programming**: the sub-problems repeat, so solve each one once and write the answer down.

```
def fib(n, seen={}):
    if n < 2:
        return n
    if n not in seen:
        seen[n] = fib(n - 1, seen) + fib(n - 2, seen)
    return seen[n]
```

6. That change makes it $O(n)$. `fib(40)` now takes 40 additions instead of 331 million calls. The algorithm did not get cleverer. It stopped repeating itself.
7. Writing answers down as you go is called memoization, and it is the top-down form. Filling a table from the smallest case upward is the bottom-up form. They compute the same thing.

■ 17.7.4 PLAIN - what is really happening inside

1. Recursion means a function calls itself. Each call gets its own **stack frame**: a slice of the stack holding its arguments, its local variables and the address to return to.

```
main          -> frame 1
  fib(4)       -> frame 2
    fib(3)     -> frame 3
      fib(2)   -> frame 4
        fib(1) -> frame 5   returns 1, frame popped
          fib(0) -> frame 5   returns 0, frame popped
```

2. Frames stack up as you go deeper and are popped as each call returns. If you never stop going deeper, the stack runs out. That is a stack overflow.
3. Default limits: CPython refuses at about 1,000 nested calls by default; the main thread stack on Linux is typically 8 MB, which is thousands of frames for compiled code.
4. Now sorting, in order of how they work.
5. **Bubble sort**: repeatedly walk the list swapping neighbours that are out of order. $O(n^2)$. Never use it, but understand it.
6. **Insertion sort**: take each item and slide it back into its place among the already-sorted front. $O(n^2)$ in general, but $O(n)$ if the data is nearly sorted, and very fast for small lists.
7. **Merge sort**: split in half, sort each half, then merge the two sorted halves in one pass. Always $O(n \log n)$. Needs extra space. Stable, meaning equal items keep their original order.
8. **Quicksort**: pick a pivot, move everything smaller left and larger right, then sort each side. $O(n \log n)$ on average, $O(n^2)$ if you keep picking bad pivots. Sorts in place, and is usually the fastest in practice.
9. Greedy against exhaustive. A **greedy** algorithm takes the best-looking step at each moment and never reconsiders. It is fast and sometimes optimal.
10. Giving change with British coins greedily always gives the fewest coins. With a hypothetical set of 1, 3 and 4, greedy makes 6 as $4+1+1$, three coins, while the best answer is $3+3$, two coins. Greedy is a gamble that needs proof.
11. **Exhaustive** search tries every possibility. It always finds the best answer and it is usually exponential.

■ 17.7.5 TECHNICAL - the engineer's version

1. Formally, $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. Big-O is an upper bound. Big-Omega is a lower bound, and Big-Theta is both.

2. Comparison sorting has a proven lower bound of $\Omega(n \log n)$ comparisons, because a decision tree with $n!$ leaves has height at least $\log_2(n!)$. Counting sort and radix sort beat it by not comparing.
3. What real libraries actually ship:

Library	Algorithm	Since
Python sorted	Timsort	2002, Tim Peters
Java objects	Timsort	Java 7, 2011
Java primitives	dual-pivot quicksort	Java 7, 2011
C++ <code>std::sort</code>	introsort	Musser, 1997
Go <code>slices.Sort</code>	pattern-defeating quicksort	Go 1.19, 2022
Rust <code>sort_unstable</code>	quicksort variant	in-place, no alloc

4. Timsort was written by Tim Peters for CPython in 2002. It finds existing sorted runs, extends short ones with binary insertion sort, and merges them under strict invariants. It is stable and it is $O(n)$ on already-sorted data. Java adopted it for object arrays in Java 7 in 2011.
5. Introsort, described by David Musser in 1997, starts as quicksort, switches to heapsort if recursion gets too deep, and finishes with insertion sort on small partitions. It gets quicksort's speed with heapsort's $O(n \log n)$ worst-case guarantee.
6. Dual-pivot quicksort was contributed by Vladimir Yaroslavskiy in 2009 and is used for Java primitive arrays, where stability is meaningless because equal integers are indistinguishable.
7. Dynamic programming was named by Richard Bellman around 1953 at RAND. He chose the words partly because they sounded impressive to a defence secretary who disliked mathematical research. It requires two properties: optimal substructure and overlapping subproblems.
8. Standard dynamic programming problems and their complexities: 0/1 knapsack $O(nW)$, longest common subsequence $O(nm)$, edit distance $O(nm)$, Floyd-Warshall all-pairs shortest paths $O(V^3)$.
9. Greedy algorithms that are provably optimal: Dijkstra's shortest path (1959, Edsger Dijkstra), Kruskal's and Prim's minimum spanning trees, and Huffman coding (David Huffman, 1952). Each has a proof, usually via a matroid or an exchange argument.
10. Tools: `timeit` in Python, `JMH` for Java, `perf record` and `perf report` on Linux, `cargo bench` and `pprof` for Go.

■ 17.7.6 WORDS – remember these

1. Algorithm — a finite recipe that terminates — a well-defined procedure mapping input to output in finite steps.
2. Big-O — how the work grows with size — an asymptotic upper bound on a cost function, ignoring constants and lower-order terms.
3. Binary search — halve the range each time — $O(\log n)$ search on a sorted sequence with random access.
4. Stable sort — equal items keep their order — a sort preserving the relative order of records with equal keys.
5. Stack frame — one call's private space — the activation record holding arguments, locals and the return address.
6. Memoization — write the answer down — caching results of pure function calls keyed by arguments.

7. Greedy — best-looking step, never reconsider — a locally optimal choice strategy, globally optimal only where provable.

17.8 Concurrency and parallelism

■ 17.8.1 PLAIN – in simple words

1. **Concurrency** is dealing with many things at once. **Parallelism** is doing many things at once.
2. One cook juggling three pans is concurrent. Three cooks each with one pan is parallel.
3. A single-core machine can be concurrent and cannot be parallel. Concurrency is about structure. Parallelism is about hardware.
4. A **process** is a running program with its own private memory. Two processes cannot touch each other's data by accident.
5. A **thread** is a strand of execution inside a process. Threads in one process share all memory, which is fast and dangerous.
6. When two threads touch the same data at the same time and the result depends on who got there first, that is a **race condition**.
7. A **lock** is a token. Only the thread holding it may touch the protected data. Everyone else waits.
8. If two threads each hold a lock the other needs, both wait forever. That is **deadlock**.
9. Concurrency is genuinely hard because the bugs depend on timing, appear once in ten thousand runs, and vanish when you add a print statement.

■ 17.8.2 PLAIN – a picture in your head

1. Picture a shared kitchen notebook holding a running total.
2. Two people each have twenty receipts to add.
3. Adding one receipt takes three actions: read the total, work out the new total in your head, write it back.
4. Person A reads 41. Before A writes, person B also reads 41. A works out 42 and writes it. B works out 42 and writes it.
5. Two receipts were added. The total went up by one. Nothing was broken, nobody made a mistake, and the answer is wrong.
6. A lock is a rule: pick up the pen before reading, put it down after writing. Only one person can hold the pen.
7. Deadlock: A holds the pen and needs the calculator; B holds the calculator and needs the pen. Neither will let go. The kitchen stops.

Where this comparison breaks: people notice when they are stuck. Threads do not. Also, real processors and compilers may reorder your reads and writes for speed, so the order you wrote is not always the order that happens. That is a whole extra layer of difficulty the notebook has no equivalent for.

■ 17.8.3 PLAIN – a worked example

1. Here is the notebook, in Python, with two real threads.

```
import threading

counter = 0

def bump():
    global counter
    for _ in range(200000):
        counter += 1
```

```
t1 = threading.Thread(target=bump)
t2 = threading.Thread(target=bump)
t1.start(); t2.start()
t1.join(); t2.join()
print(counter)      # expected 400000, often less
```

2. `counter += 1` looks like one action. It is three: load, add, store.
3. Here is one bad interleaving, step by step.

step	thread A	thread B	counter
1	load 41		41
2		load 41	41
3	add -> 42		41
4		add -> 42	41
5	store 42		42
6		store 42	42

4. Two increments happened. The counter rose by one. This is a **lost update**.
5. The fix is a lock around the three steps.

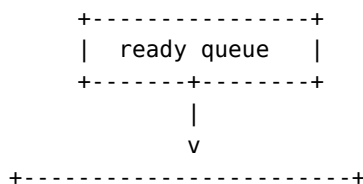
```
lock = threading.Lock()

def bump():
    global counter
    for _ in range(200000):
        with lock:
            counter += 1
```

6. Now the three steps are one indivisible unit. The section of code the lock protects is called the **critical section**.
7. The cost is real: this version is several times slower, because threads spend time waiting instead of working.

■ 17.8.4 PLAIN – what is really happening inside

1. A **mutex** is a lock allowing exactly one holder. The name is short for mutual exclusion. Acquiring an uncontended mutex is cheap, tens of nanoseconds. Acquiring a contended one means the operating system puts your thread to sleep and wakes it later, costing microseconds.
2. An **atomic operation** is one the hardware guarantees cannot be split. `lock_xadd` on x86 does load-add-store as a single unit. For a simple counter this is faster than a lock and needs no waiting.
3. Deadlock needs four conditions at once, described by Edward Coffman in 1971: mutual exclusion, hold and wait, no preemption, and circular wait. Break any one and deadlock becomes impossible.
4. The usual practical fix is lock ordering: every thread acquires locks in the same global order, which breaks circular wait.
5. **Async and await** are a different approach entirely. Instead of many threads, you have one thread and an **event loop**.
6. The loop holds a queue of ready tasks. It runs one until that task hits something slow, like waiting for the network. The task says “I am waiting” and hands control back. The loop runs another task.



```

| run task until await |
+-----+
      |
waiting? | yes -> park it, register callback
          | no  -> task finished, drop it
          v
      back to ready queue when the wait completes

```

7. So one thread can juggle ten thousand network connections, because almost all of them are waiting almost all of the time.
8. This is concurrency without parallelism, and without locks, because only one thing runs at a time. The trade is that any task doing real computation blocks everything.
9. A **coroutine** is a function that can pause in the middle and resume later, keeping its local variables. That is the machinery `await` is built on.
10. **Message passing** avoids shared memory entirely. Two workers never touch the same data. They send copies to each other through a channel. Nothing is shared, so nothing needs locking.
11. The **actor model** takes that further: an actor is an isolated unit with private state and a mailbox. It only reacts to messages. Erlang and Elixir are built entirely this way.

■ 17.8.5 TECHNICAL – the engineer’s version

1. Processes versus threads on Linux: both are tasks to the kernel, created by `clone` with different sharing flags. A process context switch costs more because it changes the page tables and flushes TLB entries. Rough figures on modern x86-64: thread switch about 1 to 2 microseconds, process switch somewhat more.
2. Memory ordering is a specification, not an implementation detail. The Java Memory Model was rewritten in JSR-133 for Java 5 in 2004. The C++11 memory model, adopted in 2011, defines `memory_order_relaxed`, `acquire`, `release` and `seq_cst`. x86-64 gives total store order almost for free; ARM and RISC-V are weakly ordered and need explicit barriers.
3. Amdahl’s law, stated by Gene Amdahl in 1967: if a fraction p of a program is parallelizable, maximum speedup on N cores is $1 / ((1 - p) + p/N)$. With $p = 0.95$, the ceiling is 20 times no matter how many cores you buy.
4. Concurrency models and their origins:

Model	Origin	Used by
Threads and locks	1960s, POSIX 1995	C, C++, Java
CSP channels	Hoare, 1978	Go, <i>occam</i>
Actors	Hewitt, 1973	Erlang, Akka
Event loop	Unix <code>select</code> , 1983	Node.js, <i>asyncio</i>
Ownership + Send/Sync	Rust, 2015	Rust

5. Go’s goroutines are user-space threads multiplexed onto operating-system threads by the runtime, starting at an 8 KB stack that grows as needed, which is why a million goroutines is routine and a million OS threads is not.
6. Erlang, created by Joe Armstrong and colleagues at Ericsson from 1986, runs millions of isolated processes with no shared memory and a “let it crash” supervision model. The AXD301 switch built on it is the source of the often-quoted nine nines availability figure, which is a marketing claim about one deployment and not a property of the language.
7. The Global Interpreter Lock in CPython is a single mutex that ensures only one thread executes Python bytecode at a time. It makes reference counting safe and single-threaded code fast. It

means CPU-bound Python threads do not scale across cores. Input/output releases it, so I/O-bound threading works.

8. This is changing, with dates. PEP 703 was accepted in July 2023. Python 3.13, released October 2024, shipped an optional experimental free-threaded build. Python 3.14, released 7 October 2025, moved that build to officially supported, though it is still not the default. Established fact: the option exists. Active question: what the single-threaded performance cost settles at, and how quickly C extensions adapt.
9. Rust prevents data races at compile time using the Send and Sync marker traits plus the borrow checker. It does not prevent deadlock, which is a liveness property, and the documentation says so plainly.
10. Tools: ThreadSanitizer (`-fsanitize=thread`), Helgrind under Valgrind, `perf sched`, Go's `-race` flag, and `jstack` for JVM thread dumps.

■ 17.8.6 WORDS – remember these

1. Concurrency — dealing with many things at once — composing independently executing tasks, whether or not they overlap in time.
2. Parallelism — doing many things at once — simultaneous execution on multiple hardware execution units.
3. Race condition — the answer depends on timing — unsynchronized access where at least one access is a write.
4. Mutex — a token only one may hold — a mutual exclusion primitive serializing entry to a critical section.
5. Deadlock — everyone waiting forever — a cycle of threads each holding a resource another needs, requiring all four Coffman conditions.
6. Atomic — cannot be split in half — an operation the hardware performs indivisibly, such as compare-and-swap.
7. Event loop — one worker, many waits — a dispatcher polling readiness and running callbacks or resuming coroutines on a single thread.
8. GIL — one Python thread at a time — CPython's global interpreter lock, optional to disable since Python 3.13 and supported since 3.14.

17.9 Errors and correctness

■ 17.9.1 PLAIN – in simple words

1. Things go wrong. Files are missing, networks drop, users type nonsense.
2. There are two main ways a function tells you it failed.
3. **Return codes**: the function returns a value meaning “it did not work”, and the caller must check it. Simple, explicit, and easy to ignore.
4. **Exceptions**: the function throws, and control jumps out to whoever declared they would handle it. Hard to ignore, but the jump is invisible when reading the code.
5. A **panic** is a third thing: the program has found a state that should be impossible and stops rather than continuing wrongly.
6. The rule of thumb: expected problems are errors, and you handle them. Impossible states are bugs, and you crash on them.
7. An **assertion** is a statement of something you believe must be true. If it is false, stop immediately.
8. A **test** is code that runs your code and checks the answer, automatically, every time.

■ 17.9.2 PLAIN – a picture in your head

1. Think about a parcel delivery.
2. A return code is the driver leaving a card saying “not delivered, reason 3”. You have to read the card. Nothing forces you to.
3. An exception is the driver pulling a fire alarm that keeps sounding until someone on some floor takes responsibility. If nobody does, the whole building is evacuated, which is the program crashing.
4. A panic is the driver finding the building on fire and refusing to enter.
5. An assertion is a sign on the loading bay saying “this door must be unlocked during working hours”. Nobody expects it to be locked. If it is, something deeper is wrong and continuing makes it worse.

Where this comparison breaks: fire alarms are rare and exceptions are not. In many programs, a file not existing is completely normal and gets thrown dozens of times a second, which is why some languages treat exceptions as a bad fit for ordinary control flow and use return values instead.

■ 17.9.3 PLAIN – a worked example

1. Four languages, the same job: open a file that may not be there.

```
FILE *f = fopen("data.txt", "r");
if (f == NULL) {
    perror("cannot open");
    return 1;
}

try:
    f = open("data.txt")
except FileNotFoundError as e:
    print("cannot open:", e)

f, err := os.Open("data.txt")
if err != nil {
    return fmt.Errorf("cannot open: %w", err)
}

let f = match File::open("data.txt") {
    Ok(f) => f,
    Err(e) => return Err(e.into()),
};
```

2. C returns a null pointer. Nothing stops you using it, and the crash comes later somewhere else.
3. Python throws. If you write no `try`, the program stops with a stack trace pointing at the exact line.
4. Go returns two values and the second is the error. You can ignore it, but the compiler complains about the unused variable and every linter shouts.
5. Rust returns a `Result`, which is either `Ok` or `Err`, and you cannot get the file out without dealing with both. Rust also has the `?` operator, which is shorthand for exactly the match above.
6. Now a real test, in Python’s built-in style.

```
def slugify(text):
    return text.strip().lower().replace(" ", "-")

def test_slugify_basic():
    assert slugify("Hello World") == "hello-world"
```

```
def test_slugify_trims():
    assert slugify(" Spaced Out ") == "spaced-out"
```

7. Run it with pytest. Two tests, two green ticks, in about 10 milliseconds.
8. Now a property-based test, which invents inputs instead of using yours.

```
from hypothesis import given, strategies as st

@given(st.text())
def test_slug_has_no_whitespace(s):
    assert not any(c.isspace() for c in slugify(s))
```

9. This one fails, quickly, with the input "\t". A tab is whitespace, it is not a space, and `replace(" ", "-")` never touches it.
10. That is the value of property-based testing in one example. It found a real bug from a rule, not from a guess about which inputs to try.

■ 17.9.4 PLAIN – what is really happening inside

1. Throwing an exception is not a normal return. The runtime walks back up the stack, frame by frame, looking for a handler that matches, running cleanup code for each frame it discards. This is called stack unwinding.
2. That is why exceptions are cheap when not thrown and expensive when thrown. Modern implementations use zero-cost tables, so the non-throwing path has no overhead at all and the throwing path costs microseconds.
3. Assertions are usually compiled out of release builds. In C, NDEBUG removes them. This means an assertion must never contain something the program needs to do, because in production it will not happen.
4. **Defensive programming** means checking inputs at the boundary of your code, where data arrives from outside, and trusting them inside.
5. The mistake is checking everywhere. Every internal function re-validating makes code longer, slower, and harder to read, without making it safer.
6. Testing comes in layers.
7. A **unit test** tests one function in isolation, in milliseconds, with no database and no network.
8. An **integration test** tests several parts together, usually with a real database, in seconds.
9. An **end-to-end test** drives the whole system the way a user would, in tens of seconds to minutes, and is the most fragile.
10. A **property-based test** states a rule that should always hold and lets the tool generate hundreds of inputs trying to break it.
11. The rough guidance is a pyramid: many unit tests, fewer integration tests, very few end-to-end tests. The reason is cost and flakiness, not purity.

■ 17.9.5 TECHNICAL – the engineer's version

1. Error strategies by language, stated exactly:

Language	Mechanism	Ignorable
C	int or null return, errno	yes, silently
Java	checked and unchecked	checked, no
Python	exceptions only	yes, uncaught crashes
Go	explicit error value	yes, linters object

Language	Mechanism	Ignorable
Rust	<code>Result<T, E></code>	<code>no, #[must_use]</code>

2. Java's checked exceptions, present since Java 1.0 in 1996, force the caller to declare or handle them. Experts genuinely disagree about this: supporters say it makes failure part of the type signature; critics, including the C# design team who deliberately left them out, say it produces empty catch blocks and throws `Exception` on every method.
3. Rust distinguishes recoverable errors (`Result`) from unrecoverable ones (`panic!`). A panic unwinds by default and can be configured to abort with `panic = "abort"` in Cargo, which produces smaller binaries.
4. Go added error wrapping with `%w` and the `errors.Is` and `errors.As` functions in Go 1.13, released September 2019.
5. Exception implementation: Itanium C++ ABI zero-cost exceptions use side tables consulted only during unwinding, so entering a `try` block costs nothing. `Setjmp/longjmp` implementations cost on entry instead and are largely historical.
6. Testing frameworks and origins: JUnit, Kent Beck and Erich Gamma, 1997, which defined the xUnit shape copied by almost everything since. QuickCheck, the first property-based tester, by Koen Claessen and John Hughes for Haskell in 2000. Hypothesis brought it to Python from 2013.
7. Coverage is a diagnostic, not a target. 100 percent line coverage proves every line ran, not that any behaviour was checked. Branch coverage is stricter and still not proof. Tools: `coverage.py`, `JaCoCo`, `go test -cover`, `cargo llvm-cov`.
8. Formal methods sit beyond testing: TLA+, created by Leslie Lamport in the 1990s, is used by AWS to check distributed protocol designs, and the seL4 microkernel has a machine-checked functional correctness proof, completed in 2009. These are real and they are expensive, so they are used where failure is catastrophic.
9. Tools: `pytest -x --lf`, `go test -race ./...`, `cargo test`, `mvn verify`, and mutation testers such as `mutmut` and `PIT`, which change your code deliberately to see whether any test notices.

■ 17.9.6 WORDS – remember these

1. Return code — a value meaning it failed — an out-of-band or sentinel result the caller must inspect.
2. Exception — a jump to whoever handles it — a non-local transfer of control with stack unwinding and frame cleanup.
3. Panic — stop, this should be impossible — an unrecoverable fault that aborts or unwinds rather than returning.
4. Assertion — a claim that must be true — a runtime check of an invariant, typically compiled out in release builds.
5. Unit test — check one piece alone — an isolated, fast, deterministic test of a single unit of behaviour.
6. Property-based test — state a rule, let it hunt — automated generation of inputs against an invariant, with shrinking to a minimal failing case.
7. Coverage — how much code the tests ran — the proportion of lines or branches executed, a diagnostic rather than a goal.

17.10 A tour of the major languages

■ 17.10.1 PLAIN – in simple words

1. There are about twenty languages you will actually meet, and they sort into a few families.
2. **Systems languages** compile straight to machine code and give you control of memory: C, C++, Rust, and Go in a gentler way.
3. **Managed languages** run on a virtual machine with automatic memory: Java, C#, Kotlin.
4. **Scripting languages** run from source with no separate build step: Python, JavaScript, Ruby, PHP.
5. **Platform languages** exist mainly to build for one place: Swift for Apple, Kotlin for Android.
6. **Specialist languages** do one job extremely well and are bad at everything else: SQL for data, Bash for gluing commands, R and MATLAB for numbers.
7. Underneath all of them is **assembly**, one line per machine instruction.
8. For each language below: the year, the person, what it was made for, what it is genuinely good at, its real weakness, and where you meet it in 2026.

■ 17.10.2 PLAIN – a picture in your head

1. Think of vehicles rather than a ladder of quality.
2. C is a motorcycle: light, fast, nothing between you and the road, and no protection when you fall.
3. Java is a coach: heavy, slow to start, carries a hundred passengers reliably for twenty years.
4. Python is a bicycle: anyone can ride it today, it goes almost anywhere, and it will not win a race.
5. Rust is a motorcycle with a mandatory riding test that you must pass before the engine will start.
6. SQL is a train: it goes only where the rails go, and on those rails nothing beats it.
7. Nobody asks which vehicle is best. They ask what the journey is.

Where this comparison breaks: vehicles cannot be combined, and languages are combined constantly. A single 2026 web product routinely uses TypeScript in the browser, Go on the server, SQL in the database, Python for the model, and C underneath all four.

■ 17.10.3 PLAIN – a worked example

1. The systems languages, with a sample each.
2. **C**. 1972, Dennis Ritchie at Bell Labs. Made to rewrite the Unix operating system in something portable. Genuinely good at: direct hardware access, a near-zero runtime, and being the language every other language talks to. Main weakness: no memory safety of any kind. In 2026 you meet it in the Linux kernel, embedded firmware, and inside the runtime of nearly every language in this list.

```
| for (int i = 0; i < n; i++) sum += a[i];
```

3. **C++**. Begun 1979 as “C with Classes” by Bjarne Stroustrup at Bell Labs, released 1985. Made to add Simula’s classes to C for large simulations. Good at: zero-overhead abstraction, the fastest code you can still structure. Weakness: enormous, with forty years of features that never leave. In 2026: game engines such as Unreal, browsers, databases, and trading systems. The C++26 standard replaced C++23 in March 2026.

```
| std::sort(v.begin(), v.end());  
| auto n = std::count_if(v.begin(), v.end(), is_big);
```

4. **Go**. Announced November 2009 by Robert Griesemer, Rob Pike and Ken Thompson at Google. Made for large server software with fast builds. Good at: simplicity you can hold in your head, built-in concurrency, one static binary with no dependencies, and compiles measured in seconds.

Weakness: deliberately plain, and error handling is repetitive on purpose. In 2026: Docker, Kubernetes, and most cloud infrastructure. Go 1.26 landed 10 February

2026.

```
go worker(ch)           // start a goroutine
msg := <-ch             // receive from a channel
```

5. **Rust.** Started 2006 as Graydon Hoare's personal project, sponsored by Mozilla from 2009, version 1.0 on 15 May 2015. Made for systems work without memory bugs. Good at: memory safety with no garbage collector, and concurrency the compiler checks. Weakness: a steep learning curve and slow compiles. In 2026: Firefox, Cloudflare, Android system components, and Linux kernel drivers, since Rust support was merged into Linux 6.1 in December

2022. Stable release in August 2026 is 1.97.

```
let names: Vec<String> =
    people.iter().map(|p| p.name.clone()).collect();
```

■ 17.10.4 PLAIN - what is really happening inside

1. The managed and scripting languages.
2. **Java.** 1995, James Gosling at Sun Microsystems, from a 1991 project called Oak. Made so one compiled program could run on any machine with a virtual machine. Good at: very large long-lived server systems, superb tooling and profilers, mature garbage collectors. Weakness: verbose, slow to start, heavy on memory. In 2026: banking, enterprise back-ends, and Android underneath. Java 25 is the current long-term-support release, from 16 September 2025.

```
List<String> big = names.stream()
    .filter(n -> n.length() > 3).toList();
```

3. **C#.** Announced 2000, released 2002 with .NET 1.0, designed by Anders Hejlsberg at Microsoft. Made as Microsoft's answer to Java. Good at: a clean modern design, LINQ for querying collections, and it shipped async/await in C# 5 in 2012, which most other languages then copied. Weakness: was tied to Windows until .NET Core in 2016. In 2026: Unity games, Windows desktop, and enterprise services on .NET 10, released November 2025.

```
var big = names.Where(n => n.Length > 3).ToList();
```

4. **Kotlin.** Announced July 2011 by JetBrains, version 1.0 in February 2016. Made as a less verbose Java that runs on the same virtual machine. Good at: null safety in the type system, coroutines, and complete Java interop. Weakness: still bound to the JVM tooling world, and compiles more slowly than Java. In 2026: Google named it the preferred Android language in 2019, and most new Android code is Kotlin.

```
val name: String? = user.name
println(name?.length ?: 0)
```

5. **Swift.** Announced June 2014 at Apple's developer conference, led by Chris Lattner. Made to replace Objective-C. Good at: safe and modern while compiling to fast native code, with automatic reference counting instead of a collector. Weakness: in practice you meet it almost only on Apple platforms. In 2026: essentially all new iOS and macOS applications.

```
let big = names.filter { $0.count > 3 }
```

6. **Python.** First release February 1991, by Guido van Rossum at CWI in the Netherlands. Made as a readable scripting language for people who found the alternatives painful. Good at: readabil-

ity, gluing other systems together, and an unmatched scientific and machine-learning ecosystem. Weakness: slow, and packaging has been a long-running sore point. In 2026: machine learning, data work, automation, and back-ends. Version 3.14 arrived 7 October 2025.

```
| big = [n for n in names if len(n) > 3]
```

7. **JavaScript.** 1995, Brendan Eich at Netscape, with the first version written in about ten days in May 1995. Made to add small interactions to web pages. Good at: it runs everywhere, in every browser on earth, with the largest package ecosystem in existence. Weakness: weak typing and a set of historical oddities that can never be removed. In 2026: every browser, plus servers through Node.js, Deno and Bun.
8. **TypeScript.** Released 1 October 2012, designed by Anders Hejlsberg and Luke Hoban at Microsoft. It is JavaScript plus a static type system, erased before running. Good at: catching whole classes of mistakes before the code ships. Weakness: another build step, and the types are checked but not enforced at runtime. In 2026 it is the default for serious front-end work. TypeScript 6.0 shipped 23 March 2026, and version 7.0, a rewrite of the compiler in Go for roughly a tenfold speedup, is in preview.

```
| const big: string[] = names.filter(n => n.length > 3);
```

9. **PHP.** 1995, Rasmus Lerdorf, originally a set of tools he called Personal Home Page. Made to put dynamic content into web pages. Good at: the simplest deployment story there is, and universal cheap hosting. Weakness: an inconsistent standard library and a poor security reputation earned in its early years. In 2026: WordPress, which runs a large fraction of all websites, plus Wikipedia and the Laravel framework. PHP 8.5 arrived 20 November 2025.
10. **Ruby.** 1995, Yukihiro Matsumoto in Japan, designed explicitly for programmer happiness. Good at: expressive code and building small domain-specific languages, which is why Rails felt like magic in 2004. Weakness: slow, and the ecosystem shrank after about 2015. In 2026: GitHub, Shopify, and a large body of Rails applications.

```
| big = names.select { |n| n.length > 3 }
```

■ 17.10.5 TECHNICAL – the engineer’s version

1. The specialist languages.
2. **SQL.** Designed 1974 as SEQUEL by Donald Chamberlin and Raymond Boyce at IBM San Jose, implementing Edgar Codd’s 1970 relational model. Declarative: you state the result, the query planner chooses the method. Good at: set operations over large relational data with an optimizer you did not write. Weakness: dialects differ enough that portable SQL is a discipline, and it is not a general-purpose language. Current standard is SQL:2023. Everywhere data is stored in 2026.

```
| SELECT country, COUNT(*) AS n
| FROM users WHERE active GROUP BY country
| HAVING COUNT(*) > 100 ORDER BY n DESC;
```

3. **Bash.** 1989, Brian Fox for the GNU project, as a free replacement for the 1979 Bourne shell. Good at: joining programs with pipes and automating a machine in a few lines. Weakness: quoting rules are genuinely treacherous and error handling is weak, which is why `set -euo pipefail` is standard advice. In 2026: continuous-integration pipelines, container images, and every server login.

```
| set -euo pipefail
| grep -c ERROR /var/log/app.log || echo 0
```

4. **R**. 1993, Ross Ihaka and Robert Gentleman at the University of Auckland, as a free implementation of the S language John Chambers created at Bell Labs in 1976. Good at: statistics, and plotting through ggplot2. Weakness: unusual evaluation semantics and poor performance outside vectorized work. In 2026: academic statistics, biostatistics and epidemiology.
5. **MATLAB**. Written by Cleve Moler in the late 1970s at the University of New Mexico as a friendly front end to the LINPACK and EISPACK Fortran libraries, and sold commercially from 1984 when MathWorks was founded. Good at: matrix and numerical work, and Simulink for control system modelling. Weakness: a paid commercial licence, and 1-based indexing that surprises everyone else. In 2026: control engineering, signal processing, and university courses.
6. **Assembly**. Not one language but one per instruction set. Kathleen Booth is credited with writing the first assembly language, working at Birkbeck College in London from 1947. One mnemonic maps to one machine instruction. Good at: exact control and hand-tuning the few functions that matter. Weakness: not portable at all, and enormous effort per line. In 2026: boot code, cryptographic primitives, and compiler output you read while debugging.

```

mov  eax, [rdi]    ; load a 32-bit value
add  eax, 1        ; add one
mov  [rdi], eax    ; store it back

```

7. Verified summary of origins:

Language	Year	Creator
Assembly	1947	Kathleen Booth
C	1972	Dennis Ritchie
SQL (SEQUEL)	1974	Chamberlin and Boyce
MATLAB	late 1970s	Cleve Moler
C++	1979 / 1985	Bjarne Stroustrup
Bash	1989	Brian Fox
Python	1991	Guido van Rossum
R	1993	Ihaka and Gentleman
Java	1995	James Gosling
JavaScript	1995	Brendan Eich
PHP	1995	Rasmus Lerdorf
Ruby	1995	Yukihiro Matsumoto
C#	2000 / 2002	Anders Hejlsberg
Go	2009	Griesemer, Pike, Thompson
Rust	2010 / 1.0 in 2015	Graydon Hoare
Kotlin	2011 / 1.0 in 2016	JetBrains
TypeScript	2012	Hejlsberg and Hoban
Swift	2014	Chris Lattner

8. Release status checked in August 2026:

Language	Current release	Date
Python	3.14	7 Oct 2025
Java	25 LTS, 26 current	Sep 2025, Mar 2026
Go	1.26	10 Feb 2026
Rust	1.97	Jul 2026
PHP	8.5	20 Nov 2025

Language	Current release	Date
TypeScript	6.0, 7.0 in preview	23 Mar 2026

9. Popularity indexes such as TIOBE, the Stack Overflow developer survey and the GitHub Octoverse report disagree with each other every year, because they measure different things: search volume, respondent self-report, and public repository activity. Treat any single ranking as one weak signal.

■ 17.10.6 WORDS – remember these

1. Systems language — you manage the memory — compiles to native code with direct memory access and no mandatory runtime.
2. Managed language — the runtime manages memory — executes on a virtual machine with automatic memory management.
3. Interop — languages calling each other — a foreign function interface, most often through the C ABI.
4. Static binary — one file, no dependencies — an executable with all libraries linked in, as Go produces by default.
5. Erasure — types vanish before running — compile-time-only types removed before execution, as in TypeScript and Java generics.
6. Dialect — the same language, different rules — vendor-specific extensions to a standard, most visible across SQL implementations.

17.11 How to choose a language for a job

■ 17.11.1 PLAIN – in simple words

1. “Which language is best” is not a real question, because it has no fixed subject. Best at what, for whom, with what deadline.
2. A better question has four parts: what must this program do, where must it run, who will maintain it, and how long must it live.
3. Constraints usually decide it before taste does. An iPhone app is Swift. A browser front end is JavaScript or TypeScript. A database query is SQL.
4. Where nothing forces the choice, the strongest factor is the team. A language nobody on the team knows costs six months of slower work.
5. The second strongest factor is the ecosystem. The right library in an average language beats a great language with nothing written for it.
6. Performance matters far less often than people believe, and far more when it matters.
7. You are also allowed to pick two. Most real systems use several.

■ 17.11.2 PLAIN – a picture in your head

1. Choosing a language is like choosing a material for a building.
2. Nobody argues that steel is better than wood. They ask about the span, the climate, the budget, and who is available to work it.
3. A material also implies its trades. Choosing steel means hiring welders. Choosing Rust means hiring or training Rust programmers.
4. And once the frame is up, changing material means rebuilding. Language choices are cheap on day one and very expensive on day 800.

Where this comparison breaks: buildings cannot be half steel and half wood in the same wall, and software routinely is. A Python service calling a Rust extension is completely normal and often the right answer.

■ 17.11.3 PLAIN – a worked example

1. **Scenario one: a machine-learning model for a research paper, six weeks.** Answer: Python. Not because it is fast, but because PyTorch, NumPy and pandas are there and every collaborator already reads it. The heavy numerical work happens in C++ and CUDA inside those libraries anyway.
2. **Scenario two: firmware for a battery-powered sensor with 64 KB of RAM.** Answer: C, or Rust if the team has the appetite. No garbage collector will fit, and you need exact control over every byte and every interrupt.
3. **Scenario three: an internal web tool for 200 staff, needed next month.** Answer: whatever the team already runs in production. Python with Django, Ruby with Rails, PHP with Laravel, or TypeScript with Node. The differences between these for this job are noise next to familiarity.
4. **Scenario four: a payments back-end that must run for fifteen years and be audited.** Answer: Java or C#. Long-term-support releases, deep tooling, large hiring pools, and a strong culture of backwards compatibility. Java 25 is supported into the 2030s.
5. **Scenario five: a network proxy handling 100,000 connections per second.** Answer: Go or Rust. Go if you want it working next month with cheap concurrency. Rust if the last 20 percent of performance and predictable latency without garbage collection pauses are worth the extra effort.
6. Notice that in five real scenarios, “which language do I like” appeared zero times.

■ 17.11.4 PLAIN – what is really happening inside

1. The hidden costs of a language choice, in the order they bite you.
2. Hiring. A language with 50,000 practitioners in your city and one with 500 are different business risks.
3. Libraries. Check whether the exact thing you need exists and is maintained, before you decide, not after.
4. Operations. Can your team debug it at 3am. Do profilers, tracing and crash reporting exist for it.
5. Build and deploy. Go gives you one file. Python gives you a dependency tree and a virtual environment. That difference shows up every single day.
6. Longevity. Ask whether the language has a paid maintainer, a standards body, or a large company depending on it. Languages with none of those have died before.
7. Interop. If you can call C from it, you can reach almost anything. Nearly every language on the list can.
8. Rewrites are the most expensive way to change your mind. The usual honest advice is to rewrite one component, measure, and only then decide.

■ 17.11.5 TECHNICAL – the engineer’s version

1. Rough performance ranking on CPU-bound work, taking optimized C as 1. These are order-of-magnitude figures from public benchmark suites and vary hugely by workload, so treat them as ranking only.

Language	Relative CPU time	Startup
C, C++, Rust	1x	under 1 ms
Go	1x to 2x	a few ms

Language	Relative CPU time	Startup
Java, C# (warm)	1x to 2x	100 ms to 1 s
JavaScript (V8)	2x to 10x	tens of ms
Python (CPython)	20x to 100x	tens of ms

2. Java and C# reach near-native speed only after the just-in-time compiler has warmed up, which is why start-up and steady-state must be quoted separately, and why ahead-of-time options such as GraalVM native image exist.
3. Python's figure collapses when the work is inside NumPy or PyTorch, because the loop then runs in C or CUDA and Python only issues the calls.
4. Decision inputs that are checkable rather than opinion: long-term-support policy and dates, the security advisory process, the package registry's provenance guarantees, the presence of an official formatter and linter, and whether the specification is a published standard or a single implementation.
5. Conway's law, published by Melvin Conway in 1968, applies: your system will mirror your communication structure, so a language choice that splits the team also splits the architecture.
6. Where experts genuinely disagree: whether a strong static type system pays for itself on small teams. Studies exist on both sides and none is conclusive, because the confounding variables are enormous.

■ 17.11.6 WORDS – remember these

1. Ecosystem — the libraries and tools around it — the package registry, frameworks, profilers and community that surround a language.
2. Long-term support — a version kept safe for years — a release with a published multi-year window for security patches.
3. Warm-up — the first minute is slower — the period before a JIT compiler has optimized the hot paths.
4. Interop — calling other languages — a foreign function interface, usually via the C application binary interface.
5. Rewrite risk — changing your mind costs years — the cost of porting a working system, historically underestimated by a large factor.

17.12 Reading other people's code and writing readable code

■ 17.12.1 PLAIN – in simple words

1. You will read far more code than you write, including your own from six months ago, which is somebody else's code.
2. So the real skill is not writing clever code. It is writing code the next person understands on the first read.
3. **Naming** is most of it. A name should say what the thing is, not what type it is and not how it works.
4. `d` tells you nothing. `days` tells you the unit. `days_since_signup` tells you the meaning.
5. A function should do **one thing**. If you need the word "and" to describe it, it is two functions.
6. Comments should explain **why**, not what. The code already says what.
7. A **style guide** is an agreement about layout so that nobody argues about it again.
8. A **formatter** rewrites your code into that layout automatically. A **linter** warns about patterns known to cause bugs.

9. **Code review** is another person reading your change before it ships.
10. **Technical debt** is a shortcut you took on purpose, that will cost more later, like a loan.

■ 17.12.2 PLAIN – a picture in your head

1. Think of code as directions written for a stranger arriving at night.
2. “Turn left at the thing” is a bad name. “Turn left at the petrol station” is a good one.
3. A step that says “drive to the roundabout and also buy milk” is two steps.
4. A note saying “this road is one-way northbound” is a good comment. A note saying “turn left here” next to the instruction “turn left here” is noise.
5. A style guide is everyone agreeing to write distances in kilometres.
6. Technical debt is the shortcut through the field. Faster tonight. Impassable after rain, and eventually you must build the road anyway, having also paid for the field.

Where this comparison breaks: directions are read once and code is read for years, by people who will change it. Code must be not only understandable but safely modifiable, which is a much higher bar.

■ 17.12.3 PLAIN – a worked example

1. Here is a real function, written badly.

```
def proc(d, f):
    r = []
    for x in d:
        if x[2] > f and x[4] == 1:
            r.append(x[0])
    return r
```

2. It works. Nobody can maintain it. What is `x[2]`, and what does 1 mean.
3. The same thing, rewritten.

```
def active_user_ids_above(users, min_age):
    """Return ids of active users older than min_age."""
    return [
        user.id
        for user in users
        if user.age > min_age and user.is_active
    ]
```

4. What changed: the function name says what comes out, the parameters are named, the fields are named instead of numbered, and the magic 1 became `is_active`.
5. Nothing got slower. The rewrite is the same number of machine operations.
6. Now comments. Here is a bad one and a good one.

```
i = i + 1          # add one to i
i = i + 1          # skip the header row
```

7. The first repeats the code and will rot the moment the line changes. The second records a fact about the data that the code cannot express.

■ 17.12.4 PLAIN – what is really happening inside

1. How to read code you did not write, in order.
2. Start at the entry point: `main`, the route handler, the command. Do not start at the top of a file.
3. Find the data structures before the functions. Once you know what the program holds, the functions usually explain themselves.

4. Run it. Put a breakpoint or a print in the middle and look at real values. Five minutes of running beats an hour of reading.
5. Use the tools: jump to definition, find all references, and `git log -p` on a file to see how it grew and why.
6. Read the tests. A good test suite is a specification with worked examples.
7. Now writing for review. Keep changes small. A 50-line change gets a real review. A 2,000-line change gets a shrug and an approval, and that is a measured effect, not a joke.
8. Separate a refactor from a behaviour change. Two commits, two reviews. Mixed together, nobody can see which lines actually changed the program.
9. Technical debt is only debt if you intend to repay it. Write it down: a comment, a ticket, or a TODO with a name and a date. Undocumented shortcuts are not debt, they are just a mess.

■ 17.12.5 TECHNICAL – the engineer’s version

1. Formatters and linters, with dates:

Tool	Language	Since
lint	C	1978, Stephen Johnson
gofmt	Go	2009, no options at all
ESLint	JavaScript	2013, Nicholas Zakas
Prettier	JS, CSS, more	2017, James Long
Black	Python	2018, Lukasz Langa
rustfmt, clippy	Rust	2016 onward
Ruff	Python	2022, written in Rust

2. gofmt made a design decision worth copying: one canonical format with no configuration. That ends the discussion permanently, which was the point. Black and rustfmt followed the same approach.
3. Ruff is roughly one to two orders of magnitude faster than the Python-based linters it replaces, because it is written in Rust and does a single pass. That speed changes behaviour: a linter fast enough to run on every keystroke gets used.
4. PEP 8, the Python style guide, was written by Guido van Rossum, Barry Warsaw and Nick Coghlan in 2001. Its 79-character line limit is a convention, not a standard, and teams commonly raise it to 88 or 100.
5. Code review as a formal practice comes from Michael Fagan’s inspection method at IBM, published in 1976. Modern lightweight pull-request review is a descendant with much of the ceremony removed.
6. Published review guidance, including Google’s engineering practices documents, converges on: review under 400 lines at a time, under 60 minutes at a stretch, and expect defect detection to fall sharply outside those bounds.
7. “Technical debt” was coined by Ward Cunningham in a 1992 OOPSLA experience report. His original meaning was deliberately shipping a not-yet-right design to learn from it, then refactoring. The common modern usage, meaning any sloppy code, is a drift from what he wrote, and he said so publicly.
8. Cyclomatic complexity, defined by Thomas McCabe in 1976, counts independent paths through a function. A value above roughly 10 is a common review threshold. It is a signal, not a rule.
9. Tools: `git blame`, `git log -S` to find when a string appeared, `radon cc` for Python complexity, `golanci-lint`, `clang-tidy`, and `pre-commit` to run all of them before a commit lands.

■ 17.12.6 WORDS – remember these

1. Refactor — change the shape, not the behaviour — a behaviour-preserving code transformation, ideally covered by existing tests.
2. Linter — a tool that warns about risky patterns — static analysis flagging likely defects and style violations without running the code.
3. Formatter — a tool that fixes layout — a deterministic rewriter producing a canonical source form.
4. Magic number — an unexplained literal — a bare constant with no name, carrying meaning nowhere recorded.
5. Code review — someone reads it before it ships — structured peer inspection of a change set prior to merge.
6. Technical debt — a shortcut with interest — deliberately deferred design work, tracked and intended to be repaid.
7. Cyclomatic complexity — how many ways through — the count of linearly independent paths through a function, McCabe 1976.

17.13 The standard library, packages and dependencies

■ 17.13.1 PLAIN – in simple words

1. The **standard library** is the set of code that ships with the language itself. Sorting, files, dates, networking, text.
2. Everything else you use is a **package**: code someone else wrote, published somewhere, that you download.
3. A **package manager** is the tool that downloads packages and keeps track of which versions you have.
4. Your package depends on other packages, which depend on others. Those are **transitive dependencies**, and there are usually far more of them than you expect.
5. A **version number** tells you how much a new release might break. That is what semantic versioning is for.
6. A **lock file** records the exact versions you actually used, so that the build on your machine and the build on the server are identical.
7. **Supply chain risk** is the plain fact that installing a package means running someone else's code, and trusting everyone they trusted.
8. In March 2016, an eleven-line package was deleted and thousands of builds around the world broke within minutes.

■ 17.13.2 PLAIN – a picture in your head

1. Think of building a car from parts catalogues.
2. The standard library is the parts that come in the box with the chassis.
3. A package is a part you order from a supplier. It arrives quickly and works.
4. But your supplier orders their bolts from someone else, who orders steel from someone else. You have contracts with one company and dependence on three hundred.
5. A lock file is writing down every part number and batch, so that next year's car is identical to this year's.
6. Supply chain risk is what happens when one bolt supplier decides to stop, or is bought by someone hostile, or ships a bad batch.

Where this comparison breaks: car parts are inspected, certified and physically present. Software dependencies are downloaded automatically by a script, often without a human ever looking, and a

compromised one runs with all the privileges of your build machine.

■ 17.13.3 PLAIN – a worked example

1. Semantic versioning, from the SemVer 2.0.0 specification written by Tom Preston-Werner. A version is three numbers.

```

2 . 7 . 3
|   |   |
|   |   +-- PATCH: bug fixes only, safe
|   +----- MINOR: new features, still compatible
+----- MAJOR: something broke, read the notes

```

2. So going from 2.7.3 to 2.7.4 should be safe. Going from 2.7.3 to 3.0.0 means the authors deliberately broke something.
3. In an npm package.json, "^2.7.3" means "any 2.x.y at or above 2.7.3", and "~2.7.3" means "any 2.7.x at or above 2.7.3".
4. This is a convention, backed by a written specification, and it depends entirely on authors honouring it. They sometimes do not.
5. Now the transitive problem. Install one popular web framework and count.

```
| npm ls --all | wc -l          # often over 1,000 lines
```

6. You chose one package. You installed several hundred, from several hundred different people, and you have read none of them.
7. The lock file is what makes this survivable. package-lock.json, Cargo.lock, poetry.lock, go.sum and Gemfile.lock all record the exact resolved version of every package, and usually a cryptographic hash of its contents.
8. Rule of thumb: commit the lock file for applications, so deployments are reproducible. For libraries, do not, so that whoever uses your library can resolve versions themselves.

■ 17.13.4 PLAIN – what is really happening inside

1. The left-pad incident, with dates.
2. On 22 March 2016, a developer named Azer Koculu had a trademark dispute over a package named kik. The npm registry transferred that name to the company.
3. In protest he unpublished all 273 of his packages from npm.
4. One of them was left-pad: eleven lines of code that pad a string on the left with spaces.
5. It was a dependency of a dependency of Babel, and through Babel of React, webpack and thousands of build systems.
6. Within minutes, builds failed worldwide with a 404 error for a package almost nobody had chosen to install.
7. npm restored the package from backup about two hours later, which was itself unprecedented, and then changed policy: a package cannot be unpublished after 24 hours if anything depends on it.
8. The technical lesson is not "do not use small packages". It is that your build had a single point of failure you did not know existed, controlled by a person you had never heard of.
9. The security version of the same lesson is worse. If someone takes over that account, they do not break your build. They add three lines that read your environment variables and send them somewhere.

10. This has happened repeatedly. In November 2018 the `event-stream` package was handed to a new maintainer who added code targeting a cryptocurrency wallet. In late 2021 the accounts behind `ua-parser-js`, `coa` and `rc` were hijacked and used to ship malware.
11. The most sophisticated known case is the `xz-utils` backdoor, CVE-2024-3094, found on 29 March 2024 by Andres Freund, a PostgreSQL developer who noticed that SSH logins were taking half a second longer than expected. It was the result of a patient multi-year effort to become a trusted maintainer.

■ 17.13.5 TECHNICAL - the engineer's version

1. Package managers by language:

Language	Manager	Registry
JavaScript	npm, yarn, pnpm	npmjs.com
Python	pip, uv, poetry	PyPI
Rust	Cargo	crates.io
Go	go modules	proxy.golang.org
Java	Maven, Gradle	Maven Central
C#	NuGet	nuget.org
Ruby	Bundler, gem	RubyGems
PHP	Composer	Packagist

2. Dates: npm launched January 2010, created by Isaac Schlueter. Maven 1.0 arrived in 2004. Cargo has shipped with Rust since 1.0 in 2015 and has never had a competitor, which is a deliberate design outcome. Go modules arrived in Go 1.11 in 2018 and became the default build mode in Go 1.16 in 2021, replacing GOPATH.
3. Go's approach differs on purpose: minimal version selection picks the lowest version satisfying all requirements, rather than the highest. Combined with a checksum database, this makes builds reproducible without a traditional resolver.
4. `go.sum` and the public checksum database, plus the module proxy, mean Go verifies that a module's contents have never changed since first seen. That is a stronger guarantee than most registries provide.
5. Diamond dependency conflicts occur when A needs C version 1 and B needs C version 2. npm resolves it by installing both, nested. Maven picks one by nearest-wins. Cargo allows two major versions to coexist. Python historically cannot, which is why virtual environments exist.
6. Supply chain defences that exist in 2026, with their status. Established: lock files with hashes, npm provenance attestations using Sigstore from 2023, PyPI trusted publishing, and two-factor authentication mandates on major registries. Active work: SLSA build-integrity levels and reproducible builds, which are real projects with partial adoption. Marketing claim: any product promising to "eliminate" supply chain risk.
7. Log4Shell, CVE-2021-44228, disclosed 9 December 2021 with a CVSS score of 10.0, was a vulnerability in a logging library that almost no Java team had consciously chosen. It is the clearest demonstration that transitive dependencies are your responsibility whether you know their names or not.
8. Standard library size is a design position. Go and Python ship large ones, Rust ships a deliberately small one and pushes the rest to crates.io, JavaScript ships almost nothing, which is exactly why `left-pad` existed.
9. Tools: npm audit, pip-audit, cargo audit, govulncheck, Dependabot, syft to generate a software bill of materials, and gype to scan one.

■ 17.13.6 WORDS – remember these

1. Standard library — what comes in the box — the modules specified and shipped with the language implementation.
2. Package manager — the tool that fetches code — resolves version constraints, downloads artefacts and records them.
3. Transitive dependency — your dependency's dependency — a package pulled in indirectly, usually the majority of the tree.
4. Semantic versioning — major, minor, patch — SemVer 2.0.0, where a major bump signals a deliberate breaking change.
5. Lock file — the exact versions you used — a pinned resolution with content hashes, giving reproducible installs.
6. Supply chain risk — trusting strangers by default — the attack surface of every upstream package, maintainer account and build step.
7. Software bill of materials — a parts list — a machine-readable inventory of every component in a build, in SPDX or CycloneDX format.

17.98 Common wrong ideas

1. Wrong: some languages are simply better than others. Right: every language is a set of trade-offs, and the sensible question is what the job demands in speed, safety, team skill, ecosystem and lifespan.
2. Wrong: Python is a slow language. Right: a language is a written specification and has no speed. CPython is a slow implementation, and the same Python calling NumPy runs the actual work in optimized C.
3. Wrong: static typing proves a program is correct. Right: it proves the shapes fit together. A program that computes the wrong average with perfectly matched types passes every type check ever written.
4. Wrong: garbage collection means you cannot leak memory. Right: it means you cannot leak unreachable memory. A cache or a listener list that keeps growing is fully reachable, and will exhaust the heap exactly as a C leak would.
5. Wrong: `x += 1` is a single operation, so two threads cannot corrupt it. Right: it is a load, an add and a store, and a thread can be interrupted between any two of them, which is precisely how lost updates happen.
6. Wrong: Big-O tells you which code is faster. Right: it tells you how cost grows with input size. For small inputs a worse complexity with a smaller constant routinely wins, which is why library sorts fall back to insertion sort on short runs.
7. Wrong: recursion is always more elegant than a loop. Right: it costs a stack frame per call, and without tail-call elimination, which most mainstream languages do not guarantee, deep recursion overflows the stack.
8. Wrong: a null check everywhere is the same as an option type. Right: a null check can be forgotten and the compiler will not notice, whereas an option type cannot be unwrapped without handling the empty case.
9. Wrong: more tests always mean better code. Right: coverage measures which lines ran, not which behaviours were checked. A property-based test that states a real rule can be worth a hundred example tests.
10. Wrong: a tiny dependency is harmless because it is only eleven lines. Right: you are trusting an account, a registry and a maintainer, as the left-pad breakage of 22 March 2016 and the xz-utils

backdoor of March 2024 both showed.

17.99 Chapter summary in 20 lines

1. A programming language is a notation with a grammar (syntax) and a meaning (semantics), defined by a document and implemented by separate tools.
2. Syntax errors are caught for you; semantic errors run happily and give the wrong answer, which is where the real difficulty lives.
3. Every language is built from the same parts: variables, literals, operators, expressions, statements, conditionals, loops, functions, scope and comments.
4. The same ten-line program in C, Python, JavaScript, Java, Go and Rust differs only in punctuation and ceremony, not in ideas.
5. A type is a set of values plus the legal operations on them; static means checked before running, dynamic means checked while running, and strong versus weak is the separate question of implicit conversion.
6. Tony Hoare added the null reference to ALGOL W in 1965 and called it his billion-dollar mistake in 2009; option types are the fix because they cannot be used without handling the absent case.
7. Manual memory management is fastest and admits leaks, double frees, use after free and dangling pointers; roughly 70 percent of reported vulnerabilities at Microsoft and Chromium were memory safety issues.
8. Reference counting is immediate and predictable but cannot reclaim cycles; tracing collection marks from roots and sweeps the rest, and generational collectors exploit the fact that most objects die young.
9. Rust's ownership and borrowing move the whole question to compile time, giving memory safety with no runtime cost and a much stricter compiler.
10. Paradigms are habits of organization: imperative, procedural, object-oriented, functional, declarative and event-driven, and real programs mix them.
11. Encapsulation, inheritance and polymorphism are the object-oriented trio; deep inheritance is the honest weak point, which is why composition is preferred and why Go and Rust omit class inheritance entirely.
12. Pure functions, immutability and higher-order functions with map, filter and reduce have spread into every mainstream language, whether or not the language calls itself functional.
13. Data structures are trades: arrays give $O(1)$ access and $O(n)$ insertion, hash tables give $O(1)$ average lookup by computing a bucket and resolving collisions, balanced trees give $O(\log n)$ with order preserved.
14. Big-O describes growth, not speed; $O(n \log n)$ sorting handles millions in a second while $O(2^n)$ handles about 26 items, and constants and cache behaviour decide the rest.
15. Real libraries ship Timsort (Python 2002, Java 7 in 2011), introsort (C++, Musser 1997), dual-pivot quicksort (Java primitives, 2009) and pattern-defeating quicksort (Go 1.19, 2022).
16. Dynamic programming turns exponential recomputation into linear work by solving each overlapping subproblem once and writing the answer down.
17. Concurrency is structure and parallelism is hardware; a race condition is unsynchronized access where one access writes, fixed by locks, atomics, message passing, or by not sharing at all.
18. Async and await are one thread and an event loop, which scales to tens of thousands of waiting connections and stalls completely on real computation; CPython's GIL became optional in 3.13 in 2024 and supported in 3.14 in 2025.
19. Errors are reported by return codes, exceptions, error values or Result types; assertions state invariants, and unit, integration, end-to-end and property-based tests each catch a different class of

mistake.

20. Choose a language from constraints, team and ecosystem rather than taste, write code the next reader understands, and treat every dependency as code you are choosing to run, as 22 March 2016 taught the whole industry.

Operating Systems

18.0 What this chapter gives you

1. You will be able to say exactly what an operating system is, and name the three jobs it exists to do.
2. You will be able to describe what a kernel is, what lives inside it, and why almost every kernel is written in C with a little assembly.
3. You will be able to narrate a computer's boot, from the power supply's ready signal to the login prompt, with real timings from a real machine.
4. You will be able to explain a process, a thread, and the difference, and read the output of `ps` without guessing.
5. You will be able to run a scheduling algorithm by hand on paper and produce the same answer the machine would.
6. You will be able to explain what happens mechanically when your program asks the kernel for something, and read a real `strace` line by line.
7. You will be able to describe file descriptors, permissions, mounting, pipes, signals and shared memory, and say which one to reach for.
8. You will be able to explain containers, virtual machines and kernel panics in terms of one idea: privilege.
9. You will be able to tell the history of UNIX, Linux, Windows, macOS and Android with correct names and years.
10. You will know what building your own operating system actually involves, and which learning resources to start from.

18.1 What an operating system is and why it must exist

■ 18.1.1 PLAIN – in simple words

1. An operating system is a program whose job is to run other programs.
2. It is the first big program that starts when you switch the machine on, and the last one running when you switch it off.
3. It has three jobs, and only three. Everything else it does is a detail of one of them.
4. Job one: **share the hardware**. One processor, one disk, one screen, many programs that all want them. Somebody must decide who gets what and when.
5. Job two: **keep programs apart**. Your music player must not be able to read or wreck your bank app's memory, whether by accident or on purpose.
6. Job three: **hide the hardware behind something simple**. Your program says "write these bytes to this file". It does not say "spin up the disk, move the head, wait".

7. Without an operating system you can still run one program. That program must then do every one of those three jobs itself, for itself.
8. The operating system is not the desktop, not the windows, not the icons. Those sit on top. The operating system is the part underneath that decides.

■ 18.1.2 PLAIN – a picture in your head

1. Think of a shared workshop with one lathe, one drill and one bench.
2. Twenty people want to build things. Left alone they would fight over the lathe, put each other's parts in the wrong bin and ruin each other's work.
3. So the workshop hires a manager.
4. The manager keeps a list of who wants the lathe and gives each person a turn of a few minutes. That is sharing the hardware.
5. The manager gives each person a locked drawer and refuses to let anyone open somebody else's drawer. That is keeping people apart.
6. The manager takes requests in plain words. You say "cut me a 40 mm disc". You do not set the lathe's speeds and feeds yourself. That is hiding the hardware behind something simple.
7. The manager also produces nothing. No customer ever pays for the manager's own work. The manager exists purely so that twenty people can share.

Where this comparison breaks: the workshop manager is a person with judgement, who can be argued with. The operating system is code, and it decides by fixed rules written years earlier by strangers. It also switches between jobs thousands of times a second, far faster than a human could hand over a lathe. And a real workshop manager cannot physically stop you opening a drawer. The operating system can, because the processor itself refuses the instruction.

■ 18.1.3 PLAIN – a worked example

1. You double-click a music player. Here is what the operating system does that you never see.
2. It finds the program file on the disk, reads its header, and works out how much memory it needs.
3. It sets aside memory for it, and arranges that the player can only see its own memory and nothing else.
4. It creates a **process**: a record saying "this program is running, here is its memory, here are its open files, here is its identity".
5. It starts the program's first instruction, and hands the processor to it.
6. A few milliseconds later it takes the processor away again and gives it to your browser, then to a background updater, then back to the player.
7. The player asks to open a sound file. It does not know which disk, which cable or which controller. It says a name. The operating system does the rest.
8. The player asks to send sound to the speakers. The operating system mixes it with the alert sound from your mail app so both are heard.
9. You close the player. The operating system reclaims the memory, closes the files, and removes the record.
10. Count the operating system's work in that story: sharing, separating and simplifying. There is nothing else.

■ 18.1.4 PLAIN – what is really happening inside

1. The trick that makes all of this possible is a feature of the processor itself, not of the software.
2. The processor can run in at least two modes. In one mode every instruction is allowed. In the other, dangerous instructions are refused.

3. The operating system's core runs in the all-allowed mode. Your programs run in the restricted mode.
4. In the restricted mode a program cannot change which memory it can see, cannot talk to devices directly, and cannot switch off interrupts.
5. So when a program needs any of those things it must ask. The asking mechanism is a special instruction that jumps into the operating system at a fixed, pre-arranged address.
6. The program does not choose where it lands. That is the whole point. It lands where the operating system decided, running the operating system's code.
7. A second hardware feature makes sharing possible: the timer interrupt. A chip pokes the processor at a fixed rate, say 250 times a second.
8. Each poke forces the processor to stop what it is doing and run operating system code. That is how a program that never gives up control is taken off the processor anyway.
9. Take away those two hardware features and no operating system can enforce anything. It becomes a library of helpful routines that programs may ignore.
10. That is exactly what life is like on a small embedded chip with no protection hardware, and it is why such systems trust their code completely.

■ 18.1.5 TECHNICAL - the engineer's version

1. Formally, an operating system is a resource manager and a virtual machine provider. Both descriptions appear in the standard textbooks: Silberschatz, Galvin and Gagne's *Operating System Concepts*, and Tanenbaum and Bos's *Modern Operating Systems*.
2. The resource manager view: it multiplexes CPU time, physical memory, storage bandwidth, network bandwidth and device access among competing requesters.
3. The virtual machine view: it presents each process with the illusion of a private processor, a private flat address space and a private set of files.
4. The enforcement primitives are hardware. On x86-64 these are the four privilege rings (only 0 and 3 are used in practice), the paging unit driven by CR3, and the `syscall` and `sysret` instruction pair.
5. On ARMv8-A the equivalents are exception levels EL0 (application), EL1 (kernel), EL2 (hypervisor) and EL3 (secure monitor), plus the `svc` instruction for supervisor calls.
6. Without an OS you are on **bare metal**. Real categories of bare-metal software: bootloaders, PC firmware, and the tightest embedded control loops.
7. A microcontroller such as an ATmega328P (used in the Arduino Uno) has 2 KB of SRAM and no memory management unit. There is no meaningful isolation to enforce, so the usual answer is a `main` with an infinite loop.
8. Between bare metal and a full OS sits the real-time operating system (RTOS). FreeRTOS is roughly 6,000 to 9,000 lines of C and gives you tasks, a scheduler, queues and semaphores, with no memory protection by default.

Layer	Isolation	Typical size
Bare metal loop	None	100 to 5,000 lines
RTOS (FreeRTOS)	Optional MPU	under 10,000 lines
Linux kernel 6.12	Full MMU + rings	over 30 million lines
Windows NT kernel	Full MMU + rings	tens of millions

9. The Linux source tree line count is the whole tree including every driver and every architecture. Any single running kernel compiles a small fraction of it. Quoting "Linux is 30 million lines" as

the size of a running kernel is wrong.

10. Tools that show you the boundary in action: `strace` on Linux, `dt russ` on macOS, Process Monitor on Windows, and `perf trace` on Linux.

■ 18.1.6 WORDS - remember these

1. Operating system — the program that runs other programs — the resource manager and abstraction layer between hardware and applications.
2. Bare metal — no operating system at all — code executing directly on hardware with no supervisor and no runtime services.
3. Kernel mode — the mode where everything is allowed — the privileged execution state, ring 0 on x86-64 or EL1 on ARMv8-A.
4. User mode — the mode where programs are restricted — the unprivileged execution state, ring 3 or EL0.
5. Timer interrupt — a regular poke from a chip — a periodic hardware interrupt used to force scheduler entry and enable preemption.
6. Multiplexing — sharing one thing among many users — time-division or space-division allocation of a single resource among several requesters.

18.2 The kernel: the part that is really the operating system

■ 18.2.1 PLAIN - in simple words

1. The **kernel** is the core of the operating system. It is the part that runs in the all-allowed mode of the processor.
2. It is a single program file on disk. On Linux it is often called `vm linux`. On Windows it is `ntoskrnl.exe`. On macOS it is the `kernel` file inside a kernel collection.
3. When people say “Linux” strictly, they mean only this file and its source code. Everything else you use is separate software.
4. Things that are inside the kernel: the scheduler, memory management, the file systems, the network stack, and most device drivers.
5. Things that are outside the kernel: the shell, the window system, the web browser, the compiler, the login screen, the package manager.
6. The dividing line is privilege, not importance. Your window system is very important and still runs as an ordinary program.
7. Code inside the kernel can do anything, so a mistake there can destroy the whole machine. Code outside can only wreck itself.
8. Designers therefore argue endlessly about how much should be inside.

■ 18.2.2 PLAIN - a picture in your head

1. Think of a hospital. The kernel is the operating theatre and the staff allowed inside it.
2. Inside the theatre you may do anything to the patient. Nothing stops you. That power is needed, and it is dangerous.
3. Everything else in the hospital — reception, the cafe, the pharmacy counter, the ward — happens outside, under normal rules.
4. A **monolithic kernel** is a hospital where the theatre is huge: pharmacy, imaging, pathology and the blood bank are all inside the sterile zone. Fast, because nobody walks anywhere. Risky, because one contaminated glove contaminates everything.

5. A **microkernel** is a hospital where the theatre is a tiny room containing only the surgeon. Pharmacy, imaging and pathology are separate departments you telephone. Safer. Slower, because of all the telephoning.
6. A **hybrid** is a hospital that says “tiny theatre” in the brochure, and in practice keeps imaging and pathology inside anyway.

Where this comparison breaks: hospital departments genuinely cannot infect each other from down the corridor, whereas a microkernel’s separated services still share the same physical memory chips and the same processor. The isolation is enforced by the memory management unit, not by distance. And a phone call between hospital departments costs seconds; a message between microkernel services costs about a microsecond. The relative costs are completely different.

■ 18.2.3 PLAIN – a worked example

1. Ask: where does the code that understands the ext4 filesystem live?
2. In Linux, inside the kernel. It is a module, `ext4.ko`, loaded into kernel memory and running with full privilege.
3. In MINIX 3, a filesystem server is an ordinary user-mode process. If it crashes, MINIX restarts it and the machine keeps running.
4. In Windows, NTFS is `ntfs.sys`, a kernel-mode driver. Full privilege.
5. Now ask: where does the code that draws your windows live?
6. On Linux, outside the kernel entirely, in the X server or a Wayland compositor. The kernel only provides the low-level graphics device interface.
7. On Windows since NT 4.0 in 1996, a large part of the window manager and the graphics engine was moved **into** kernel mode for speed. That decision was controversial and caused years of graphics-driver blue screens.
8. Same job, three different sides of the line, for reasons of speed and history rather than principle.

■ 18.2.4 PLAIN – what is really happening inside

1. A kernel is not a process. Nothing schedules it as a job. It is a body of code that gets entered when something happens.
2. There are exactly three ways in. Remember these three and kernel behaviour stops being mysterious.
3. Way one: a **system call**. A program deliberately executes the special instruction that jumps into the kernel.
4. Way two: an **interrupt**. A device raises a line, the processor stops the current instruction stream and runs a kernel handler.
5. Way three: an **exception** or fault. The program did something the processor could not complete: touched an unmapped page, divided by zero, executed an illegal instruction.
6. In all three cases the processor switches mode, switches to a kernel stack, and jumps to an address the kernel published in advance.
7. The kernel does the work, then returns, and the processor drops back to the restricted mode at the exact instruction that follows.
8. Between those events the kernel is doing nothing at all. An idle Linux machine really is idle. The kernel is a set of reactions, not a running loop.
9. There are exceptions to that: kernels also run their own threads for background work. On Linux you can see them in `ps` with names in square brackets, like `[kworker/0:1]`.

■ 18.2.5 TECHNICAL – the engineer’s version

1. **Monolithic:** all OS services execute in a single kernel address space in supervisor mode. Communication is a plain function call. Examples: Linux, FreeBSD, all classic UNIX.
2. **Microkernel:** the kernel provides only address spaces, threads and inter-process communication. Drivers, filesystems and network stacks run as user-mode servers. Examples: MINIX 3, QNX Neutrino, seL4, GNU Hurd.
3. **Hybrid:** a microkernel-derived structure with performance-critical services linked into kernel space. Examples: Windows NT, XNU. Some engineers reject “hybrid” as a marketing word for “monolithic with a Mach heritage”. Say that both readings are defensible.
4. **Exokernel:** the kernel only multiplexes and protects hardware, exposing it almost raw; libraries in user space implement the abstractions. From MIT, Dawson Engler and Frans Kaashoek, 1995. Nearly all influence, almost no deployment. The idea resurfaced in unikernels and in Linux `io_uring`.
5. The **Tanenbaum-Torvalds debate** began on 29 January 1992 in the `comp.os.minix` newsgroup, in a post by Andrew S. Tanenbaum, author of MINIX and of the standard textbook, titled “LINUX is obsolete”.
6. Tanenbaum’s two claims: monolithic kernels are the wrong design, and writing one in 1991 was “a giant step back into the 1970s”; and Linux was too tied to the Intel 386 to survive changes in hardware.
7. Torvalds’s replies: microkernels are theoretically nicer but MINIX had real design faults, targeting the 386 was a deliberate choice for a learning project, and application-level portability is what users actually feel.
8. How it turned out, honestly: Tanenbaum was right about portability mattering and Linux was later ported to more than 20 architectures. He was wrong about the market outcome. Torvalds was right that a working monolithic kernel beat an elegant unfinished one. Neither side won the design argument outright, and seL4’s formal proof of correctness in 2009 showed the microkernel case is still very much alive.

Kernel	Structure	Primary languages
Linux	Monolithic, modular	C, asm, Rust
Windows NT	Hybrid	C, C++, asm
XNU (macOS)	Hybrid, Mach + BSD	C, C++, asm
MINIX 3	Microkernel	C
seL4	Microkernel, verified	C, asm

9. Why C: it compiles to predictable machine code with no hidden runtime, no garbage collector, no exceptions and no implicit allocation. A kernel cannot depend on services that only exist once the kernel is running.
10. Why assembly is still needed: the very first instructions after reset, context switch, interrupt entry and exit, setting control registers, and atomic primitives. In Linux 6.x the `arch/` directory holds this, a few tens of thousands of assembly lines against millions of lines of C.
11. Why C++ appears in places: XNU’s IOKit driver framework uses a restricted subset of Embedded C++ with no exceptions, no templates and no multiple inheritance. Windows drivers use C++ widely.
12. **Rust in Linux:** Miguel Ojeda’s Rust for Linux work was merged for Linux 6.1, released 11 December 2022. The first production Rust drivers landed in Linux 6.8 in March 2024, including the

Android Binder driver and two Ethernet PHY drivers. As of December 2025 Rust is an official kernel language alongside C and assembly, not an experiment.

13. Why Rust: about two-thirds of serious kernel security bugs historically come from memory-safety errors — use after free, buffer overflow, data races. Rust's borrow checker rejects most of those at compile time with no runtime cost. This is established, not marketing: Microsoft and Google have both published the roughly 70 percent figure for their own codebases.
14. Windows also ships Rust in the kernel: Microsoft stated in 2023 that parts of the Windows kernel, including a rewritten win32kbase font parser and DWM core components, are now Rust.

■ 18.2.6 WORDS – remember these

1. Kernel — the core that has full power — the privileged portion of the OS that executes in supervisor mode.
2. Monolithic kernel — one big privileged block — all services share a single kernel address space and call each other directly.
3. Microkernel — a tiny privileged core — only address spaces, threads and IPC are privileged; other services run as user-mode servers.
4. Kernel module — a piece of kernel you can add later — dynamically loadable object linked into the running kernel's address space.
5. Interrupt — a device raising its hand — an asynchronous hardware signal that diverts the processor into a registered handler.
6. Exception — the processor could not continue — a synchronous fault such as a page fault, divide by zero or invalid opcode.

18.3 The boot sequence, step by step

■ 18.3.1 PLAIN – in simple words

1. Booting is the process of getting from “no power” to “a working machine”, using nothing but what is already stored in the hardware.
2. The word comes from “pulling yourself up by your own bootstraps”, which is the impossible thing this process appears to do.
3. It is a chain. Each stage is small and stupid, and its only real job is to find and start the next, slightly cleverer stage.
4. Stage 0: the power supply tells the rest of the machine that the voltages are stable and it is safe to start.
5. Stage 1: the processor starts executing at one fixed address that is burned into the chip's design. It always starts there. It cannot start anywhere else.
6. Stage 2: at that address is **firmware**, a small program in a chip on the board. It tests the machine and finds something to boot from.
7. Stage 3: the firmware loads a **bootloader** from disk. The bootloader knows how to find the operating system.
8. Stage 4: the bootloader loads the kernel into memory and jumps into it.
9. Stage 5: the kernel sets itself up, finds the real disk, and starts the very first ordinary program.
10. Stage 6: that first program starts everything else, and eventually you get a login prompt.

■ 18.3.2 PLAIN – a picture in your head

1. Think of arriving at a large office building at night, with only a note in your pocket saying “go to the front desk”.
2. At the front desk is a laminated card. It says: check the lights work, check the lifts work, then take the lift to floor 3.
3. On floor 3 is another card: it says which of several offices you want, and what the key code is.
4. In that office is a thick manual. Reading it takes a while. It tells you how to run the whole building.
5. Once you have read the manual, you telephone the staff and they arrive one by one and start their own jobs.
6. Finally the reception desk opens and members of the public can come in. That is your login prompt.
7. Notice that no stage had to know everything. Each stage only had to know enough to find the next.

Where this comparison breaks: you are one person walking through a building, whereas the real sequence involves several processors. On a modern x86 machine the main processor is not even the first thing that runs; a separate management engine inside the chipset starts first and holds the main processor in reset. And the “cards” are not just instructions, they are complete programs with drivers, network stacks and graphics, sometimes larger than early operating systems.

■ 18.3.3 PLAIN – a worked example

1. Here is a real boot, measured on the Linux machine this chapter was written on. The times are seconds since the kernel took control, printed by `dmesg`.

```
0.000000 Linux version 6.18.5, gcc 15.2.0
0.000064 tsc: Detected 2100.000 MHz processor
0.165101 Booting paravirtualized kernel on KVM
0.173953 random: crng init done
0.475547 smpboot: CPU0: Intel(R) Xeon(R) @ 2.10GHz
0.494363 smpboot: Total of 2 processors activated
0.633029 Memory: 8203096K/8388216K available
0.636116 devtmpfs: initialized
0.927746 Unpacking initramfs...
1.039733 virtio_blk virtio1: [vda] 536870912 blocks
1.162061 Freeing unused kernel image memory: 2892K
1.163192 Write protecting kernel read-only data: 26624k
1.174639 Run /process_api asinit process
2.551159 EXT4-fs (vda): mounted filesystem r/w
```

2. Read that from the top. For the first 0.165 seconds the kernel is doing nothing but working out what machine it is on.
3. At 0.475 the first processor is identified, and by 0.494 the second processor has been woken up and put to work. Before that moment the machine had one working core no matter how many it owns.
4. At 0.633 the memory manager reports what it has: 8,203,096 KB usable out of 8,388,216 KB total. The missing 185 MB is the kernel itself plus reserved regions.
5. At 0.927 the kernel unpacks the **initial ramdisk**, a small filesystem held in memory that contains the drivers needed to reach the real disk.
6. At 1.039 the block device driver finds the disk. Only now does a disk exist as far as the kernel is concerned.

7. At 1.162 the kernel frees its own setup code, which will never run again. 2,892 KB returned to the pool.
8. At 1.174 the kernel starts the first user-mode program. On this machine that is `/process_api`. On a normal Linux desktop the line reads `Run /sbin/init as init process`.
9. At 2.551 the real root filesystem is mounted. Everything after this point is ordinary software.
10. The whole kernel phase took 2.55 seconds, and 1.37 of those seconds were spent after the first user program had already started.

■ 18.3.4 PLAIN – what is really happening inside

1. **Power good.** The power supply raises a signal when its outputs are within tolerance. Until then the whole board is held in reset. On a desktop this takes roughly 100 to 500 milliseconds after you press the button.
2. **Reset vector.** The processor comes out of reset with its registers at fixed values and starts fetching from one address chosen by the chip designers. Nothing is loaded from disk yet, because nothing can read a disk.
3. That address is wired so that it lands in the firmware chip, not in RAM. RAM is not even usable yet, because the memory controller has not been set up.
4. **Firmware.** The firmware runs. It configures the memory controller, tests the machine, and builds tables describing what hardware exists.
5. The self test at this stage is called POST, power-on self test. If it fails before the screen works, the machine reports the failure by beeping in a pattern or flashing a light, because that is all it has.
6. **Boot device selection.** The firmware works down a list of places to look: internal disk, USB, network. You can edit this list in the firmware settings.
7. **Bootloader.** The firmware loads a small program from the chosen device and jumps to it. This program's whole job is to find the kernel.
8. Why a separate bootloader exists at all: the firmware does not know what a Linux kernel is, and the kernel is too big and too clever to be started directly by dumb firmware.
9. **Loading the kernel.** The kernel file on disk is usually compressed. The bootloader loads it, and a small decompressor stub at its front expands the real kernel into memory and jumps to it.
10. **Initial ramdisk.** The bootloader also loads a second file: a compressed archive of a tiny root filesystem. The kernel unpacks it into memory.
11. Why: the real disk might need a driver, and that driver might live on the real disk. The ramdisk breaks the circle by carrying the driver with it.
12. **Kernel initialization.** The kernel sets up its page tables, starts the other processors, initializes the scheduler and memory manager, and probes for devices.
13. **Mounting the root filesystem.** With a working disk driver, the kernel mounts the real root and switches to it, then throws away the ramdisk.
14. **The first user process.** The kernel executes one program in user mode and gives it process ID 1. If that program ever exits, the kernel panics, because there is nothing left to run.
15. **Services.** Process 1 reads its configuration and starts everything else: logging, networking, the sound daemon, the display manager.
16. **Login.** The display manager or the terminal `getty` draws a prompt. The boot is over.

■ 18.3.5 TECHNICAL – the engineer’s version

1. On x86-64 the reset vector is physical address `0xFFFFFFF0`, sixteen bytes below the top of the 4 GiB space, executed in 16-bit real mode with CS base set so the first fetch lands in the firmware flash. This is fixed by Intel and AMD, not by software.
2. The processor comes out of reset in real mode even on a 2026 machine, and the firmware must walk it through protected mode to long mode. That is 1978 compatibility still costing time in every boot.
3. On ARMv8-A the reset address is implementation defined and supplied by the SoC; the boot core starts at EL3 and steps down to EL2 and EL1.
4. **BIOS** is the legacy firmware interface, from the IBM PC of 1981. It reads the first 512-byte sector, the master boot record, checks for the signature `0x55AA` at offset 510, loads it at `0x7C00` and jumps there. 446 bytes of code is all you get.
5. **UEFI** replaced it. Intel began the work in 1998 for Itanium as the Intel Boot Initiative, later EFI; Intel stopped at version 1.10 and handed it to the Unified EFI Forum, which published UEFI 2.0 in 2006. The current specification is UEFI 2.11, published in December 2024.
6. UEFI boots differently: it reads a GPT-partitioned disk, mounts the EFI System Partition, which the specification requires to be FAT12, FAT16 or FAT32, and executes a PE-format `.efi` executable from it.
7. Standard ESP paths: `\EFI\B00T\B00TX64.EFI` is the fallback the specification mandates; `\EFI\Microsoft\Boot\bootmgfw.efi` is Windows Boot Manager; `\EFI\ubuntu\grubx64.efi` is a distribution’s GRUB. The first is a standard. The others are conventions.
8. **Secure Boot** is a UEFI feature: firmware verifies the signature on the loaded image against keys in the `db` variable. Most distributions use a small Microsoft-signed shim that then verifies the distribution’s own key.
9. **GRUB 2** is the common Linux bootloader. It reads the filesystem, presents a menu, and uses the Linux boot protocol: it loads `vmlinuz` and `initrd`, fills in a `boot_params` structure, and jumps to the kernel entry point.
10. The kernel command line is passed as a string. On the machine used above it contained `rdinit=/process_api`, which is what caused PID 1 to be that program rather than `/sbin/init`.
11. The initial ramdisk on modern Linux is an **initramfs**: a `cpio` archive, usually compressed with `gzip`, `zstd` or `lz4`, unpacked into a `tmpfs`. The older `initrd` was a block-device image. Both names are still used loosely.
12. **PID 1** in practice: `systemd` on most Linux distributions since Fedora 15 in May 2011, Arch in October 2012, and Debian and Ubuntu from 2015; `launchd` on macOS since Mac OS X 10.4 Tiger in 2005, written by Dave Zarzycki; `wininit.exe` and `smss.exe` on Windows; `init` from BusyBox or OpenRC on smaller Linux systems.
13. Typical timings on a real x86-64 laptop, measured with `systemd-analyze`. These are approximate and vary widely by machine.

Phase	Typical time	Tool to measure
Firmware (UEFI)	1 to 12 s	<code>systemd-analyze</code>
Bootloader	0.1 to 1 s	<code>systemd-analyze</code>
Kernel to PID 1	1 to 4 s	<code>dmesg</code> timestamps
Userspace to login	1 to 15 s	<code>systemd-analyze blame</code>

14. Commands that observe this: `dmesg` for kernel timestamps, `systemd-analyze` for the phase split, `systemd-analyze blame` for the slowest services, `systemd-analyze critical-chain` for the de-

pendency path, `bootctl` for UEFI state, and `efibootmgr -v` for the boot entry list.

15. The honest version: the sequence described here starts at the main processor's reset vector, and on modern x86 that is not the true beginning. Intel's Management Engine and AMD's Platform Security Processor run first, verify the firmware, and release the main cores. You cannot observe or disable that from the operating system.

```

power good
  |
  v
CPU reset vector 0xFFFFFFFF0 (real mode)
  |
  v
firmware: POST, memory init, ACPI tables
  |
  v
boot device list -> ESP -> B00TX64.EFI / GRUB
  |
  v
load vmlinuz + initramfs, fill boot_params
  |
  v
kernel: decompress, page tables, SMP, drivers
  |
  v
unpack initramfs -> find real root -> switch_root
  |
  v
exec PID 1 (systemd / launchd / init)
  |
  v
services -> display manager -> login prompt

```

■ 18.3.6 WORDS – remember these

1. Bootstrapping — starting from nothing by stages — a chain loader sequence in which each stage loads and transfers control to the next.
2. Reset vector — the one address a CPU starts at — the architecturally fixed fetch address after reset, `0xFFFFFFFF0` on x86-64.
3. Firmware — the program in a chip on the board — non-volatile platform code implementing BIOS or UEFI services.
4. POST — the switch-on self test — power-on self test, reporting faults by beep codes or diagnostic LEDs before video exists.
5. Bootloader — the program that finds the kernel — a chain-loading stage such as GRUB 2, Windows Boot Manager or `boot.efi`.
6. Initramfs — a tiny filesystem carried in memory — a compressed cpio archive unpacked into `tmpfs`, holding the drivers needed to mount the real root.
7. PID 1 — the first ordinary program — the `init` process, ancestor of all user processes; its exit causes a kernel panic.

18.4 Processes

■ 18.4.1 PLAIN – in simple words

1. A **program** is a file on disk. It does nothing. It is just bytes.
2. A **process** is that program while it is running: the code, plus its memory, plus its current position, plus everything the system is holding for it.
3. The same program can be running as many processes at once. Ten terminal windows are ten processes from one program file.
4. Each process gets a number, the **process ID**, usually written PID. It is how everything else refers to it.
5. Every process except the very first has a **parent**: the process that created it. That gives you a family tree with PID 1 at the root.
6. A process is not always running. Most of the time most processes are asleep, waiting for a key press, a disk read or a network packet.
7. When a process finishes, it does not vanish immediately. It leaves behind a small note saying how it ended, and waits for its parent to read it.
8. If the parent never reads that note, the dead process stays in the table as a **zombie**: dead, but still listed.
9. If a parent dies before its child, the child is adopted by PID 1. Such a child is called an **orphan**.

■ 18.4.2 PLAIN – a picture in your head

1. Think of a cookery book on a shelf. That is the program.
2. Now think of you, in a kitchen, halfway through a recipe from it. That is the process.
3. The process is not the recipe. It is the recipe **plus** the ingredients on the bench, the pan on the heat, your finger on the current line, and the oven timer you set.
4. Your sister can cook the same recipe in a different kitchen at the same time. Same program, two processes, completely separate benches.
5. The doorbell rings. You mark your place, wipe your hands and answer it. When you come back you read your mark and continue. That is a context switch.
6. If you leave the kitchen for good, someone must still come and clear up and note whether the dish worked. Until they do, the mess stays on the bench. That is a zombie.

Where this comparison breaks: you have one pair of hands, so you genuinely stop cooking when you answer the door. A modern processor has several cores and can truly cook several dishes at once. And your memory of where you were is in your head; a process's memory of where it was is written into a data structure that another program could, with enough privilege, read or alter.

■ 18.4.3 PLAIN – a worked example

1. Here is a real program that creates a child and replaces the child with a different program. This is how every UNIX shell starts every command.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void) {
    printf("before fork: pid=%d ppid=%d\n",
           getpid(), getppid());
    pid_t p = fork();
    if (p == 0) {
        printf("child : pid=%d ppid=%d fork gave %d\n",
               getpid(), getppid(), p);
        execl("/bin/echo", "echo", "child is now echo",
              NULL);
    }
}
```

```

        _exit(127);
    }
    printf("parent: pid=%d fork gave %d\n", getpid(), p);
    int st; waitpid(p, &st, 0);
    printf("parent: child %d exited with %d\n",
          p, WEXITSTATUS(st));
    return 0;
}

```

2. Compiled with gcc and run, it printed this, exactly:

```

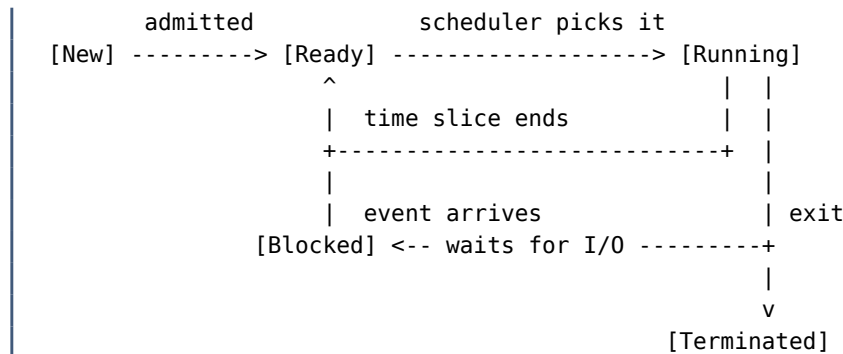
before fork: pid=5344 ppid=4984
child : pid=5346 ppid=5344 fork gave 0
child is now echo
parent: pid=5344 fork gave 5346
parent: child 5346 exited with 0

```

3. Read it carefully. fork was called once and returned twice, in two different processes.
4. In the child it returned 0. In the parent it returned 5346, the child's PID. That is the only way each side can tell which one it is.
5. The child's own PID is 5346 and its parent PID is 5344, which is the parent. The family link is real and visible.
6. Then `execl` replaced the child completely. Same PID 5346, entirely different program. The line "child is now echo" was printed by `/bin/echo`.
7. The parent waited, and collected the exit status 0.
8. One honest detail. When the same program's output was sent into a pipe instead of a screen, the line `child : pid=...` never appeared at all. The C library buffers output when it is not a terminal, the text was still sitting in the buffer, and `execl` threw the whole buffer away when it replaced the process image. This is a real bug people hit; the fix is `fflush` before `exec`.

■ 18.4.4 PLAIN – what is really happening inside

1. The kernel keeps one record per process. Textbooks call it the **process control block**. Linux calls it `struct task_struct`.
2. That record holds, at minimum: the process ID, the parent's ID, the state, the saved register values, a pointer to the memory map, the table of open files, the owning user, the scheduling priority and the accounting counters.
3. On Linux 6.x `struct task_struct` is a few kilobytes and has well over a hundred fields. All processes live in a linked structure the kernel walks.
4. A process moves between a small number of **states**. There are five that matter.
5. **New**: the record exists but the process has not started running.
6. **Ready**: it could run right now, it just does not have a processor.
7. **Running**: it is executing on a processor this instant.
8. **Blocked** or waiting: it asked for something that has not arrived, such as a disk read, and cannot proceed.
9. **Terminated**: it has finished but the record has not yet been removed.
10. Only three transitions are interesting. Ready to running is the scheduler choosing it. Running to ready is the scheduler taking the processor away. Running to blocked is the process itself asking for something.
11. Note the asymmetry: blocked goes to ready, never straight to running. When your disk read finishes you become eligible; you do not jump the queue.



12. Creating a process on UNIX is two steps, deliberately. `fork` makes a copy of the calling process. `exec` throws away the copy's program and loads a new one in its place.
13. That looks wasteful and is not, because `fork` does not really copy the memory. It marks every page as shared and read-only, and only makes a real copy of a page when one side writes to it. This is **copy on write**.
14. Windows does it in one step instead. `CreateProcess` takes the program name and a pile of options and makes a new process running that program. There is no `fork`.
15. The two-step design is why UNIX shells are simple: between `fork` and `exec` the child can quietly rearrange its own file descriptors, and the new program inherits the arrangement without knowing anything about it.

■ 18.4.5 TECHNICAL - the engineer's version

1. Linux does not really have separate `fork`, `vfork` and thread creation. All three are `clone()` with different flag sets. `fork` is `clone` with `SIGCHLD` and nothing shared.
2. This trace, captured with `strace` on the shell command `sh -c 'echo hello world | tr a-z A-Z'`, shows a real `fork`:

```
clone(child_stack=NULL,
      flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID
        |SIGCHLD,
      child_tidptr=0x7f26e1971a10) = 18232
```

3. Process states in `ps` on Linux, from the `STAT` column: R runnable or running, S interruptible sleep, D uninterruptible sleep (usually disk), T stopped, Z zombie, I idle kernel thread. Suffixes: s session leader, l multi-threaded, + foreground, < high priority, N low.
4. D state deserves special mention. A process in uninterruptible sleep cannot be killed, not even with signal 9, because it is inside a kernel path that cannot safely unwind. Persistent D state means broken storage or a hung network filesystem.
5. A real zombie, produced deliberately by forking a child that exits while the parent does not call `wait`:

```
PID  PPID  STAT  COMMAND
20066 20065  Z     zombie
```

6. A real orphan, produced by a parent that exits before its child. The child printed its parent ID twice, one second apart:

```
parent 23414: exiting immediately
child 23415: ppid is 23414
child 23415: ppid is now 1
```

7. Reparenting to PID 1 is the kernel's rule. On systemd systems a process can instead register itself as a subreaper with `prctl(PR_SET_CHILD_SUBREAPER)`, and orphans go to it rather than to PID 1.
8. PID limits: the default maximum on 64-bit Linux is 4,194,304, set in `/proc/sys/kernel/pid_max`. The historical default of 32,768 came from a 16-bit signed field. PIDs wrap around and are reused, which is a real source of race conditions in scripts that kill by PID.
9. Per-process limits are visible in `/proc/PID/limits`. On the machine used here:

Limit	Soft	Hard
Max stack size	8388608	unlimited
Max processes	32056	32056
Max open files	20000	20000
Max core file size	0	unlimited

10. `/proc/PID/stat` is the raw record, 52 space-separated fields. The first seven are pid, comm, state, ppid, pgrp, session, tty. Field 14 and 15 are user and system jiffies consumed. `ps` and `top` read exactly this file.
11. Windows: `CreateProcessW` takes 10 parameters including `STARTUPINFO` and `PROCESS_INFORMATION`. A Windows process starts with one thread and a PEB (process environment block). Handles are inherited only if marked inheritable and `bInheritHandles` is true.
12. Observation tools by platform: `ps aux`, `ps -ef`, `pstree`, `top`, `htop` and `/proc` on Linux; `ps`, `top` and Activity Monitor on macOS; `dtruss` and `sample` on macOS; Task Manager, Process Explorer, Process Monitor and `Get-Process` on Windows.
13. Useful `ps` incantations. `ps -eo pid,ppid,stat,pri,ni,etime,comm` prints the fields this section talks about. `ps -eLf` shows one line per thread. `ps -eo pid,rss,vsz,comm --sort=-rss` finds memory hogs.

■ 18.4.6 WORDS – remember these

1. Program — a file of instructions — a passive executable image on storage.
2. Process — a program while running — an execution instance with its own address space, descriptor table, credentials and scheduling state.
3. Process control block — the kernel's record of a process — `task_struct` on Linux, `EPROCESS` on Windows.
4. PID — a process's number — a reusable integer identifier, capped by `/proc/sys/kernel/pid_max`.
5. fork — make a copy of yourself — `clone()` creating a new address space via copy-on-write page sharing.
6. exec — become a different program — replace the current process image while keeping the PID and inherited descriptors.
7. Zombie — dead but still listed — a terminated process whose exit status has not yet been reaped by `wait`.
8. Orphan — a child whose parent died — a process reparented to PID 1 or to the nearest subreaper.
9. Copy on write — pretend to copy, copy only if changed — shared read-only page mappings duplicated lazily on the first write fault.

18.5 Threads

■ 18.5.1 PLAIN – in simple words

1. A **thread** is one flow of execution: one finger moving down the list of instructions.
2. A process has at least one thread. It may have many.
3. All the threads in one process share the same memory. If one writes to a variable, the others see the change instantly.
4. Each thread has its own **stack**, because each thread is in a different place in the program and has its own local variables and its own chain of calls.
5. That is the entire difference. Shared memory, separate position and stack.
6. Threads are used when several parts of one job must happen at once: one thread drawing the screen, another loading a file, another talking to the network.
7. Because they share memory, threads are fast to create and cheap to talk between.
8. Because they share memory, one bad thread can corrupt the data of all the others, and there is no protection at all.
9. That trade is the whole story of threads: speed and sharing, bought with danger.

■ 18.5.2 PLAIN – a picture in your head

1. A process is a flat with a locked front door. A thread is a person living in it.
2. Two flats cannot see into each other. That is process isolation, enforced by the lock.
3. Two flatmates share the kitchen, the fridge and every cupboard. Neither needs permission to open anything. That is thread sharing.
4. Each flatmate still has their own bedroom with their own things. That is the per-thread stack.
5. If one flatmate takes the milk while the other is pouring it, you get a mess. Nobody broke a rule. They just both used one thing at once. That is a race condition.
6. Moving into a spare bedroom is quick. Getting a whole new flat takes longer and costs more. That is why threads are cheaper than processes.

Where this comparison breaks: flatmates can talk and agree who uses the kitchen. Threads cannot agree about anything unless you write the agreement yourself, in the form of locks. And a flat has one kitchen; a multi-core processor gives each thread its own set of caches, so two threads can genuinely believe different things about the same variable until the caches are made to agree.

■ 18.5.3 PLAIN – a worked example

1. Measured on the machine used for this chapter, a 2.10 GHz Intel Xeon with two cores, Linux 6.18, glibc on Ubuntu 24.04.
2. One thousand threads created, then all joined. One thousand processes forked, then all reaped.

pthread_create only	:	32.8 us
fork only (small process)	:	182.5 us
fork only (256 MB dirty)	:	2620.1 us

3. Creating a thread cost 32.8 microseconds. Creating a process cost 182.5 microseconds. The thread is about 5.6 times cheaper.
4. Now the third line. The same fork, after the parent had allocated and written to 256 MB of memory, cost 2,620 microseconds. Fourteen times more.
5. Why: copy-on-write does not copy the data, but it must still copy the page tables that describe it. 256 MB is 65,536 pages of 4 KB, and every entry must be duplicated and marked read-only.
6. The honest version: “a thread is far cheaper than a process” is only true sometimes. Fork’s cost depends on how much memory the parent has mapped. Thread creation does not depend on that at all.

7. A separate measurement, thread create and join one at a time, gave 80.2 microseconds against 134.3 for fork and wait. The gap narrows because the join and the wait dominate.
8. Take the lesson, not the numbers: threads are cheaper, but on Linux the gap is a factor of a few, not a factor of a thousand. On Windows the gap is larger, because Windows process creation is genuinely heavier.

■ 18.5.4 PLAIN – what is really happening inside

1. Here is exactly what is shared between two threads of one process, and what is not.
2. Shared: the code, the global variables, the heap, the open file table, the current directory, the user identity, the signal handlers.
3. Not shared: the stack, the register values including the program counter, the thread ID, the signal mask, and any thread-local storage.
4. When the kernel switches from thread A to thread B in the same process, it saves A's registers and loads B's, and that is nearly all.
5. When it switches between threads of different processes, it must also switch the memory map, which means loading a new page table root and, on older processors, flushing the address translation cache.
6. That flush is why cross-process switches cost more than same-process ones. Modern processors avoid most of it with tagged translation caches: PCID on x86-64 since Westmere in 2010, ASID on ARM.
7. There are two places threads can be implemented.
8. **Kernel-level threads:** the kernel knows about each one and schedules them individually. If one blocks on disk, the others keep running. This is what Linux, Windows and macOS do.
9. **User-level threads:** a library inside your process switches between them, and the kernel sees only one thread. Switching is very fast because no mode change happens. But if one blocks in the kernel, all of them stop.
10. Modern languages have revived the user-level idea under new names: goroutines in Go, virtual threads in Java 21, async tasks in Rust and Python. They pair it with a runtime that never blocks the underlying kernel thread, which removes the old flaw.
11. A **thread pool** exists because creating a thread costs tens of microseconds and most tasks are shorter than that. You create a fixed set of threads once and feed them work from a queue for the life of the program.

■ 18.5.5 TECHNICAL – the engineer's version

1. The POSIX threads API is IEEE Std 1003.1c-1995, universally called pthreads. The core calls are `pthread_create`, `pthread_join`, `pthread_detach`, `pthread_mutex_lock` and `pthread_cond_wait`.
2. On Linux a pthread is created by `clone()` with `CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD`. The kernel calls the result a task; there is no separate thread object.
3. Because of `CLONE_THREAD`, all threads of a process share a thread group ID, which is the value `getpid()` returns. The per-thread identifier is the TID, returned by `gettid()`. `ps -elf` shows both as PID and LWP.
4. Default thread stack size on glibc is the process stack `rlimit`, typically 8 MiB, reserved as virtual address space and committed lazily. On musl it is 128 KiB. On Windows the default reserve is 1 MiB with 4 KiB committed.
5. That 8 MiB reservation is why you cannot have a million pthreads on a 64-bit system with default settings and a modest `vm.max_map_count`, even though the memory is never touched.
6. Threading models, in the classic taxonomy: 1:1 (one kernel thread per user thread; Linux NPTL, Windows), N:1 (all user threads on one kernel thread; green threads), M:N (a runtime multi-

plexes M user threads onto N kernel threads; Go, Erlang, Java 21 virtual threads).

- Linux abandoned M:N deliberately. NPTL, the Native POSIX Thread Library by Ulrich Drepper and Ingo Molnár, shipped in glibc 2.3.2 in 2003 and replaced LinuxThreads with a strict 1:1 model, because the kernel scheduler was made good enough that a userspace one added nothing.

Model	Kernel sees	Blocking one thread
1:1 NPTL, Windows	Every thread	Others keep running
N:1 green threads	One thread	All stop
M:N Go, Erlang	N carriers	Runtime reschedules

- Thread pool sizing rules of thumb. CPU-bound work: pool size equal to the core count. I/O-bound work: cores multiplied by one plus the ratio of wait time to compute time. Both are heuristics, not laws.
- Real pool defaults: Java's `ForkJoinPool.commonPool` uses cores minus one; .NET's `ThreadPool` starts at the core count and grows by one thread every 500 ms under starvation; nginx uses one worker process per core with an event loop rather than a thread pool.
- Observation: `ps -eLf`, `top -H`, `htop` with thread view on Linux; `/proc/PID/task/` lists one directory per thread; `perf sched` shows switch events; Windows Performance Analyzer and Process Explorer on Windows.

■ 18.5.6 WORDS – remember these

- Thread — one flow through the code — a schedulable execution context with its own stack and register set.
- Thread group — the threads of one process — tasks sharing a TGID, which is what `getpid()` returns on Linux.
- Race condition — two threads touching one thing at once — an outcome that depends on unsynchronized interleaving of accesses.
- Thread pool — reuse threads instead of making new ones — a fixed set of worker threads fed by a task queue.
- 1:1 model — one kernel thread per program thread — the NPTL and Windows model, where the kernel schedules every thread directly.
- M:N model — a runtime shuffles many onto few — user-space scheduling of green threads onto a smaller pool of kernel carriers.

18.6 Scheduling

■ 18.6.1 PLAIN – in simple words

- Your machine seems to run fifty programs at once. If it has eight cores, at most eight instructions are actually being executed at any instant.
- The trick is speed. The system gives each program a tiny slice of time, then switches to the next, and goes round and round.
- Switch fast enough and it looks continuous, in the same way that separate photographs shown quickly look like a moving picture.
- The part of the operating system that decides who goes next is the **scheduler**.
- Its job is to answer one question, over and over, thousands of times a second: of everything that could run right now, which one runs next, and for how long.
- There is no perfect answer, because the goals fight each other.

7. You want the machine to be busy. You want short jobs to finish quickly. You want the mouse pointer to move the instant you move the mouse. You want no program to be forgotten forever.
8. A scheduler that is good at one of those is usually worse at another. Every real scheduler is a compromise that someone chose.

■ 18.6.2 PLAIN - a picture in your head

1. Picture a doctor's surgery with one doctor and a waiting room.
2. **First come first served:** strictly in arrival order. Perfectly fair, and one person with a forty-minute problem makes everyone else late.
3. **Shortest job first:** call in whoever will be quickest. The waiting room empties fastest. Someone with a long problem may sit there all day.
4. **Round robin:** everyone gets five minutes, then goes to the back of the queue if they are not finished. Nobody waits forever. Everyone comes back several times, and each return wastes a minute settling in.
5. **Priority:** emergencies first. Right, and if emergencies keep arriving the ordinary patient never gets seen. That is **starvation**.
6. **Multilevel feedback:** several waiting rooms. Everyone starts in the fast one. If you use your whole slot without finishing you are moved to a slower room with longer slots. Short visits stay fast, long ones do not clog things.

Where this comparison breaks: the receptionist knows roughly how long each appointment will take. A scheduler does not, and cannot: predicting how long a program will run is equivalent to solving the halting problem. Every real scheduler that claims to run the shortest job first is in fact guessing from recent history. And the doctor takes a minute to switch patients; a processor switches in about a microsecond, which changes every calculation.

■ 18.6.3 PLAIN - a worked example

1. Four jobs arrive at a single core. Times are in arbitrary equal units.

Job	Arrives	Needs
A	0	7
B	2	4
C	4	1
D	5	4

2. Total work is 16 units, so whatever we do, the last job finishes at time 16. Only the waiting changes.
3. **First come first served.** A runs 0 to 7, B 7 to 11, C 11 to 12, D 12 to 16.
4. Waiting times: A 0, B 5, C 7, D 7. Average wait 4.75.
5. **Shortest job first, without interrupting.** A runs 0 to 7 because it is the only one there. At 7 the choices are B (4), C (1), D (4). C is shortest, so C runs 7 to 8, then B 8 to 12, then D 12 to 16.
6. Waiting times: A 0, B 6, C 3, D 7. Average wait 4.00.
7. **Shortest remaining time first**, which is the same idea but allowed to interrupt. A starts. At time 2 B arrives needing 4, A has 5 left, so B takes over. At 4 C arrives needing 1, B has 2 left, so C takes over and finishes at
 5. B then finishes 5 to 7. D runs 7 to 11. A finally finishes 11 to 16.
8. Waiting times: A 9, B 1, C 0, D 2. Average wait 3.00. Best of the four, and A paid for it.
9. **Round robin with a slice of 2 units.** New arrivals join the back of the queue before a job that has just been interrupted.

Slice	Runs	Queue afterwards
0 to 2	A	B, A
2 to 4	B	A, C, B
4 to 6	A	C, B, D, A
6 to 7	C done	B, D, A
7 to 9	B done	D, A
9 to 11	D	A, D
11 to 13	A	D, A
13 to 15	D done	A
15 to 16	A done	empty

10. Waiting times: A 9, B 3, C 2, D 6. Average wait 5.00.
11. Put the four side by side.

Algorithm	Avg wait	Avg turnaround
First come first served	4.75	8.75
Shortest job first	4.00	8.00
Shortest remaining first	3.00	7.00
Round robin, slice 2	5.00	9.00

12. Round robin came last on both averages. It is still what real interactive systems use, because averages are not what a human notices.
13. What a human notices is that under round robin, C started 2 units after it arrived and B started immediately. Under first come first served, C sat still for 7 units. Responsiveness is not in the average.

■ 18.6.4 PLAIN – what is really happening inside

1. A **context switch** is the act of taking the processor away from one thread and giving it to another.
2. Here is exactly what gets saved, in order.
3. The general-purpose registers. On x86-64 that is 16 of them, 8 bytes each.
4. The program counter, so the thread resumes at the right instruction. On x86-64 this is already on the kernel stack, pushed by the interrupt.
5. The flags register, which holds the results of the last comparison.
6. The stack pointer.
7. The floating point and vector register state. On x86-64 with AVX-512 this is the big one: over 2,500 bytes. Modern kernels defer this and only save it if the new thread actually uses those registers.
8. Then, if the new thread belongs to a different process, the page table root register is reloaded, which changes what memory is visible.
9. The direct cost of all that is roughly one to three microseconds.
10. The indirect cost is larger and invisible: the new thread finds the caches full of the old thread's data, and runs slowly until it has refilled them. Measured cache-warming costs of 10 to 100 microseconds are common.
11. This is why a scheduler that switches too often makes a machine slower even though it looks fairer.
12. The **time slice** or **quantum** is how long a thread may run before the scheduler is asked again. Typical values are 1 to 10 milliseconds, which is a thousand to ten thousand times the switch cost.
13. **Preemptive** multitasking means the system can take the processor away whether or not the program agrees. This requires the timer interrupt.

14. **Cooperative** multitasking means a program keeps the processor until it chooses to give it up. One program with an infinite loop freezes everything.
15. Cooperative was not a theory. Classic Mac OS up to version 9 and Windows 3.x both worked that way, and both froze exactly as described.

■ 18.6.5 TECHNICAL – the engineer’s version

1. Scheduling metrics, defined precisely. Turnaround time is completion minus arrival. Waiting time is turnaround minus service time. Response time is first-run minus arrival. Throughput is completed jobs per unit time.
2. **Linux CFS**, the Completely Fair Scheduler, was written by Ingo Molnár, inspired by Con Kolivas’s Rotating Staircase Deadline scheduler, and merged in Linux 2.6.23 in October 2007.
3. CFS abolished fixed time slices. It tracked *vruntime*, virtual runtime, per task, scaled by the task’s weight, and always ran the task with the smallest *vruntime*. The ready set was a red-black tree, so picking the next task was $O(\log n)$.
4. Nice values map to weights by a table where each step of 1 changes CPU share by about 10 percent, and nice 0 has weight 1024. Nice ranges from -20 to +19.
5. **EEVDF** replaced CFS in Linux 6.6, released 29 October 2023. The name is Earliest Eligible Virtual Deadline First, from a 1995 paper by Ion Stoica and Hussein Abdel-Wahab.
6. EEVDF gives each task a virtual deadline computed from its requested slice and its lag, and runs the eligible task with the earliest deadline. It handles latency-sensitive tasks properly, which CFS could only do through a growing pile of heuristics and out-of-tree patches.
7. Linux scheduling classes, in strict priority order: `SCHED_DEADLINE`, then the real-time classes `SCHED_FIFO` and `SCHED_RR`, then `SCHED_OTHER` and `SCHED_BATCH`, then `SCHED_IDLE`. Verified on the machine used here with `chrt -m`:


```
SCHED_OTHER min/max priority : 0/0
SCHED_FIFO  min/max priority : 1/99
SCHED_RR    min/max priority : 1/99
SCHED_BATCH min/max priority : 0/0
SCHED_IDLE  min/max priority : 0/0
SCHED_DEADLINE min/max      : 0/0
```
8. A `SCHED_FIFO` task at priority 99 that never blocks will hang the machine, or would if not for the throttle: `sched_rt_runtime_us` defaults to 950000 out of `sched_rt_period_us` of 1000000, so real-time tasks are capped at 95 percent of each second.
9. `SCHED_RR` round-robin slice on this machine, from `/proc/sys/kernel/sched_rr_timeslice_ms`, is 100 ms.
10. **Windows** uses 32 priority levels: 0 is the zero-page thread, 1 to 15 are dynamic, 16 to 31 are real-time. The level is computed from a process priority class and a thread priority offset.

Priority class	Base level
Idle	4
Below normal	6
Normal	8
Above normal	10
High	13
Real-time	24

11. Windows boosts a thread's priority when its wait completes, and boosts the foreground window's thread, then decays the boost by one level per quantum. Quantum on Windows client is 2 clock intervals, roughly 20 to 30 ms; on Server it is 12 intervals, roughly 120 to 180 ms, favouring throughput.
12. macOS uses a multilevel feedback queue with four bands: normal, system high priority, kernel mode only, and real-time. It also exposes Grand Central Dispatch quality-of-service classes, from `QOS_CLASS_USER_INTERACTIVE` down to `QOS_CLASS_BACKGROUND`, which map onto scheduler parameters and, on Apple silicon, onto the choice of performance or efficiency core.
13. Real context switch counts. On the machine used here, `/proc/stat` reported `ctxt 4153017` after 95 minutes of light use, roughly 730 switches per second across two cores.
14. Per-process counters live in `/proc/PID/status` as `voluntary_ctxt_switches` (the task blocked) and `nonvoluntary_ctxt_switches` (the scheduler preempted it). A high nonvoluntary count means CPU contention; a high voluntary count means I/O.
15. Observation and control: `chrt` to read and set policy, `nice` and `renice` for weights, `taskset` for CPU affinity, `perf sched latency` and `perf sched map` for switch-level traces, `pidstat -w` for switch rates, `vmstat 1` for the `cs` column, and `schedtool` for everything at once.
16. Where experts disagree: whether a general-purpose kernel should ever adopt a pluggable scheduler interface. Linux merged `sched_ext`, which allows BPF programs to implement schedulers, in Linux 6.12 in November 2024, after years of objection that it would fragment behaviour. Both sides had a real point.

■ 18.6.6 WORDS – remember these

1. Scheduler — the chooser of who runs next — the kernel component implementing a policy over the set of runnable tasks.
2. Context switch — swapping one thread for another — saving and restoring architectural state, and possibly the address space.
3. Quantum — how long you get before being asked to stop — the scheduling time slice, typically 1 to 10 ms on desktop systems.
4. Preemption — being interrupted whether you like it or not — involuntary removal of a running task, driven by the timer interrupt.
5. Starvation — never getting a turn — indefinite postponement of a low-priority task by a stream of higher-priority ones.
6. Priority inversion — a low task blocking a high one — a high-priority task waiting on a lock held by a low-priority task; fixed by priority inheritance.
7. CFS — Linux's fair scheduler from 2007 — weight-proportional virtual runtime scheduling over a red-black tree.
8. EEVDF — its 2023 replacement — earliest eligible virtual deadline first, merged in Linux 6.6.

18.7 System calls

■ 18.7.1 PLAIN – in simple words

1. Your program cannot open a file. It genuinely cannot. The instruction that would talk to the disk is refused in user mode.
2. So it asks the operating system to do it. That request is a **system call**.
3. A system call is not a function call into a library. It is a controlled jump across the boundary between your program and the kernel.
4. There are only a few hundred of them, and they are the complete list of everything your program can ask the machine to do.
5. Everything else your program does — arithmetic, loops, string handling — needs no permission and never crosses the boundary.
6. So the full power of a program is: any computation it likes, plus this fixed menu of requests.
7. When people say a language “can do system programming”, they mean it can make these calls directly rather than through several layers.
8. Every file you open, every byte you send over the network, every window you draw and every key you read passes through this menu.

■ 18.7.2 PLAIN – a picture in your head

1. Think of a bank counter with thick glass and a small hatch.
2. You are outside. The money is inside. You cannot reach it.
3. There is a printed list of things you may ask for: withdraw, deposit, balance, transfer. Numbered 1 to 40.
4. You fill in a slip: request number, then the details in fixed boxes. You push it through the hatch.
5. The clerk checks the slip, does the work inside where you cannot see, and pushes back a result: either what you asked for, or a refusal code.
6. You cannot ask for anything not on the list. You cannot go behind the counter. You cannot see how the vault works.
7. Filling in a slip and waiting takes longer than counting your own coins. So you learn to ask for a lot at once rather than making forty small requests.

Where this comparison breaks: the clerk is a separate person, whereas the kernel runs on the very same processor as your program, just in a different mode. And the bank clerk cannot look inside your wallet; the kernel can read and write every byte your program owns, and does, in order to fill in your answers. The boundary protects the kernel from you. It does not protect you from the kernel.

■ 18.7.3 PLAIN – a worked example

1. Here is a complete C program:

```
#include <stdio.h>
int main(void) { printf("hello\n"); return 0; }
```

2. Run under `strace`, which prints every system call, it made exactly these, with the boring middle removed:

```
execve("./hello", [ "./hello" ], 0x7ffd... ) = 0
brk(NULL)                                = 0x5587a3d30000
access("/etc/ld.so.preload", R_OK)        = -1 ENOENT
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=53283}) = 0
mmap(NULL, 53283, PROT_READ, MAP_PRIVATE, 3, 0) = ...
```

```

close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6",
        O_RDONLY|O_CLOEXEC)             = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0..."..., 832) = 832
mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE, 3, 0) = ...
mmap(..., PROT_READ|PROT_EXEC, MAP_FIXED, 3, 0x28000)
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f22105ce740) = 0
set_tid_address(0x7f22105cea10)         = 18134
mprotect(0x7f22103ff000, 16384, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024})
fstat(1, {st_mode=S_IFIFO|0600, st_size=0}) = 0
getrandom("\x8a\x08\xd7\x55...", 8, GRND_NONBLOCK) = 8
brk(0x5587a3d51000)                     = 0x5587a3d51000
write(1, "hello\n", 6)                   = 6
exit_group(0)                           = ?

```

3. Now every line, in order.
4. `execve` is the call that loaded this program. It is the first line of every trace because the trace starts in the shell, before the program exists.
5. `brk(NULL)` asks where the heap currently ends. It is a query, not a change.
6. `access("/etc/ld.so.preload")` returned `-1 ENOENT`, meaning the file does not exist. That is not an error; the dynamic linker is checking for an optional override and finding none.
7. `openat` on `/etc/ld.so.cache` returned 3. That number is a **file descriptor**: the handle you use for every later operation on that file. 0, 1 and 2 are already taken by standard input, output and error.
8. `fstat(3, ...)` asks how big it is: 53,283 bytes.
9. `mmap` maps those bytes into memory rather than reading them, so the library cache can be looked at as an array.
10. `close(3)` releases the descriptor. Descriptor 3 is now free again, which is why the very next `openat` also returns 3.
11. The second `openat` finds the C library itself, all 2,125,328 bytes of it.
12. `read(3, "\177ELF...", 832)` reads the first 832 bytes: the ELF header. The first four bytes are `0x7F` then `ELF`, the magic number that identifies the file format.
13. Then four `mmap` calls place the library into memory with different permissions: read-only for constants, read plus execute for code, read plus write for data.
14. `arch_prctl(ARCH_SET_FS, ...)` sets the FS segment base, which is how thread-local storage is found on x86-64.
15. `set_tid_address` tells the kernel where to clear a word when this thread dies, which is how `pthread_join` learns the news.
16. `mprotect(..., PROT_READ)` makes the relocation table read-only after it has been filled in. That is a hardening measure called RELRO.
17. `prlimit64` reads the stack limit: 8,192 times 1,024 bytes, that is 8 MiB.
18. `fstat(1, ...)` checks what standard output is. Here it reported `S_IFIFO`, a pipe. That is why the C library chose full buffering rather than line buffering.
19. `getrandom(..., 8, ...)` fetches 8 random bytes for the stack canary, the guard value that detects buffer overflows.
20. `brk` grows the heap by 132 KiB for the output buffer.
21. `write(1, "hello\n", 6)` is the one call the program actually meant to make. Six bytes to descriptor 1. It returned 6, meaning all six were accepted.
22. `exit_group(0)` ends every thread of the process with status 0.

23. Count them: about twenty-five system calls to print one word, and only one of them was the point. The rest is the cost of dynamic linking.

■ 18.7.4 PLAIN – what is really happening inside

1. Mechanically, on a 64-bit Intel or AMD processor, a system call is this.
2. The program puts a number in the register `rax`. That number says which service it wants. `write` is 1. `read` is 0. `openat` is 257.
3. It puts the arguments in registers, in a fixed order: `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`. Up to six.
4. It executes one instruction: `syscall`.
5. The processor then does several things at once, in hardware. It saves the return address into `rcx` and the flags into `r11`. It switches to privilege ring 0. It loads the instruction pointer from a control register that only the kernel could write.
6. That control register, `MSR_LSTAR`, was set during boot to the address of the kernel's entry stub. So the program lands exactly where the kernel decided, and nowhere else.
7. The kernel stub switches to a kernel stack, saves the rest of the registers, checks that the number in `rax` is within range, and calls the function in the system call table at that index.
8. The function does the work. It may block, in which case the scheduler picks somebody else and this thread resumes here later.
9. The result goes into `rax`. Errors are returned as small negative numbers.
10. The kernel restores registers and executes `sysret`, which puts the processor back into ring 3 at the address in `rcx`.
11. The C library then looks at `rax`. If it is between -1 and -4095, the library stores its negation in `errno` and returns -1 to you.
12. That is why C code checks for -1 and then reads `errno`. The kernel never set `errno`; the library did.
13. Cost, measured on the machine used here: a bare `getpid` system call took 0.089 microseconds, that is 89 nanoseconds, about 187 cycles at 2.10 GHz.
14. That is genuinely expensive next to an ordinary function call of one or two nanoseconds, and cheap next to a disk read of 100 microseconds.
15. Some calls are so common that the kernel arranges for them to need no boundary crossing at all. It maps a small shared page called the **vDSO** into every process, containing code and data that answer them in user mode.
16. Measured on this machine: `clock_gettime` through the vDSO took 0.030 microseconds; forced through the real system call it took 0.144 microseconds. Nearly five times cheaper for doing the same thing.

■ 18.7.5 TECHNICAL – the engineer's version

1. The x86-64 Linux calling convention for system calls: number in `rax`, args in `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`, return in `rax`. Note `r10` rather than `rcx`, because `syscall` overwrites `rcx` with the return address.
2. On AArch64 Linux: number in `x8`, args in `x0` to `x5`, instruction `svc #0`, return in `x0`. On RISC-V: number in `a7`, args `a0` to `a5`, instruction `ecall`.
3. Older x86 mechanisms, in historical order: `int 0x80` software interrupt (slow, about 1,000 cycles), `sysenter` and `sysexit` from Pentium II, `syscall` and `sysret` from AMD64. Windows used `int 0x2E` before switching to `sysenter`.
4. Linux has 300 to 460 system calls depending on architecture and version. Verified numbers from `asm/unistd_64.h` on this machine:

Call	Number	Purpose
read	0	Read from a descriptor
write	1	Write to a descriptor
close	3	Release a descriptor
mmap	9	Map memory or a file
clone	56	Create process or thread
execve	59	Replace process image
exit_group	231	End all threads
openat	257	Open relative to a dir fd

5. Grouped by purpose, the ones worth memorizing:

Group	Calls
Process	fork, clone, execve, wait4
Process	exit_group, kill, getpid
Files	openat, read, write, close
Files	lseek, statx, unlink, rename
Memory	mmap, munmap, mprotect, brk
Network	socket, bind, listen, accept
Network	connect, sendto, recvfrom
Sync	futex, poll, epoll_wait
Devices	ioctl, fcntl

6. `ioctl` is the escape hatch: a single call with a device-specific command number, used for everything that does not fit the read and write model. It is widely regarded as an ugly but necessary design.
7. Counting calls with `strace -c` on `ls /tmp` on this machine gave 76 calls total, 5 of which returned errors. The largest single group was 17 `mmap` calls, all from dynamic linking.
8. Windows equivalents are in `ntdll.dll` as `Nt` or `Zw` functions: `NtCreateFile`, `NtReadFile`, `NtWriteFile`, `NtAllocateVirtualMemory`. The documented API you are meant to call is `Win32`, in `kernel32.dll`, which calls these. Microsoft does not guarantee the numbers, and they change between builds, which is why malware that hardcodes them breaks after updates.
9. macOS syscall numbers are split into classes by the top byte: class 1 is Mach traps, class 2 is BSD calls. `write` is `0x20000004`.
10. Reducing the cost: `io_uring`, merged in Linux 5.1 in May 2019, replaces per operation calls with two shared ring buffers, allowing batched submission and completion with, in the best case, zero system calls per operation.
11. The security cost: Meltdown and Spectre, disclosed 3 January 2018, forced kernel page-table isolation, which roughly doubled or tripled system call cost on affected processors. Measured overheads of 5 to 30 percent on system call heavy workloads were common in 2018.
12. Tools: `strace` and `ltrace` on Linux, `dttruss` and `dttrace` on macOS (which require disabling System Integrity Protection for many targets), Process Monitor and Event Tracing for Windows on Windows, and `bpftool` on modern Linux for anything the others cannot do.

■ 18.7.6 WORDS – remember these

1. System call — asking the kernel for something — a controlled mode transition into a numbered kernel service routine.

2. File descriptor — a small number naming an open thing — an index into the per-process open file descriptor table.
3. `errno` — the reason the last request failed — a thread-local integer set by the C library from the kernel's negative return value.
4. `vDSO` — a shortcut that skips the kernel — a kernel-provided shared object mapped into every process, serving calls like `clock_gettime` in user mode.
5. `ioctl` — the request that covers everything else — a device-specific control call carrying an opaque command number and argument.
6. `io_uring` — batching requests instead of asking one at a time — shared submission and completion ring buffers, in Linux since 5.1.

18.8 Memory management from the operating system's side

■ 18.8.1 PLAIN - in simple words

1. Chapter 11 covered memory in depth. One line to bring it back: every program sees its own private set of addresses starting at zero, and the hardware translates those into real addresses in the physical chips.
2. That translation is done by a unit in the processor using tables the kernel builds. The kernel owns those tables. No program can edit its own.
3. Here we care about the part the kernel does, which is three jobs.
4. Job one: hand out **page frames**. Physical memory is cut into fixed blocks, usually 4 KB. The kernel keeps track of which are used and which are free.
5. Job two: decide what to do when there is not enough. Either free something, or write something to disk, or kill a program.
6. Job three: enforce limits, so one program cannot take everything.
7. A surprising fact: the kernel routinely promises more memory than it has.
8. It does that because programs ask for far more than they use, and most of the promise is never called in.
9. When the promise is called in and the memory does not exist, something has to die. The kernel picks a victim and kills it.

■ 18.8.2 PLAIN - a picture in your head

1. Think of an airline selling seats on a plane with 180 seats.
2. Experience says about 5 percent of people never turn up. So the airline sells 189 tickets. That is **overcommit**.
3. Almost always it works and the plane goes out full. Occasionally everyone turns up, and someone gets removed from the flight.
4. The airline does not remove people at random. It has a rule: latest booking, cheapest fare, traveling alone.
5. The kernel does the same. It has a scoring rule and it kills the highest scorer. That code is called the **out of memory killer**.
6. Memory the program asked for but never touched is a ticket that was never used. The kernel only allocates a real page frame the first time you actually write to the address.

Where this comparison breaks: the airline can offer you money to take a later flight. The kernel cannot negotiate. It sends signal 9 and the process ends instantly with no chance to save anything. And an airline's overbooking is calculated from real statistics; Linux's default overcommit setting is a rough heuristic that guesses, and it can be switched off entirely.

■ 18.8.3 PLAIN – a worked example

1. A program calls `malloc(1024 * 1024 * 1024)`, asking for one gigabyte, on a machine with 8 GB of RAM and 6.7 GB free.
2. The C library sees a large request and calls `mmap` rather than `brk`.
3. The kernel checks its overcommit policy. The default on this machine, from `/proc/sys/vm/overcommit_memory`, is 0: the heuristic mode. It agrees.
4. The kernel records a mapping one gigabyte long in the process's list of memory regions and returns a pointer. **Not one byte of physical memory has been allocated.**
5. `free` reports no change. The program's virtual size, `VSZ` in `ps`, jumps by 1 GB. Its resident size, `RSS`, does not move.
6. The program writes one byte at the start. The processor tries to translate that address, finds no page table entry, and raises a **page fault**.
7. The kernel handles the fault, takes one free 4 KB frame, zeroes it, points the table entry at it, and restarts the instruction. `RSS` goes up by 4 KB.
8. Only when the program walks the whole gigabyte does the kernel hand over 262,144 frames and the memory is really consumed.
9. Now suppose it does that when only 200 MB is free. The kernel first tries to reclaim: it drops cached file pages, and swaps out unused anonymous pages.
10. If that is not enough, the out-of-memory killer runs. It reads a score for every process and kills the highest.
11. On this machine the score for a small shell process was 666 out of 1000, from `/proc/self/oom_score`, and the tunable `/proc/self/oom_score_adj` was 0. Setting the adjustment to -1000 makes a process immune.
12. In the kernel log you would then see a line beginning `Out of memory: Killed process 1234 (chrome)`, followed by a full report of who had what.

■ 18.8.4 PLAIN – what is really happening inside

1. The kernel divides physical memory into page frames and keeps one small record per frame. On Linux that is `struct page`, about 64 bytes each.
2. For 8 GB of RAM with 4 KB pages, that is 2,097,152 frames and about 128 MB of records. The kernel's own bookkeeping costs about 1.6 percent of memory.
3. Free frames are managed by the **buddy allocator**: free memory is held in lists of blocks of 1, 2, 4, 8 up to 1024 pages. Splitting and merging is done by halving and pairing, which keeps fragmentation manageable.
4. Small kernel objects do not each take a page. A second allocator, the slab allocator, carves pages into same-sized objects with free lists. You can see it in `/proc/slabinfo`.
5. Memory that holds file contents is not wasted. The **page cache** keeps recently read file data in otherwise free memory and hands it back instantly when reclaimed. That is why "free memory" on a healthy Linux box is near zero and this is correct, not a problem.
6. When memory runs short the kernel walks its lists of pages and decides what to evict, approximately least-recently-used. Clean file pages are cheapest to drop, because the copy on disk is still valid. Dirty pages must be written first. Anonymous pages, which have no file, must go to swap.
7. If there is no swap and nothing left to reclaim, the only remaining action is to kill something.
8. The out-of-memory score is roughly proportional to the process's resident memory plus its swap use, expressed out of 1000, then adjusted by `oom_score_adj`. The biggest user usually dies. That is deliberate: killing the biggest recovers the most memory for the fewest deaths.

■ 18.8.5 TECHNICAL – the engineer’s version

1. Linux overcommit modes, in `/proc/sys/vm/overcommit_memory`:

Value	Name	Behaviour
0	Heuristic	Refuse only absurd requests
1	Always	Never refuse anything
2	Strict	Refuse beyond the limit

2. In mode 2, the limit is swap plus `overcommit_ratio` percent of RAM. The default ratio is 50, verified on this machine. With 8 GB and no swap, that allows only 4 GB of committed address space in total, which is why mode 2 is rarely used without tuning.
3. Windows does not overcommit in the Linux sense. It maintains a commit charge against a commit limit equal to RAM plus the pagefile, and `VirtualAlloc` with `MEM_COMMIT` fails when the limit is reached. The failure is returned to the program rather than delivered later as a kill.
4. That is a genuine design difference with consequences. Linux gives you better memory utilization and a risk of sudden death; Windows gives you honest allocation failures and requires a large pagefile.
5. Page sizes: 4 KiB base on x86-64 and on Linux ARM64 by default; 2 MiB and 1 GiB huge pages on x86-64; 16 KiB base on Apple silicon macOS and on iOS. Transparent Huge Pages on Linux promotes eligible 2 MiB regions automatically and can be controlled per process with `madvise`.
6. Per-process limits come from `setrlimit` and are visible in `/proc/PID/limits`. `RLIMIT_AS` caps total address space, `RLIMIT_DATA` the heap, `RLIMIT_STACK` the main stack (8 MiB soft on this machine), `RLIMIT_MEMLOCK` how much may be pinned.
7. Cgroups give stronger limits than `rlimits`, because they apply to a group and are enforced by the reclaim path. In cgroup v2, `memory.max` is a hard cap that triggers cgroup-local OOM kill, `memory.high` is a throttling threshold, and `memory.low` protects a floor during reclaim.
8. Useful counters, all real files: `/proc/meminfo` for the global picture, `/proc/PID/status` for `VmRSS`, `VmSize` and `VmSwap`, `/proc/PID/smmaps` for per-mapping detail, and `/proc/pressure/memory` for PSI stall percentages, which is the single best early warning of memory trouble.
9. Reading `/proc/meminfo` on this machine gave `MemTotal: 8216168 kB`, `MemFree: 6709900 kB`, `MemAvailable: 7407208 kB` and `Cached: 847024 kB`. `MemAvailable` is the number to trust; `MemFree` ignores reclaimable cache.
10. Tools: `free -m`, `vmstat 1`, `smem`, `ps -eo pid,rss,vsz --sort=-rss`, `pmap`, `slabtop`, and `dmesg | grep -i "killed process"` to find OOM events after the fact. On macOS, `vm_stat`, `footprint` and Activity Monitor’s memory pressure graph.

■ 18.8.6 WORDS – remember these

1. Page frame — one fixed block of real memory — a physical page, usually 4 KiB on x86-64.
2. Page fault — the translation was missing — a trap raised when a virtual address has no valid mapping, handled by the kernel.
3. Overcommit — promising more than you have — allowing total mapped address space to exceed RAM plus swap.
4. OOM killer — the code that picks who dies — the out-of-memory handler that selects and `SIGKILLs` the highest scoring process.
5. Page cache — free memory holding file data — the unified cache of file-backed pages, reclaimable without I/O when clean.

6. Buddy allocator — splitting and pairing free blocks — the power-of-two physical page allocator underneath the kernel's memory subsystem.

18.9 The filesystem from the operating system's side

■ 18.9.1 PLAIN - in simple words

1. A disk stores numbered blocks. It knows nothing about files, names or folders. Those are entirely the operating system's invention.
2. The filesystem is the code that turns "a file called notes.txt in a folder called work" into "blocks 41,922 to 41,930 on this disk".
3. There are dozens of filesystems: ext4, NTFS, APFS, XFS, Btrfs, FAT32, ZFS. They all store data differently.
4. Your programs do not know which one they are using, and should not care.
5. That works because the kernel has a layer in the middle that presents one uniform set of operations: open, read, write, close, list.
6. Every filesystem implements that same set. The layer picks the right one based on where the file lives.
7. **Mounting** is how a filesystem gets attached to the tree of names. You say "this disk appears at /media/photos" and from then on it does.
8. When you open a file, you get back a small number. Every later operation uses that number. The kernel remembers everything else.
9. Permissions decide who may do what. On UNIX they are simple and were designed in 1971. On Windows they are elaborate lists.

■ 18.9.2 PLAIN - a picture in your head

1. Think of a large library.
2. The shelves hold numbered boxes. Box 41,922 contains some pages. Boxes know nothing about titles. That is the disk.
3. The card catalogue maps a title to a list of box numbers. That is the filesystem's index, called the inode table on UNIX.
4. The front desk is where you ask for a book. You give a title; they hand you a ticket with a number on it, say ticket 3. That is your file descriptor.
5. From then on you say "more from ticket 3" and they know exactly which book and which page you were on. You never say the title again.
6. Two people can hold two tickets for the same book, each on a different page. Two descriptors, one file, two positions.
7. Adding a new wing to the library and declaring that its shelves count as part of aisle F is mounting.

Where this comparison breaks: a library has one physical copy of a book, so two readers cannot both have it. A filesystem hands out as many descriptors as are asked for, and if two writers write at once without coordinating, the result is a mixture of both. The filesystem does not lock anything for you unless you ask.

■ 18.9.3 PLAIN - a worked example

1. Real permission bits, produced on the machine used here by creating files and running `ls -l`:

```
-rw-r--r-- 1 root root    0 a.txt      644
-rwxr-x--- 1 root root    0 b.sh       750
drwxr-sr-x 2 root root 4096 d          2755
```

```
-rwsr-xr-x 1 root root 0 s 4755
-rwxrwxrwt 1 root root 0 t 1777
```

2. Read the first column in four pieces. First character: file type. - is an ordinary file, d a directory, l a symbolic link, c a character device, b a block device, s a socket, p a pipe.
3. Then three groups of three: what the **owner** may do, what the **group** may do, what **everyone else** may do.
4. In each group: r read, w write, x execute. A dash means not allowed.
5. So -rw-r--r-- is: owner may read and write; group may read; others may read. Nobody may execute.
6. The numeric form packs each group into one octal digit, with read worth 4, write worth 2, execute worth 1. Owner 4+2=6, group 4, others 4, giving 644.
7. b.sh at 750 is owner read, write and execute (7), group read and execute (5), others nothing (0).
8. Now the odd ones. s shows -rwsr-xr-x, mode 4755. The s where the owner's x should be is the **setuid** bit. It means: when this program runs, it runs as the file's owner, not as you.
9. That is how /usr/bin/passwd works. Verified on this machine, it really is -rwsr-xr-x 1 root root 64152. You need to change a root-owned file, so the program briefly becomes root.
10. Setuid is the single most dangerous bit in UNIX. A bug in a setuid-root program is a way for any user to become root. Modern systems replace it with capabilities wherever they can.
11. d shows drwxr-sr-x, mode 2755. The s in the group position on a directory is **setgid**: new files created inside inherit the directory's group. That is how shared project folders work.
12. t shows -rwxrwxrwt, mode 1777. The final t is the **sticky bit**. On /tmp it means: anyone may create files here, but you may only delete your own. Without it, any user could delete anyone's temporary files.
13. On a directory the bits mean something different from a file: r lists the names, w creates and deletes entries, x allows you to traverse into it.
14. A directory with x but not r is real and useful: you can open a known path inside it but cannot see what is there.

■ 18.9.4 PLAIN – what is really happening inside

1. The **virtual filesystem** layer, VFS, is a set of function pointers. Each filesystem registers its own implementations of open, read, write, lookup and about forty others.
2. When your program calls read, the kernel finds the file's inode, follows its pointer to the operations table, and calls whichever function is there. ext4's read or NFS's read or procs's read.
3. That indirection is why /proc/self/status can be read like a file even though there is no such file on any disk. Reading it runs a kernel function that generates the text on the spot.
4. There are three tables involved in open files, and confusing them is the most common source of misunderstanding.
5. Table one: the **file descriptor table**, one per process. It maps small integers to entries in table two.
6. Table two: the **open file table**, system-wide. Each entry holds the current position, the access mode, and a pointer into table three.
7. Table three: the **inode table**. One entry per actual file, holding size, owner, permissions, timestamps and the block map.
8. Now the behaviours make sense. dup copies a descriptor entry, so both point at the same open file entry, so both share one position. open on the same file twice makes two open file entries, so two independent positions.
9. And after fork, parent and child have separate descriptor tables that point at the same open file entries, so their positions move together.

10. Real descriptors, from `/proc/self/fd` on this machine:

```
0 -> /dev/null
1 -> /tmp/.../output
2 -> /tmp/.../output
3 -> /proc/6536/fd
```

11. Note that 1 and 2 point at the same file: standard output and standard error were both redirected to one place.
12. `open` does this: parse the path one component at a time, checking traverse permission on each directory; find the final inode; check the requested access against the permission bits; allocate an open file entry; find the lowest free descriptor number; return it. The rule that it is always the lowest free number is a standard, and shell redirection depends on it.
13. `read` does this: look up the descriptor, take the position, ask the filesystem for those bytes, which usually come from the page cache, copy them into the program's buffer, advance the position, return the count.
14. `write` usually does not touch the disk at all. It copies into the page cache and marks it dirty. A kernel thread writes it out later. `fsync` is what forces it out now.
15. `close` releases the descriptor and decrements a reference count on the open file entry. Only when that count reaches zero is the entry freed.

■ 18.9.5 TECHNICAL – the engineer's version

1. Linux VFS core objects: `struct super_block` for a mounted filesystem, `struct inode` for a file, `struct dentry` for a name-to-inode cache entry, and `struct file` for an open instance.
2. The dentry cache is the reason repeated path lookups are fast. Without it, opening `/usr/share/doc/x/y` would require reading four directories from disk every time.
3. Mounting attaches a superblock at a mountpoint. On this machine `/proc/mounts` showed real entries:

```
proc /proc proc rw,relatime 0 0
sysfs /sys sysfs rw,relatime 0 0
devtmpfs /dev devtmpfs rw,size=4102960k,mode=755 0 0
tmpfs /dev/shm tmpfs rw,size=8216168k 0 0
devpts /dev/pts devpts rw,mode=600 0 0
```

4. Note that `proc`, `sysfs`, `devtmpfs` and `tmpfs` have no disk at all. They are kernel interfaces wearing a filesystem's clothes. That is the VFS abstraction paying off.
5. UNIX permission checking order is strict: if you are the owner, only the owner bits apply, even if the group bits are more permissive. Then group, then other. First match wins, and it is not a union.
6. `umask` masks off bits from newly created files. The default 022 verified here turns a requested 666 into 644 and a requested 777 into 755.
7. POSIX access control lists, from the withdrawn POSIX 1003.1e draft 17 but implemented anyway, add named users and groups: `setfacl -m u:alice:rw file`, read back with `getfacl`. A file with an ACL shows a + after its mode in `ls -l`.
8. Windows uses a completely different model. Every securable object has a security descriptor containing an owner SID, a group SID, a DACL (discretionary access control list) and a SACL (system ACL, for auditing).
9. A DACL is an ordered list of access control entries, each granting or denying specific rights to a SID. There are far more than three rights: `FILE_READ_DATA`, `FILE_WRITE_ATTRIBUTES`, `DELETE`, `WRITE_DAC`, `WRITE_OWNER`, `SYNCHRONIZE` and more.

10. Evaluation order matters: deny entries are processed before allow entries, and inherited entries after explicit ones. Tools: `icac ls` on the command line, the Security tab in Explorer.
11. Filesystem comparison, all figures from the filesystems' own documentation:

Filesystem	Max file size	Journal
ext4	16 TiB	Yes
XFS	8 EiB	Yes
NTFS	8 PiB (Win 10+)	Yes
APFS	8 EiB	Copy on write
Btrfs	16 EiB	Copy on write
FAT32	4 GiB minus 1	No

12. FAT32's 4 GiB limit is the reason a single large video file cannot be copied to many USB sticks as sold. It is a 32-bit size field, not a bug.
13. Tools: `ls -l` and `fuser` to see who has a file open, `df` and `du` for space, `stat` for one file's metadata, `mount` and `findmnt` for the mount tree, `tune2fs -l` for ext4 superblock details, `debugfs` for raw inspection, `fsck` for repair.

■ 18.9.6 WORDS – remember these

1. Filesystem — the scheme that turns names into blocks — the on-disk format plus the driver implementing it.
2. VFS — one interface over many filesystems — the virtual filesystem switch, dispatching operations through per-filesystem function tables.
3. Inode — the record describing one file — metadata and block map, identified by number rather than by name.
4. File descriptor — the ticket you get from `open` — a per-process index into the descriptor table, always the lowest free number.
5. Mount — attaching a filesystem into the name tree — binding a superblock to a directory so paths below it resolve there.
6. `setuid` — the program runs as its owner, not you — mode bit 4000, causing the effective UID to be set from the file's owner on `exec`.
7. Sticky bit — you may only delete your own files here — mode bit 1000 on a directory, restricting deletion to owners.
8. Page cache write-back — write returns before the disk is touched — buffered writes marked dirty and flushed later, forced by `fsync`.

18.10 Inter-process communication

■ 18.10.1 PLAIN – in simple words

1. Processes are deliberately kept apart. That is the whole point of the design.
2. But they often need to work together, so the operating system provides controlled ways to pass information across the wall.
3. There are six that matter, and they differ mainly in three questions: how fast, how much can you send, and can the two sides be on different machines.
4. **Pipes**: a one-way stream of bytes from one process to another. Made by the shell every time you use the `|` symbol.
5. **Named pipes**: the same thing, but with a name in the filesystem, so unrelated processes can find it.

6. **Signals:** a single number sent to a process to interrupt it. Almost no information, but it arrives even if the target is stuck.
7. **Message queues:** a post box holding whole messages with boundaries preserved, so a 20-byte message arrives as 20 bytes and not as part of a stream.
8. **Shared memory:** two processes agree to see the same physical memory. The fastest possible, because nothing is copied.
9. **Sockets:** a two-way connection, which works between processes on one machine and between machines across a network, with the same code.
10. **Remote procedure call:** a layer on top that makes a call to another machine look like an ordinary function call in your program.

■ 18.10.2 PLAIN – a picture in your head

1. Two offices in the same building need to exchange work.
2. A pipe is a chute in the wall. Papers go in one end and come out the other, in order, one way only. Simple and fast.
3. A named pipe is the same chute with a label on it, so anyone in the building can find it rather than only the two who installed it.
4. A signal is a fire alarm. It carries one bit of meaning, you cannot send data with it, and it interrupts whatever the other office was doing.
5. A message queue is a set of pigeonholes. Each item stays a separate item. Nothing merges.
6. Shared memory is knocking a hole in the wall and putting one desk half in each room. Both sides see the same paper immediately. Nobody carries anything. And both sides can grab the same sheet at once, so you need a rule about whose turn it is.
7. A socket is a telephone line, which happens to work identically whether the other office is next door or in another country.

Where this comparison breaks: a chute holds an unlimited pile of paper. A real pipe holds a fixed amount, 64 KiB by default on Linux, and when it is full the writer simply stops until the reader catches up. That blocking behaviour is not a flaw, it is the flow control that makes pipelines work on files larger than memory.

■ 18.10.3 PLAIN – a worked example

1. This shell command:


```
| sh -c 'echo hello world | tr a-z A-Z'
```
2. Traced with `strace -f`, the kernel calls were exactly these, with process IDs at the left:

```
18231 pipe2([3, 4], 0)           = 0
18231 clone(...SIGCHLD...)      = 18232
18231 close(4)                  = 0
18232 close(3)                  = 0
18232 dup2(4, 1)                = 1
18232 close(4)                  = 0
18232 write(1, "hello world\n", 12) = 12
18232 +++ exited with 0 +++
18231 clone(...SIGCHLD...)      = 18233
18231 close(3)                  = 0
18233 dup2(3, 0)                = 0
18233 close(3)                  = 0
18233 execve("/usr/bin/tr", ["tr", "a-z", "A-Z"]) = 0
18233 read(0, "hello world\n", 8192) = 12
```



```

18233 read(0, "", 8192)           = 0
18233 write(1, "HELLO WORLD\n", 12) = 12
18231 wait4(-1, [exited 0], 0, NULL) = 18232
18231 wait4(-1, [exited 0], 0, NULL) = 18233

```

3. Line by line. pipe2 creates the pipe and returns two descriptors at once: 3 is the read end, 4 is the write end. One object, two handles.
4. clone makes the first child, PID 18232, which will run echo. It inherits both descriptors.
5. The parent closes 4, its copy of the write end, because it will not write.
6. The child closes 3, its copy of the read end, because it will not read.
7. dup2(4, 1) is the key move. It makes descriptor 1, standard output, point at the same open file entry as descriptor 4. Now anything written to standard output goes into the pipe.
8. The child then closes 4, because 1 already refers to the pipe. Two handles to the same thing are no longer needed.
9. write(1, "hello world\n", 12) puts 12 bytes into the pipe buffer.
10. The second child, 18233, does the mirror image: dup2(3, 0) makes standard input read from the pipe, then execve replaces it with /usr/bin/tr.
11. Notice that tr does none of this. By the time tr starts, the plumbing is already done and it just reads standard input like any program. That is why every UNIX tool composes.
12. read(0, "hello world\n", 8192) gets the 12 bytes. The next read returns 0.
13. That zero is end-of-file, and it happens only because every copy of the write end has been closed. If any process had kept one open, tr would have waited forever. Forgetting to close the write end is the classic pipe bug.
14. Finally write(1, "HELLO WORLD\n", 12) sends the result to the terminal, and the shell reaps both children with wait4.

■ 18.10.4 PLAIN – what is really happening inside

1. A pipe is a fixed-size circular buffer in kernel memory with two ends and no name. 64 KiB on Linux by default, adjustable with fcntl and F_SETPIPE_SZ.
2. Writing to a full pipe blocks. Reading an empty pipe with writers still open blocks. Reading an empty pipe with no writers returns 0. Writing to a pipe with no readers delivers SIGPIPE, which kills the writer by default.
3. That last rule is why head on a huge file stops quickly rather than reading everything: head exits, the pipe loses its reader, the producer is killed.
4. A signal is a bit set in the target's task structure. Nothing is delivered immediately. The next time that process is about to return to user mode, the kernel notices the bit and diverts it to the handler.
5. That is why a process stuck in uninterruptible kernel sleep does not respond to signals: it is never about to return to user mode.
6. Signal 9, SIGKILL, and signal 19, SIGSTOP, cannot be caught or ignored, because they are handled by the kernel itself rather than delivered.
7. Shared memory is the only mechanism with no copying. The kernel maps the same physical page frames into two processes' page tables. After setup, the kernel is not involved at all.
8. That is also its danger. With no kernel in the loop there is no ordering, so the two sides must agree on synchronization themselves, usually with a semaphore or an atomic flag plus memory barriers.
9. Sockets carry a protocol stack even when both ends are on one machine. A UNIX domain socket skips the network stack and is roughly twice as fast as a loopback TCP socket for the same data.

■ 18.10.5 TECHNICAL – the engineer’s version

Mechanism	Copies	Boundaries	Across machines
Pipe	2	Stream	No
Named pipe / FIFO	2	Stream	No
Signal	0	1 number	No
POSIX mq	2	Messages	No
Shared memory	0	You decide	No
UNIX socket	2	Both modes	No
TCP socket	2+	Stream	Yes

- Two copies means user to kernel and kernel to user. `splice` and `vmsplice` on Linux can reduce pipe transfers to one copy or zero by moving page references instead of bytes.
- Signals: 31 standard signals plus 32 real-time signals on Linux. Real-time signals queue and carry a small value; standard signals do not queue, so two `SIGUSR1` sent quickly may be delivered once.
- A demonstration on this machine sent `SIGUSR1` to itself three times in a tight loop and the handler ran three times, because each was delivered before the next was raised. Send them faster than delivery and you would lose some.
- The first six signals as printed by `kill -l` here: 1 `SIGHUP`, 2 `SIGINT`, 3 `SIGQUIT`, 4 `SIGILL`, 5 `SIGTRAP`, 6 `SIGABRT`. Also worth knowing: 9 `SIGKILL`, 11 `SIGSEGV`, 13 `SIGPIPE`, 15 `SIGTERM`, 17 `SIGCHLD`, 19 `SIGSTOP`.
- `SIGTERM` asks politely and can be caught for cleanup. `SIGKILL` cannot. A shutdown script that sends only `SIGKILL` loses data; `systemd` sends `SIGTERM`, waits `DefaultTimeoutStopSec` (90 s by default), then `SIGKILL`.
- System V IPC, from AT&T System V in 1983: `msgget`, `semget`, `shmget`, with keys instead of file descriptors and no automatic cleanup. Limits on this machine from `ipcs -l`: 32,000 message queues maximum, 8,192-byte maximum message size, 4,096 shared memory segments.
- POSIX IPC, IEEE 1003.1b-1993, is the modern replacement: `mq_open`, `sem_open`, `shm_open`, all using file-descriptor semantics and names under `/dev/shm` and `/dev/mqueue`.
- Modern Linux additions: `eventfd`, `signalfd` and `timerfd` turn events, signals and timers into readable descriptors, so they can be waited on by `epoll` alongside sockets. `memfd_create` gives an anonymous file for sharing.
- Higher-level buses: D-Bus on Linux desktops, Mach ports on macOS, Binder on Android, ALPC on Windows. All are message passing with identity and permissions on top.
- Remote procedure call history: Birrell and Nelson’s 1984 paper *Implementing Remote Procedure Calls* set the model. Sun RPC, RFC 1057 in 1988, powered NFS. Modern systems use gRPC over HTTP/2 with Protocol Buffers, or JSON over HTTP.
- The honest version, from Waldo, Wyant, Wollrath and Kendall’s 1994 Sun paper *A Note on Distributed Computing*: a remote call is not a local call. It can be slow, it can fail halfway, and it can succeed while the reply is lost. Any RPC layer that hides this will eventually mislead you.
- Tools: `lssof -p PID` to see pipes and sockets, `ss -x` for UNIX sockets, `ipcs` and `ipcrm` for System V objects, `strace -e trace=ipc`, and `dbus-monitor` for D-Bus traffic.

■ 18.10.6 WORDS – remember these

- Pipe — a one-way chute between processes — an anonymous kernel ring buffer with a read end and a write end, 64 KiB by default on Linux.
- FIFO — a pipe with a name — a filesystem object that unrelated processes can open to reach the same buffer.

3. Signal — a numbered interruption — an asynchronous notification delivered on return to user mode; SIGKILL and SIGSTOP cannot be caught.
4. Shared memory — the same physical pages in two processes — zero-copy IPC requiring caller-supplied synchronization.
5. Socket — a connection endpoint — a descriptor supporting stream or datagram communication, local via AF_UNIX or remote via AF_INET.
6. RPC — calling a function on another machine — a stub-and-skeleton layer that marshals arguments over a transport, with failure modes a local call lacks.

18.11 Device management and drivers

■ 18.11.1 PLAIN – in simple words

1. Chapter 13 covered devices and buses. One line to bring it back: a device is a lump of hardware that the processor talks to through registers, memory regions and interrupts.
2. A **driver** is the piece of software that knows how to talk to one specific device. It is the translator between the general and the particular.
3. The operating system's job here is to make every disk look like a disk, so that the filesystem code does not need to know which brand it is.
4. It does that by defining an interface, and requiring every driver to implement it: open, close, read, write, control.
5. On UNIX systems, devices then appear as files in /dev, so programs can talk to hardware using the same calls they use for files.
6. **Hot plug** is the ability to add and remove hardware while running. The kernel notices, loads a driver, and tells userspace.
7. The **I/O scheduler** decides in what order pending disk requests are sent, because order matters enormously for speed on a spinning disk and matters less on a solid-state one.

■ 18.11.2 PLAIN – a picture in your head

1. Think of an electrical socket standard.
2. A kettle, a lamp and a laptop charger are completely different inside. They all end in the same plug.
3. The socket is the interface. The plug is the driver. The building's wiring does not know or care what is on the other end.
4. A new appliance arrives, you plug it in, and the building needs no changes. That is a loadable driver.
5. If a badly made appliance shorts out, it can trip the whole building, because it is connected directly to the mains. That is why a bad driver crashes the kernel.

Where this comparison breaks: an appliance can only draw power, whereas a driver runs inside the kernel with the same privileges as the kernel itself. A closer comparison would be an appliance that could rewire the building. This is exactly why driver signing exists, and why microkernels move drivers out of the kernel.

■ 18.11.3 PLAIN – a worked example

1. Real device files from /dev on the machine used here:

```
crw-rw-rw- 1 root root 1, 3 /dev/null
crw-rw-rw- 1 root root 1, 5 /dev/zero
```

```
crw-rw-rw- 1 root root 1, 8 /dev/random
brw----- 1 root root 254, 0 /dev/vda
```

2. The first character is the device class. **c** means **character device**: a stream of bytes, read one after another, no seeking. **b** means **block device**: a fixed-size array of blocks you can jump around in.
3. Where a normal file would show its size, a device file shows two numbers. `/dev/null` is 1, 3.
4. The first is the **major number**: which driver handles this. Major 1 is the kernel's memory device driver. Major 254 here is virtio block.
5. The second is the **minor number**: which device that driver should use. Minor 3 is null, minor 5 is zero, minor 8 is random. Same driver, three behaviours.
6. So `/dev/null` contains no data at all. It is a name, a type and two integers. Everything it does is code in the kernel.
7. Writing to `/dev/null` calls a function that discards the bytes and reports success. Reading from it returns end of file immediately.
8. Permissions on device files are ordinary UNIX permissions, and they are the access control. `/dev/vda`, the raw disk, is `brw-----`: only root may open it. `/dev/null` is `crw-rw-rw-`: anyone.

■ 18.11.4 PLAIN – what is really happening inside

1. When you open `/dev/null`, the VFS sees the inode is a character device, reads major 1 minor 3 from it, and looks up major 1 in the character device table.
2. It replaces the file's operations table with that driver's, so every later read and write goes to driver code.
3. The device file is therefore not a link to hardware. It is a name that carries two numbers, and the numbers select code.
4. Drivers do their work in two halves, because interrupts must be short.
5. The top half runs when the interrupt arrives. It does the minimum: acknowledge the device, grab the data, and schedule the rest.
6. The bottom half runs slightly later with interrupts enabled, and does the real work. Linux calls these softirqs, tasklets and workqueues.
7. Hot plug on Linux works like this: the bus controller raises an interrupt, the bus driver reads the new device's identity numbers, the kernel matches those numbers against every driver's table of supported devices, and loads the one that claims it.
8. It then sends an event up to userspace over a netlink socket, where udev receives it and creates the device file, sets permissions and may run rules.
9. The I/O scheduler sits between the filesystem and the driver. It holds requests briefly so it can merge adjacent ones and reorder them.
10. On a spinning disk that reordering is worth an enormous amount, because a head seek costs about 5 to 10 milliseconds and a reordering that avoids one saves more time than all the software involved.
11. On an SSD there is no head, so reordering saves much less, and the modern default is a scheduler that mostly stays out of the way.

■ 18.11.5 TECHNICAL – the engineer’s version

1. Linux device model: `struct bus_type`, `struct device`, `struct device_driver` and `struct class`, all exposed under `/sys`. `/sys/bus/pci/devices` lists every PCI device; `/sys/class/net` lists every network interface.
2. Driver binding is by ID table. A PCI driver declares a `struct pci_device_id[]` of vendor and device IDs; the core matches and calls the driver’s probe. Modules carry these tables in a `.modalias` section so `udev` can load them on demand.
3. Character devices are registered with `alloc_chrdev_region` and `cdev_add`; block devices go through the block layer, which today is `blk-mq`, the multi-queue block layer merged in Linux 3.13 in 2014 and made the only path in Linux 5.0 in 2019.
4. Linux I/O schedulers available in the multi-queue layer:

Scheduler	Best for	Note
none	NVMe SSD	No reordering at all
mq-deadline	Mixed, SATA SSD	Deadline per request
bfq	Desktop, HDD	Fair queueing, latency
kyber	Fast multi-queue	Target latency tuning

5. Read and set it per device: `cat /sys/block/sda/queue/scheduler` shows the list with the active one in brackets; writing a name changes it.
6. Windows uses the Windows Driver Model and, since Vista, the Windows Driver Frameworks. KMDF drivers run in kernel mode; UMDF drivers run in user mode, which is Windows quietly adopting the microkernel argument for classes of device where the performance cost is acceptable.
7. Driver signing: 64-bit Windows has required kernel-mode driver signatures since Vista in 2006, and since Windows 10 version 1607 in 2016 new kernel drivers must be signed through the Microsoft hardware developer portal.
8. macOS deprecated kernel extensions and replaced them with DriverKit and System Extensions from macOS 10.15 Catalina in 2019. DriverKit drivers run in user space. On Apple silicon, loading a kext requires lowering the security policy and rebooting.
9. Real interrupt counts from `/proc/interrupts` on the machine used here: IRQ 26 is `ttyS0`, the serial port, with 12,584 interrupts on CPU0 and 0 on CPU1. Interrupt affinity, not chance, put them all on one core.
10. Tools: `lspci -v`, `lsusb`, `lsblk`, `lsmod`, `modinfo`, `udevadm monitor`, `dmesg` on Linux; `system_profiler` and `ioreg` on macOS; Device Manager, `pnputil` and `driverquery` on Windows.

■ 18.11.6 WORDS – remember these

1. Driver — code that knows one specific device — a kernel or user-mode module implementing a standard interface for particular hardware.
2. Device file — a name that selects driver code — a filesystem node carrying a type plus major and minor numbers.
3. Major number — which driver — the index selecting a registered driver in the character or block device table.
4. Minor number — which instance — the value the driver interprets to pick a device or behaviour.
5. Top half and bottom half — do the urgent bit now, the rest later — interrupt handler versus deferred processing in `softirq` or `workqueue` context.
6. I/O scheduler — deciding the order of disk requests — the block layer elevator merging and re-ordering requests, chosen per device.

18.12 Protection and isolation

■ 18.12.1 PLAIN – in simple words

1. Everything in this chapter that stops one program hurting another comes from one idea: some code is trusted and some is not, and the hardware knows which.
2. The processor runs in **user mode** or **kernel mode**. In user mode the dangerous instructions simply fail.
3. A program cannot reach another program's memory because the address it uses is not a real address. It is translated, and the translation table only contains its own memory.
4. If it tries an address that is not in its table, the processor raises a fault, and the kernel usually kills it. That is the segmentation fault.
5. On top of that base, systems add finer tools.
6. **Capabilities** cut root's absolute power into pieces, so a program can be allowed to bind a low port without being allowed to do everything.
7. **Sandboxes** shrink what a program may ask for at all, by filtering the system call list.
8. **Containers** give a process a private view of the filesystem, the process list, the network and the users, while sharing one kernel.
9. **Virtual machines** go further and give a whole fake computer, with its own kernel, on top of a program called a hypervisor.
10. When the trusted code itself fails, there is nothing above it to catch the error, so the machine stops. That is a kernel panic or a blue screen.

■ 18.12.2 PLAIN – a picture in your head

1. Think of a building with staff passes.
2. Kernel mode is the master key. User mode is a pass that opens only your own office.
3. Capabilities are what happens when you cut the master key into forty separate keys and hand out only the ones each person needs.
4. A sandbox is an office where most of the doors have been bricked up, so even a valid key opens nothing.
5. A container is a floor of the building with its own numbering, its own directory board and its own staff list. It feels like a whole building. Walk down the fire stairs far enough and it is still the same building, with one boiler serving everything. That boiler is the shared kernel.
6. A virtual machine is a separate building, with its own boiler, built inside a warehouse. Much stronger separation, much more concrete.

Where this comparison breaks: a wall in a building is physical, and no clever argument gets you through it. Processor isolation is enforced by logic that can have design flaws, which is exactly what Spectre and Meltdown were in 2018: correct permission checks defeated by timing side effects of speculation. The walls were real. Information still leaked around them.

■ 18.12.3 PLAIN – a worked example

1. Namespaces are the mechanism containers are built from. Every process on Linux belongs to one of each kind. Real listing from `/proc/self/ns` on this machine:

```
cgroup -> cgroup:[4026531835]
ipc    -> ipc:[4026531839]
mnt    -> mnt:[4026531832]
net    -> net:[4026531833]
pid    -> pid:[4026531836]
time   -> time:[4026531834]
```



```
user    -> user:[4026531837]
uts     -> uts:[4026531838]
```

- Those numbers are namespace identifiers. Two processes with the same number share that view of the world.
- Run a container and it gets different numbers for some of these. Same kernel, different view.
- A process in a new PID namespace sees itself as PID 1 and cannot see any process outside. `ps` inside a container genuinely shows only the container's processes, because the kernel is answering a different question.
- A process in a new mount namespace has its own filesystem tree. That is how a container's / can be an Ubuntu image on a Fedora host.
- A process in a new network namespace has its own interfaces, its own routing table and its own port numbers. Two containers can both listen on port 80.
- Namespaces control what you can see. **Cgroups** control how much you can use. Real controllers mounted on this machine include `cpu`, `cpuacct`, `cpuset`, `memory`, `blkio`, `pids`, `devices` and `freezer`.
- Put the two together and you have a container: a normal process, with a private view from namespaces and a resource cap from cgroups. There is no "container" object in the Linux kernel at all.

■ 18.12.4 PLAIN - what is really happening inside

- Privilege rings on x86 were designed with four levels, 0 to 3. Every general purpose operating system uses exactly two: 0 for the kernel, 3 for programs.
- Rings 1 and 2 were meant for drivers and services. They were abandoned because they do not exist on other architectures, and portable kernels could not rely on them.
- Virtualization added a level below 0. On Intel it is VMX root mode, on AMD it is SVM host mode, on ARM it is EL2. The hypervisor runs there, so a guest kernel can believe it is in ring 0 while actually being supervised.
- That hardware support arrived in 2005 and 2006, with Intel VT-x and AMD-V. Before it, virtualization required rewriting guest instructions on the fly.
- Capabilities on Linux split root into about 40 separate powers, such as `CAP_NET_BIND_SERVICE` for low ports, `CAP_SYS_ADMIN` for a great deal, and `CAP_DAC_OVERRIDE` for ignoring file permissions.
- Sandboxing with `seccomp` works by giving the kernel a filter program that is run on every system call, deciding allow, deny, kill or trap. Once installed it cannot be removed, and the process cannot regain the lost ability.
- A kernel panic happens when kernel code hits a condition it cannot recover from: a null pointer dereference in kernel space, a corrupted internal structure, or an unhandled exception in an interrupt handler.
- There is no higher authority to catch the error and no isolated context to kill, so the only safe action is to stop and print what happened.
- Windows does exactly the same thing and calls it a bug check. The blue screen shows a stop code and, on modern versions, writes a crash dump to disk before restarting.

■ 18.12.5 TECHNICAL - the engineer's version

- Linux namespace types and the `clone` flags that create them:

Namespace	Flag	Isolates
Mount	<code>CLONE_NEWNS</code>	Filesystem tree

Namespace	Flag	Isolates
PID	CLONE_NEWPID	Process IDs
Network	CLONE_NEWNET	Interfaces, ports
User	CLONE_NEWUSER	UID/GID mapping
UTS	CLONE_NEWUTS	Hostname
IPC	CLONE_NEWIPC	SysV IPC, mqueues
Cgroup	CLONE_NEWCGROUP	Cgroup root view
Time	CLONE_NEWTIME	Boot and monotonic

2. Dates: the mount namespace arrived in Linux 2.4.19 in 2002, the rest through 2.6.x, user namespaces completed in 3.8 in 2013, and the time namespace in 5.6 in 2020.
3. Cgroups came from Google, written by Paul Menage and Rohit Seth, merged as “control groups” in Linux 2.6.24 in January 2008. Cgroup v2 with a unified hierarchy became the default in most distributions from around 2021.
4. Container runtimes are userspace assemblers of these primitives: Docker from 2013, then the OCI runtime specification, runc, containerd, podman, and Kubernetes as the orchestrator above them.
5. The honest version: a container shares the host kernel, so a kernel privilege escalation escapes it. A virtual machine shares only the hypervisor, which has a much smaller attack surface. Anyone who tells you containers are as isolating as VMs is selling something.
6. The middle ground is real: Firecracker, released by AWS in 2018, is a minimal virtual machine monitor of roughly 50,000 lines of Rust, with a device model of five devices, booting in about 125 milliseconds. gVisor intercepts system calls in a userspace kernel instead. Kata Containers puts a real VM under a container interface.
7. Mandatory access control layers sit above discretionary permissions: SELinux (originating from the United States National Security Agency, merged in Linux 2.6 in 2003), AppArmor, and on macOS the Sandbox subsystem plus System Integrity Protection, introduced in OS X 10.11 El Capitan in 2015.
8. Hypervisor types: type 1 runs on the hardware directly (Xen from 2003, VMware ESXi, Microsoft Hyper-V, KVM once you accept that Linux becomes the hypervisor); type 2 runs as an application (VirtualBox, VMware Workstation, older Parallels). The line is blurry and the classification is a convention, not a standard.
9. Panic and bug check diagnostics: kdump and crash on Linux, /var/crash dumps, panic=1 on the kernel command line to reboot instead of hanging (which was set on the machine used here); WinDbg with !analyze -v on Windows minidumps in C:\Windows\Minidump.
10. Common Windows stop codes worth knowing: IRQL_NOT_LESS_OR_EQUAL usually a driver touching paged memory at high interrupt level; PAGE_FAULT_IN_NONPAGED_AREA bad pointer or failing RAM; SYSTEM_SERVICE_EXCEPTION a fault in a system call path. All three most often mean a driver, not Windows itself.

■ 18.12.6 WORDS – remember these

1. Privilege ring — how much the processor will let you do — hardware protection level, 0 for kernel and 3 for user on x86-64.
2. Capability — one slice of root's power — a per-thread bit such as `CAP_NET_BIND_SERVICE`, granted independently of UID 0.
3. Secomp — a filter on which system calls you may make — a BPF program applied to every syscall, irreversible once installed.
4. Namespace — a private view of one kind of resource — the kernel mechanism giving a process its own PIDs, mounts, network or users.
5. Cgroup — a cap on how much you may use — hierarchical resource accounting and limiting for CPU, memory, I/O and process count.
6. Hypervisor — the thing that runs whole operating systems — software at a privilege level below the guest kernel, using VT-x, AMD-V or EL2.
7. Kernel panic — the trusted code failed — an unrecoverable kernel fault with no higher authority to handle it; a bug check or blue screen on Windows.

18.13 UNIX: the full story

■ 18.13.1 PLAIN – in simple words

1. Nearly every operating system you will use descends from one project, or was built to imitate it.
2. It began as a failure. In the mid-1960s MIT, General Electric and Bell Labs tried to build Multics, an ambitious time-sharing system for the GE-645.
3. Multics was late, huge and complicated. Bell Labs pulled out in 1969.
4. Two of the Bell Labs people, Ken Thompson and Dennis Ritchie, missed the pleasant working environment they had lost.
5. Thompson found a little-used PDP-7 in a corner and, in 1969, wrote a small system for it, helped by Ritchie and Rudd Canaday.
6. It was a joke on the name: Multics was big and did everything at once, so theirs was **Unics**, later spelled UNIX.
7. In 1970 the group got a PDP-11, and UNIX became a real working system.
8. In 1973 they rewrote almost the whole thing in C, a language Ritchie had made for the purpose. Almost nobody wrote operating systems in a high-level language then.
9. That single decision is why UNIX spread. Move the C compiler to a new machine and most of the operating system comes with it.

■ 18.13.2 PLAIN – a picture in your head

1. Think of a kitchen with a drawer full of single-purpose tools: a peeler, a grater, a sieve, a whisk. Each does one thing well.
2. The alternative is one huge machine with forty buttons that peels, grates, sieves and whisks, badly, and cannot be repaired.
3. UNIX chose the drawer. Small programs, each doing one job, that can be connected end to end.
4. The connection is the pipe: the output of one becomes the input of the next, without either knowing the other exists.
5. To make that work, everything must be the same shape, so the shape chosen was plain lines of text.
6. That is why `ls | grep report | wc -l` works, and why you can invent a combination nobody has ever tried and it will still work.

Where this comparison breaks: kitchen tools are physical and cannot be combined into new tools. UNIX programs can be composed into scripts that become new programs, so the drawer grows. And the text-stream convention has real costs: programs must parse each other's output, which is fragile, and this is exactly the argument PowerShell makes by passing objects instead.

■ 18.13.3 PLAIN – a worked example

1. The UNIX philosophy, as Doug McIlroy, the head of the Bell Labs computing research department and the inventor of the pipe, stated it in the 1978 Bell System Technical Journal:
2. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.
3. Expect the output of every program to become the input to another, as yet unknown, program. Do not clutter output with extraneous information.
4. Design and build software, even operating systems, to be tried early, ideally within weeks. Do not hesitate to throw away the clumsy parts and rebuild them.
5. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some out after you have finished using them.
6. Two further principles are usually added, from Rob Pike and others: everything is a file, and text is the universal interface.
7. The first is why a disk, a terminal, a random number source and a running process all appear as things you can open and read.

■ 18.13.4 PLAIN – what is really happening inside

1. AT&T could not sell UNIX. A 1956 consent decree, settling an antitrust case, forbade the Bell System from entering businesses outside common carrier communications.
2. So AT&T licensed UNIX to universities for a nominal fee, with source code. That accident is why an entire generation of computer scientists learned operating systems by reading a real one.
3. At Berkeley, students and staff extended their copy heavily: the C shell, the vi editor, virtual memory, the fast filesystem, job control, and the sockets API that the whole internet uses.
4. Their distribution was the Berkeley Software Distribution, BSD, from 1978.
5. The 1984 breakup of AT&T removed the restriction, and AT&T began selling UNIX commercially as System V. Suddenly the free university tradition and a commercial product were the same code.
6. Then came the **UNIX wars**: through the late 1980s, vendors split into rival camps, each with an incompatible UNIX, each claiming to be the standard.
7. Customers responded by demanding a written specification that any vendor could implement, which produced POSIX.
8. In 1992 AT&T's Unix System Laboratories sued Berkeley Software Design over BSD code. The case settled in February 1994, requiring a handful of files to be removed, and 4.4BSD-Lite was released free of AT&T code.
9. The delay mattered. During the two years that BSD's legal status was unclear, a student in Helsinki released a kernel with no legal cloud over it at all.

■ 18.13.5 TECHNICAL – the engineer's version

1. Key dates, all verifiable: Multics development from 1964 at MIT Project MAC with Bell Labs and General Electric; Bell Labs withdrew in 1969; Thompson's PDP-7 work began in 1969; the PDP-11/20 arrived in 1970 and the name UNIX was adopted; Version 4 was rewritten in C in 1973.
2. Ritchie and Thompson's paper *The UNIX Time-Sharing System* appeared in Communications of the ACM, volume 17 number 7, July 1974, and is the document that made the outside world take notice.

3. Thompson and Ritchie received the ACM Turing Award in 1983 for UNIX and for the theory of operating systems.
4. **POSIX** is IEEE Std 1003.1, first published in 1988. The name was suggested by Richard Stallman when the committee wanted something pronounceable. It standardizes the C API, the shell and utilities, not the kernel design.
5. The Single UNIX Specification, maintained by The Open Group, is what confers the right to use the UNIX trademark. macOS is certified UNIX. Linux is not, and has never applied. AIX, HP-UX and Solaris are.
6. The family tree, with founding dates:

System	From	Started
Research UNIX	Bell Labs	1969
BSD	UC Berkeley	1978
System V	AT&T	1983
SunOS / Solaris	Sun	1982 / 1992
AIX	IBM	1986
HP-UX	HP	1984
MINIX	Tanenbaum	1987
Linux	Torvalds	1991
FreeBSD, NetBSD	386BSD	1993
OpenBSD	NetBSD fork	1996
macOS	NeXTSTEP + BSD	2001

7. The split that matters technically: System V gave us `init` runlevels, System V IPC and the `/etc/init.d` layout; BSD gave us sockets, the fast filesystem, `rc` scripts and much of the networking code that every system, including Windows, borrowed.
8. Both branches are still running in production. FreeBSD powers Netflix's content delivery servers and is the base of the PlayStation 4 and 5 system software; OpenBSD produces OpenSSH, which almost every server on earth runs.

■ 18.13.6 WORDS - remember these

1. Multics — the ambitious project UNIX reacted against — the 1964 MIT, GE and Bell Labs time-sharing system that Bell Labs abandoned in 1969.
2. UNIX philosophy — small tools, joined together — composition of single-purpose programs over a universal text interface.
3. BSD — the Berkeley branch of UNIX — the distribution that produced sockets, the fast filesystem and today's FreeBSD, NetBSD and OpenBSD.
4. System V — the AT&T commercial branch — the source of runlevels, System V IPC and much enterprise UNIX practice.
5. POSIX — the written rules any UNIX-like system can follow — IEEE Std 1003.1, first published in 1988, defining API, shell and utilities.
6. Single UNIX Specification — what lets you call it UNIX — The Open Group's certification, held by macOS, AIX, HP-UX and Solaris, not by Linux.

18.14 Linux

■ 18.14.1 PLAIN – in simple words

1. In 1991 a 21-year-old student in Helsinki, Linus Torvalds, wanted a UNIX-like system for his new 386 PC. The teaching system MINIX was restricted and he could not change it freely.
2. On 25 August 1991 he posted to the newsgroup comp.os.minix. The message is worth quoting exactly, because it is famous for being wrong:

```
Hello everybody out there using minix -
```

```
I'm doing a (free) operating system (just a hobby, won't be  
big and professional like gnu) for 386(486) AT clones.
```

3. The post continued that he had ported bash 1.08 and gcc 1.40, asked what features people wanted, and added a postscript: the system was free of any MINIX code, had a multi-threaded filesystem, and was “NOT portable” because it used 386 task switching.
4. Version 0.01 was released on 17 September 1991.
5. He wrote only the kernel. Everything else that made it usable — the shell, the compiler, the text tools, the C library — already existed, from the GNU project.
6. The GNU project had been started by Richard Stallman in 1983 to build a complete free UNIX-like system. By 1991 GNU had almost everything except a working kernel.
7. So the two halves fitted together exactly. That is not luck; it is what happens when both sides target the same POSIX interface.
8. In January 1992, with version 0.12, Torvalds put Linux under the GNU General Public License. He has repeatedly called this the single best decision he made.

■ 18.14.2 PLAIN – a picture in your head

1. Think of an engine and a car.
2. The Linux kernel is the engine. On its own it moves nothing and has no seats.
3. GNU supplied the gearbox, the wheels, the steering and the dashboard.
4. A **distribution** is the finished car: someone chose an engine version, bolted on a particular set of parts, painted it, tested it and put a badge on it.
5. Ubuntu, Debian, Fedora, Arch and Android are all cars built around the same family of engines, and they do not look or drive alike.
6. This is why “which is better, Linux or Ubuntu” is a confused question. One is an engine, the other is a car containing it.

Where this comparison breaks: a car engine is useless outside a car, whereas the Linux kernel really does run alone in embedded systems with a single program on top. And a car maker builds one car; a distribution assembles tens of thousands of independently developed pieces it did not write and cannot fully test.

■ 18.14.3 PLAIN – a worked example

1. What a distribution actually adds to the kernel, concretely:
2. A C library, usually glibc, sometimes musl. The kernel does not provide one.
3. A userland: GNU coreutils, or BusyBox on small systems, or Toybox on Android.
4. An init system and service manager, usually systemd since about 2015.
5. A package manager and a repository of tested, signed, compiled packages: apt with .deb, dnf with .rpm, pacman, apk.
6. A default configuration for thousands of programs, which is most of the real work and almost none of the credit.

7. A desktop environment, if any: GNOME, KDE Plasma, Xfce.
8. A release policy: Debian stable freezes and supports for years; Arch ships updates continuously; Ubuntu LTS releases every two years in April with five years of support.
9. Security updates, which is the service people actually pay for.
10. Numbers for scale: Debian 12, released June 2023, contains over 64,000 binary packages. The kernel is one of them.

■ 18.14.4 PLAIN – what is really happening inside

1. How a change gets into Linux, in order:
2. You write a patch and send it as plain text email to the mailing list for the relevant subsystem, with a Signed-off-by line certifying its origin.
3. The subsystem maintainer reviews it. Most patches are rejected or sent back several times. Review is public and often blunt.
4. If accepted, it goes into the maintainer's tree, then into linux-next, where it is built and tested against everything else nightly.
5. When the merge window opens, the maintainer sends a pull request to Torvalds, who merges it.
6. The merge window is two weeks. Then come seven or eight weekly release candidates, and then the release. The whole cycle is nine to ten weeks and has been remarkably regular since 2005.
7. Torvalds wrote Git in 2005 in about ten days, specifically because this workflow needed a tool that did not exist. The tool exists because of the process, not the other way round.
8. Version numbers carry no meaning beyond ordering. Torvalds increments the first digit when the second gets "too big", by his own description. That is why 2.6.39 became 3.0, 3.19 became 4.0, and 6.17 became 7.0 in April 2026.
9. Long-term support kernels are chosen once a year and maintained for two to six years, or longer under the Civil Infrastructure Platform.

■ 18.14.5 TECHNICAL – the engineer's version

1. Verified version history and current state, from kernel.org as of 9 August 2026: mainline 7.2-rc7, stable 7.1.8, longterm 6.18.44, 6.12.103, 6.6.151, 6.1.182, 5.15.215 and 5.10.264.

Version	Released	Note
0.01	17 Sep 1991	First release
0.12	Jan 1992	Relicensed to GPL
1.0	14 Mar 1994	First stable
2.6.0	17 Dec 2003	O(1) scheduler era
6.1	11 Dec 2022	Rust support, LTS
6.6	29 Oct 2023	EEVDF replaces CFS
6.12	17 Nov 2024	sched_ext, LTS
7.0	12 Apr 2026	Numbering rollover

2. Licensing: the kernel is GPL version 2 only, not "version 2 or later". That deliberate choice means it can never be relicensed to GPLv3 without the agreement of thousands of copyright holders.
3. The user-space boundary is covered by a syscall exception note, which is why proprietary applications on Linux are legal while proprietary in-kernel drivers are legally contested.
4. The GNU/Linux naming argument, stated fairly. The Free Software Foundation's position: the system is mostly GNU with a Linux kernel, so calling the whole thing Linux erases the project that supplied most of it and the freedom philosophy that motivated it. The common counter-position: the name refers to the kernel and to common usage, most modern systems contain far

less GNU code than in 1993, and Android contains none. Both sides are factually correct about different things; the disagreement is about naming, not about facts.

5. Where Linux runs, with real figures. Supercomputers: on the June 2026 TOP500 list all 500 systems run a Linux-based operating system, as they have since November 2017. The list is led by LineShine at the National Supercomputing Centre in Shenzhen at 2,198.40 PFlop/s, ahead of El Capitan at Lawrence Livermore at 1,809.00 PFlop/s and Frontier at Oak Ridge at 1,353.00 PFlop/s.
6. Mobile: Android, which is a Linux kernel with a completely different userland, accounts for roughly 70 percent of smartphone operating system share worldwide, with iOS most of the rest. Share figures move and vary by source, so treat the exact number as approximate.
7. Servers and cloud: Linux is the majority operating system on public cloud instances and on web-facing servers, by a wide margin in every survey; exact percentages vary by methodology and are worth checking before quoting.
8. Embedded: routers, televisions, cars, cameras, industrial controllers, spacecraft. The Mars helicopter Ingenuity ran Linux on a Snapdragon 801 with the F Prime framework, the first Linux flight on another planet, in 2021.
9. Desktop: the smallest share, in the low single digits of percent, and rising slowly. Steam's hardware survey and Statcounter both place it around 2 to 5 percent in 2025 and 2026 depending on method.
10. Contribution scale: roughly 1,500 to 2,000 developers from over 200 companies contribute to each release. The largest corporate contributors in recent years include Intel, Red Hat, Google, AMD, Linaro and Huawei. The image of Linux as hobbyist code has been wrong for over fifteen years.

■ 18.14.6 WORDS – remember these

1. Kernel versus distribution — the engine versus the finished car — the Linux kernel is one component; a distribution assembles it with libc, userland, init, packaging and configuration.
2. GPL — the licence that requires sharing changes — the GNU General Public License version 2 for the kernel, requiring derived works to be distributed under the same terms with source.
3. Merge window — the two weeks when new features are accepted — the first phase of each nine to ten week Linux release cycle.
4. LTS kernel — the version kept alive for years — a long-term support release chosen annually and maintained with backported fixes.
5. Signed-off-by — the line certifying where a patch came from — the Developer Certificate of Origin attestation required on every kernel patch.
6. Userland — everything above the kernel — the libraries, shells and utilities that make a kernel into a usable system.

18.15 Windows

■ 18.15.1 PLAIN – in simple words

1. Windows has two completely separate ancestries that were merged only in 2001.
2. The first is **MS-DOS**. Tim Paterson at Seattle Computer Products wrote 86-DOS in about six weeks in 1980, as a port of the ideas in Digital Research's CP/M to the 8086 processor.
3. Microsoft licensed it, hired Paterson in May 1981, bought the rights that July for \$25,000, and licensed it to IBM, which shipped it as PC DOS 1.0 in August 1981.
4. MS-DOS had no protection, no multitasking and no memory management. One program ran and owned the machine.
5. Windows 1.0, in November 1985, was not an operating system. It was a program you ran from DOS that drew overlapping panels and let you switch between applications. Windows 2.0, 3.0 in 1990 and 3.1 in 1992 were the same idea, much improved.
6. The second ancestry is **Windows NT**, started from nothing in 1988 by a team under Dave Cutler, whom Microsoft had recruited from Digital Equipment Corporation.
7. Cutler had designed the RSX-11M and VMS operating systems at Digital. NT is full of VMS ideas, and the resemblance was noticed immediately.
8. Windows NT 3.1 shipped on 27 July 1993. It had protection, preemptive multitasking, proper security and portability across processors.
9. For eight years Microsoft sold two families: the NT line for business, and Windows 95, 98 and Me for home users, which were still DOS underneath.
10. Windows XP in 2001 put everyone on the NT kernel. Every Windows since is NT.

■ 18.15.2 PLAIN – a picture in your head

1. Think of a town with an old wooden high street and, on the edge, a new engineered town centre built to modern codes.
2. The wooden high street is the DOS line. Everything is convenient, nothing is fireproof, and one careless shop burns the row down.
3. The new centre is NT: fire doors, separate services, inspections. Slower to build, much heavier, and it does not burn down.
4. For years people kept shopping on the wooden street because that is where all the shops they liked were.
5. Eventually the new centre was made to look exactly like the old street, so nobody had to change their habits, and the wooden buildings were removed.
6. That disguise is the Win32 API, and it is the real reason the migration worked.

Where this comparison breaks: the disguise is not cosmetic. Win32 is a genuine programming interface with defined semantics, and NT implements it as one of several personalities over a lower interface. And the wooden street was never fully removed: enough compatibility behaviour survives in Windows 11 that programs from 1995 often still run.

■ 18.15.3 PLAIN – a worked example

1. The **registry** is the part of Windows most often described wrongly, so here it is properly.
2. It is a hierarchical database of settings, stored in a small number of binary files called hives, and exposed as a tree of keys and values.
3. Before it, settings lived in hundreds of .ini text files scattered everywhere. There was no way to set permissions on them, no transactions, and no way to find them all.
4. The registry gave one namespace, per-value access control, and a defined API. That was a real engineering improvement, not a bureaucratic one.
5. The visible top-level keys and what they actually are:

Key	What it really is
HKEY_LOCAL_MACHINE	Machine-wide settings
HKEY_CURRENT_USER	A link into HKEY_USERS
HKEY_USERS	Every loaded user hive
HKEY_CLASSES_ROOT	Merge of machine and user
HKEY_CURRENT_CONFIG	A link to a hardware key

6. Only HKEY_LOCAL_MACHINE and HKEY_USERS hold real data. The others are views and links, which is why the same value appears in two places.
7. On disk the hives are C:\Windows\System32\config\SYSTEM, SOFTWARE, SAM and SECURITY, plus NTUSER.DAT in each user's profile.
8. Value types include REG_SZ (text), REG_DWORD (32-bit number), REG_BINARY and REG_MULTI_SZ.
9. Registry cleaners are, with rare exceptions, useless. A registry with 200,000 unused keys costs a modern machine no measurable time, because lookups are indexed. This is settled, not controversial.

■ 18.15.4 PLAIN – what is really happening inside

1. The NT kernel is layered, and the layers have names worth knowing.
2. The **HAL**, hardware abstraction layer, is hal.dll. It hides differences between motherboards: interrupt controllers, timers, bus access. Above it, the rest of the kernel sees one abstract machine.
3. The **kernel** proper, sometimes called the microkernel, handles scheduling, interrupts, synchronization primitives and multiprocessor coordination. It is small and never paged out.
4. The **executive** sits above it and contains the real subsystems: the object manager, process manager, memory manager, I/O manager, cache manager, security reference monitor and configuration manager. All of these live in ntoskrnl.exe.
5. The **object manager** is NT's most distinctive idea. Processes, threads, files, mutexes, registry keys and devices are all objects in one namespace with one set of rules for naming, referencing and securing.
6. Above the executive, in user mode, are **subsystems**: personalities that present a particular API. Win32 was one, and OS/2 and POSIX subsystems existed early on and were later removed.
7. That structure is why the Windows Subsystem for Linux was possible. WSL 1, in 2016, was a new subsystem translating Linux system calls to NT ones. WSL 2, from 2019, gave up and shipped a real Linux kernel in a lightweight Hyper-V virtual machine instead, because full syscall compatibility proved harder than virtualization.

■ 18.15.5 TECHNICAL – the engineer’s version

1. NT 3.1 shipped for four architectures in 1993: Intel x86, DEC Alpha, MIPS and later PowerPC. That portability was designed in from the start, in contrast to early Linux.
2. NT’s design lineage from VMS is direct: asynchronous I/O request packets, the interrupt request level scheme, the object and handle model, and the layered driver stack all have VMS counterparts.
3. The version numbering is a source of confusion. NT 3.1, 3.5, 3.51, 4.0, then 5.0 was Windows 2000, 5.1 was XP, 6.0 Vista, 6.1 Windows 7, 6.2 Windows 8, 6.3 Windows 8.1, then 10.0 for both Windows 10 and Windows 11.
4. Current state as of August 2026: Windows 11 version 26H1, build 10.0.28000, released 10 February 2026, a platform release focused on ARM devices including Snapdragon X2. Version 25H2, build 26200, shipped 30 September 2025.
5. **NTFS**, shipped with NT 3.1 in 1993, is a journaling filesystem built around a Master File Table in which every file, including the MFT itself, is a record with a set of attributes. Small files live entirely inside their MFT record. It supports hard links, alternate data streams, compression, encryption, quotas and sparse files.
6. Alternate data streams are worth knowing about: `file.txt:hidden` is a second stream on the same file, invisible to `dir`, and historically a malware hiding place.
7. **Win32** is the documented API, in `kernel32.dll`, `user32.dll` and `gdi32.dll`. It calls the undocumented native API in `ntdll.dll`. Microsoft’s compatibility promise attaches to Win32, not to the native layer.
8. Driver signing timeline: 64-bit editions have required signed kernel-mode drivers since Vista in 2006; since Windows 10 version 1607 in 2016, new kernel drivers must be submitted to and signed by the Microsoft hardware developer portal. Attestation signing covers most, EV certificates are required to submit.
9. Modern additions: virtualization-based security uses Hyper-V to isolate credentials in a separate virtual trust level; Windows 11 requires TPM 2.0 and UEFI Secure Boot; the July 2024 CrowdStrike incident, in which a faulty configuration update to a kernel-mode security driver bug-checked roughly 8.5 million Windows machines worldwide, restarted the industry argument about how much security software belongs in the kernel at all.
10. Tools: Sysinternals suite (Process Explorer, Process Monitor, Autoruns, WinObj, Handle), WinDbg with the public symbol server, Performance Monitor, Event Viewer, `wevtutil`, and PowerShell’s `Get-Process` and `Get-Service`.

■ 18.15.6 WORDS – remember these

1. MS-DOS — the single-tasking ancestor — the 16-bit real-mode system bought from Seattle Computer Products in 1981 and shipped as PC DOS 1.0.
2. Windows NT — the engineered line that survived — Dave Cutler’s 1993 kernel with protection, portability and preemptive multitasking.
3. HAL — the layer that hides the motherboard — `hal.dll`, abstracting interrupt controllers, timers and bus access.
4. Executive — the main body of kernel services — object, process, memory, I/O, cache, security and configuration managers inside `ntoskrnl.exe`.
5. Subsystem — a personality presenting one API — user-mode environment such as Win32, and formerly OS/2 and POSIX.
6. Registry — one database instead of thousands of ini files — hierarchical configuration store in binary hives with per-value access control.
7. NTFS — the Windows filesystem since 1993 — journaling filesystem organized around a Master

File Table of attributed records.

18.16 macOS, iOS, Android and the rest

■ 18.16.1 PLAIN – in simple words

1. Modern Apple systems come from a company Apple did not own in 1996.
2. Steve Jobs left Apple in 1985 and founded NeXT, which built expensive workstations and an operating system called NeXTSTEP.
3. NeXTSTEP combined the Mach kernel from Carnegie Mellon with BSD UNIX code and a set of programming frameworks written in Objective-C.
4. Apple, meanwhile, had spent most of the 1990s failing to replace the ageing classic Mac OS, which had no memory protection and cooperative multitasking.
5. Apple announced it would buy NeXT on 20 December 1996, and closed the deal on 7 February 1997 for about \$427 million. Jobs returned with it.
6. NeXTSTEP became the base of Mac OS X, released for desktops on 24 March 2001.
7. The same foundation, with a different interface layer, is iOS, iPadOS, watchOS, tvOS and visionOS. One core, many products.
8. Android took a different route entirely: the Linux kernel, with none of the GNU userland, and an application layer designed from scratch by Google.

■ 18.16.2 PLAIN – a picture in your head

1. Think of two ways of putting a new engine into an old car brand.
2. Apple bought a whole different car, rebadged it, and gradually made it look like the old one. The dashboard was familiar, everything under it was new.
3. Google took a well-known engine, threw away every other part of the donor car, and built a completely new vehicle around it.
4. So an Android phone runs Linux, and almost nothing you know about a Linux desktop applies to it: different C library, different init, different graphics, different IPC, different package format.
5. Calling Android “Linux” is true about the engine and misleading about the car.

Where this comparison breaks: Apple did not simply rebadge. XNU is a genuine hybrid whose Mach layer and BSD layer are welded together in ways neither original design intended, and it has been rewritten heavily over 25 years.

■ 18.16.3 PLAIN – a worked example

1. **XNU** is the Apple kernel. The name stands for “X is Not Unix”, a joke from the NeXT era, and it is made of three parts.
2. Part one: **Mach**, from Carnegie Mellon University, originally a microkernel research project by Rick Rashid and Avie Tevanian. Apple uses a heavily modified OSF MK 7.3. It provides virtual memory, tasks, threads, scheduling and Mach ports for messaging.
3. Part two: **BSD**, derived from 4.3BSD and refreshed from FreeBSD. It provides the POSIX API, the process model, users and permissions, the network stack and the VFS.
4. Part three: **IOKit**, the driver framework, written in a restricted subset of Embedded C++ with no exceptions, no templates and no multiple inheritance.
5. All three run in one address space in kernel mode. Mach’s separation is a structure, not an isolation boundary. That is why XNU is called hybrid.
6. **Darwin** is the open-source part: XNU plus the BSD userland plus the core libraries. Apple publishes it. The frameworks, the interface and the applications on top are not open.

7. Above Darwin sit the frameworks: Cocoa and AppKit on macOS, UIKit on iOS, written in Objective-C, and SwiftUI, from 2019, written in Swift.
8. Objective-C came from NeXT. Swift was announced in June 2014 and is now the default for new Apple development, with Objective-C still fully supported.

■ 18.16.4 PLAIN – what is really happening inside

1. **System Integrity Protection**, introduced in OS X 10.11 El Capitan in 2015, is a rule enforced by the kernel that even root may not modify protected system locations, load unsigned kernel extensions or attach a debugger to Apple binaries.
2. It is configured from the recovery environment, not from the running system, which is the point: a compromised running system cannot turn it off.
3. Since macOS 10.15 Catalina in 2019, the system volume is a separate read-only volume, and since Big Sur in 2020 it is a cryptographically sealed snapshot.
4. The **Apple silicon transition** was announced at WWDC on 22 June 2020. The first M1 Macs shipped in November 2020, and the transition finished with the Mac Pro in June 2023.
5. Two translation layers made it survivable: Rosetta 2, which translates x86-64 binaries ahead of time on install, and universal binaries containing both architectures in one file.
6. **Android** is a Linux kernel plus an entirely separate userland.
7. Its C library is Bionic, written by Google, not glibc. Its init is Android's own. Its IPC is **Binder**, a kernel driver originally from Be Incorporated's OpenBinder, doing object-oriented calls between processes.
8. Its application runtime is **ART**, the Android Runtime, introduced as a preview in Android 4.4 KitKat in 2013 and made the only runtime in Android 5.0 Lollipop in 2014, replacing the older Dalvik virtual machine.
9. ART started as purely ahead-of-time compilation. Since Android 7.0 Nougat it is a hybrid: new apps run interpreted and JIT-compiled while a profile is collected, and the hot code is compiled ahead of time when the device is idle and charging.
10. Between the kernel and the framework sits Android's **HAL**, a set of defined interfaces every vendor must implement, so that Android can be ported without Google seeing the vendor's driver source.
11. Project Treble, from Android 8.0 in 2017, split the vendor implementation from the framework so the framework can be updated independently. It was a response to the update problem, and it helped without solving it.

■ 18.16.5 TECHNICAL – the engineer's version

1. Current versions as of August 2026: macOS 26 Tahoe, released 15 September 2025, introducing the Liquid Glass design across Apple platforms.

System	Kernel	Userland
macOS	XNU (Mach+BSD)	BSD + Apple
iOS, iPadOS	XNU	BSD + Apple
Android	Linux	Bionic, Toybox, ART
ChromeOS	Linux	Chrome, Ash, crosvm
FreeBSD	FreeBSD	BSD
QNX	QNX microkernel	POSIX

2. Apple filesystem history: HFS from 1985, HFS+ from 1998, and APFS announced in 2016, shipped on iOS 10.3 in March 2017 and macOS 10.13 High Sierra in September 2017. APFS is copy-on-write with cheap snapshots, native encryption and space sharing across volumes in a container.
3. Security hardware: the Secure Enclave, a separate co-processor with its own boot ROM, present since the A7 in 2013, holding keys and biometric templates that the main processor never sees.
4. On Apple silicon, memory pages are 16 KiB rather than 4 KiB, which changes memory footprint measurements noticeably compared with x86 Macs.
5. Android's kernel is a fork with substantial out-of-tree patches, though the Generic Kernel Image programme since Android 11 has pushed vendors towards a common core. Android security patches are shipped monthly on a published bulletin schedule.
6. ChromeOS is Gentoo-derived Linux with a hardened design: verified boot with a read-only rootfs, automatic background updates to an alternate partition, and Linux applications run inside a virtual machine using crosvm and Termina. Announced 7 July 2009, first Chromebooks June 2011.
7. **Real-time operating systems** are a different category with a different promise: not speed, but a bounded worst-case response time. Names worth knowing: VxWorks (which ran the Mars rovers Spirit, Opportunity, Curiosity and Perseverance), QNX Neutrino (a true microkernel, dominant in automotive infotainment), FreeRTOS, Zephyr and RTEMS.
8. The distinction that matters: hard real-time guarantees a deadline and treats a miss as a failure; soft real-time treats a miss as degraded quality. Linux with the PREEMPT_RT patch set, merged into mainline in Linux 6.12 in November 2024 after roughly twenty years of development, offers strong soft real-time and, with care, hard real-time on suitable hardware.

■ 18.16.6 WORDS - remember these

1. XNU — Apple's kernel — a hybrid of Mach for memory and messaging, BSD for POSIX and networking, and IOKit for drivers.
2. Darwin — the open part of macOS — XNU plus BSD userland and core libraries, published by Apple without the frameworks or interface.
3. Mach port — Apple's basic IPC handle — a capability referring to a message queue, underlying most macOS inter-process communication.
4. SIP — root is not allowed either — System Integrity Protection, kernel enforced since OS X 10.11 in 2015, configurable only from recovery.
5. Binder — Android's IPC — a kernel driver providing object-oriented calls between processes, derived from OpenBinder.
6. ART — Android's application runtime — profile-guided hybrid of interpretation, JIT and ahead-of-time compilation, replacing Dalvik in Android 5.0.
7. RTOS — an operating system that promises deadlines — bounded worst-case latency rather than best average throughput.

18.17 How you would actually write one

■ 18.17.1 PLAIN - in simple words

1. Writing an operating system is not mysterious. It is a long series of small, well-documented steps, each of which is achievable in an evening.
2. The reason few people finish is not difficulty. It is that there are hundreds of steps and no user to please until quite far in.
3. The minimum thing that deserves the name is small: it must start on its own, take control of the processor, put something on the screen, and respond to a key press.

4. That is achievable in a weekend, in about 300 lines, and it is a real operating system in the same sense that a paper aeroplane is a real aircraft.
5. After that the work has a natural order, because each part needs the one before it.
6. You need interrupts before you can have a keyboard. You need a memory allocator before you can have processes. You need processes before a scheduler. You need a disk driver before a filesystem. You need a filesystem before a shell.
7. You do this on a simulated machine, not real hardware, because a simulated machine restarts in a second and lets you inspect every register.
8. Everything you need is documented, free, and has been done by thousands of people who wrote down what went wrong.

■ 18.17.2 PLAIN – a picture in your head

1. Think of building a house on empty ground where nothing exists, not even roads.
2. First you need a track wide enough to bring in a digger. That is the bootloader: just enough to get the next thing in.
3. Then foundations: level ground, marked out, load bearing. That is switching the processor into its proper mode and setting up the tables it needs.
4. Then the wiring loom, before any room is finished, because everything depends on it. That is the interrupt table.
5. Then plumbing: a system for allocating what is scarce. That is the memory manager.
6. Only then do rooms appear, and people move between them. That is processes and the scheduler.
7. The kitchen and the front door come last, and they are the only parts a visitor ever notices. That is the filesystem and the shell.

Where this comparison breaks: a builder can inspect a half-built house by walking through it. When your kernel fails, the machine simply stops, with no message, because the code that would print a message is the code that broke. Building a way to see inside your own system is a real early task, not a luxury.

■ 18.17.3 PLAIN – a worked example

1. The smallest complete thing: a boot sector that prints a character. This is real, assembles with NASM, and runs in QEMU.

```
bits 16
org 0x7c00

start:
    mov ah, 0x0e          ; BIOS teletype function
    mov al, 'K'
    int 0x10              ; call BIOS video service
    jmp $                  ; loop here forever

    times 510-($-$$) db 0
    dw 0xaa55              ; boot signature
```

2. Line by line. `bits 16` says generate 16-bit code, because the processor starts in real mode.
3. `org 0x7c00` tells the assembler that this code will live at address 0x7C00, which is where BIOS loads a boot sector. That is fixed, not a choice.
4. `mov ah, 0x0e` selects the BIOS teletype output service, `mov al, 'K'` is the character, and `int 0x10` calls the BIOS video interrupt.
5. `jmp $` jumps to itself forever, because there is nothing to return to.

6. `times 510-($-$$) db 0` pads with zeros up to byte 510.
7. `dw 0xaa55` writes the two-byte signature that BIOS checks before it will treat the sector as bootable.
8. Build and run it with two commands:

```
nasm -f bin boot.asm -o boot.bin
qemu-system-i386 -drive format=raw,file=boot.bin
```

9. A window opens and shows the letter K. Nothing else on that machine exists. No operating system, no library, no memory manager, no C runtime. You wrote every instruction that ran.
10. That is 512 bytes and it is the honest starting point of every hobby OS.

■ 18.17.4 PLAIN – what is really happening inside

1. The realistic order of work, with what each stage costs an average learner working evenings.
2. **Boot and print** — one weekend. The example above.
3. **Protected mode or long mode** — one to two weeks. Build a global descriptor table, set a bit in a control register, and jump. Getting the jump right is the classic first wall.
4. **A C environment** — one week. Write a linker script, a small startup stub, and build with a cross-compiler so you are not accidentally linking against your host's C library.
5. **Interrupts** — two to three weeks. An interrupt descriptor table, entry stubs that save registers, remapping the interrupt controller, a timer and a keyboard.
6. **Physical and virtual memory** — one month. A page frame allocator, page tables, and a heap. This is where most projects stall, because bugs here produce silent corruption rather than error messages.
7. **Processes and a scheduler** — one month. Save and restore context, a task list, round robin. The first successful switch between two tasks is the most satisfying moment in the project.
8. **User mode** — two to four weeks. Separate address spaces, a system call entry, and a separate stack per privilege level.
9. **A disk driver and a filesystem** — one to two months. ATA PIO is simplest. FAT16 is the easiest real filesystem to read, and writing is much harder than reading.
10. **A shell** — two weeks, and it feels like finishing, because for the first time you can type at your own system and it answers.
11. Total, for one person working evenings: roughly six months to a year to reach a usable shell. Two weekends to reach something that boots and responds.

■ 18.17.5 TECHNICAL – the engineer's version

1. Tooling, all free. A cross-compiler built as `i686-elf-gcc` or `x86_64-elf-gcc`, so headers and libraries from your host cannot leak in. NASM or GAS for assembly. GNU ld with a custom linker script. GNU make.
2. Emulators: QEMU for speed and GDB support, Bochs for its built-in debugger which can single-step at the hardware level, and VirtualBox for a final check against something closer to real firmware.
3. Debugging technique that changes everything: run QEMU with `-s -S` and attach GDB to port 1234. You can then single-step your kernel from the very first instruction, with symbols.
4. Bootloader choices: write your own for learning, use GRUB with the Multiboot2 specification to skip straight to a 32-bit or 64-bit C environment, or use Limine for a modern UEFI-capable option.
5. Learning resources, by name.

Resource	Kind	Focus
OSDev Wiki	Reference	Everything, x86 detail
Three Easy Pieces	Textbook	Free, thorough
xv6, MIT 6.1810	Teaching OS	Readable UNIX in C
Writing an OS in Rust	Tutorial	Modern, step by step
The little book about	Short book	Minimal x86 kernel
MINIX 3	Microkernel	Tanenbaum's system
SerenityOS	Live project	Desktop, from nothing
Redox OS	Live project	Microkernel in Rust

- Notes on those. The OSDev wiki is the single most useful page collection in this field, and its “Beginner Mistakes” page will save you weeks. *Operating Systems: Three Easy Pieces*, by Remzi and Andrea Arpaci-Dusseau at Wisconsin, is free online and is the best modern textbook. *The little book about OS development*, by Erik Helin and Adam Renberg, is the shortest complete path from nothing to a small x86 kernel. xv6 is MIT’s teaching reimplement of Sixth Edition UNIX in ANSI C for RISC-V, with a line-by-line commentary book. Philipp Oppermann’s *Writing an OS in Rust* is the best written tutorial series of the last decade. SerenityOS was started by Andreas Kling in 2018 and is now a complete graphical UNIX-like system written from nothing including its own browser engine.
- Specifications you will actually read: the Intel Software Developer’s Manual volume 3 for system programming, the AMD64 Architecture Programmer’s Manual volume 2, the Multiboot2 specification, the UEFI specification, the ATA and NVMe specifications, and the OSDev wiki’s summaries of all of them.
- The honest scope warning. A hobby kernel that boots, schedules and runs a shell is perhaps 5,000 to 20,000 lines. A system you could daily-drive needs drivers for real hardware, a network stack, a graphics stack, a browser and a toolchain, and that is where the number reaches millions. SerenityOS took a full-time effort and many contributors over years to reach a usable desktop.
- Why do it anyway: after building a scheduler you will never again be confused about what a context switch is, and after building a page allocator you will never again be confused about virtual memory. The understanding is disproportionate to the code.

■ 18.17.6 WORDS – remember these

- Cross-compiler — a compiler that targets a different system — a toolchain such as `i686-elf-gcc` that cannot accidentally use the host’s libraries.
- Boot sector — the 512 bytes BIOS will run — loaded at `0x7C00` and required to end with the signature `0xAA55`.
- Global descriptor table — the table that describes memory regions — the x86 structure that must exist before entering protected mode.
- Interrupt descriptor table — the list of handler addresses — the table mapping interrupt and exception numbers to entry points.
- Multiboot2 — a standard way for a bootloader to hand over — the specification GRUB implements, delivering a kernel into a known state with a memory map.
- Linker script — the file that decides where code lands — the `ld` input that defines sections and load addresses for a freestanding binary.

18.98 Common wrong ideas

1. Wrong: the operating system is the desktop, the icons and the windows. Right: those are ordinary programs. The operating system is the kernel plus the services underneath, and a machine with no graphics at all still has a complete operating system.
2. Wrong: Linux is an operating system. Right: Linux is a kernel. What you install is a distribution, which adds a C library, a userland, an init system and tens of thousands of packages. Android proves the point: same kernel, none of the rest.
3. Wrong: a process and a program are the same thing. Right: a program is a file that does nothing. A process is that program while running, with its own memory, descriptors and scheduling state. One program can be many processes.
4. Wrong: multitasking means the processor runs several programs at once. Right: on one core it runs one thing at a time and switches quickly. Real simultaneity requires several cores. On this machine the kernel had performed 4,153,017 context switches in 95 minutes to produce the illusion.
5. Wrong: threads are always much cheaper than processes. Right: measured here, creating a thread cost 32.8 microseconds against 182.5 for a fork, about 5.6 times. But forking after dirtying 256 MB cost 2,620 microseconds, because page tables must still be copied. The ratio depends on what the process holds.
6. Wrong: `malloc` gives you memory. Right: it gives you addresses. Physical memory is allocated one 4 KiB page at a time, on the first write to each page, through a page fault. Asking for a gigabyte moves VSZ and leaves RSS unchanged.
7. Wrong: the out-of-memory killer means your machine ran out of RAM through bad management. Right: it means the kernel's earlier promises could not all be kept at once. Overcommit is deliberate, because programs reserve far more than they touch.
8. Wrong: a system call is just a function call into a library. Right: it is a hardware mode transition. The `syscall` instruction changes privilege level and jumps to an address only the kernel could set. It cost 89 nanoseconds here against one or two for a normal call.
9. Wrong: `write` returning success means the data is on the disk. Right: it means the data is in the page cache and marked dirty. Only `fsync`, `fdatasync` or `O_DIRECT` involve the storage device. Databases care about this difference more than anything else.
10. Wrong: containers are lightweight virtual machines. Right: containers are ordinary processes with a restricted view, built from namespaces and cgroups, sharing the host kernel. A kernel vulnerability escapes a container and does not escape a virtual machine.
11. Wrong: a blue screen or kernel panic means the operating system is badly written. Right: it usually means a driver failed. Drivers run with full privilege, there is no higher authority to catch their errors, and stopping is safer than continuing with corrupted state.
12. Wrong: `kill -9` is the proper way to stop a program. Right: `SIGKILL` cannot be caught, so the program cannot flush buffers, release locks or write its state. `SIGTERM` first, wait, then `SIGKILL`. A process stuck in D state ignores both.

18.99 Chapter summary in 20 lines

1. An operating system exists to do three things: share the hardware, keep programs apart, and hide devices behind a simple interface.
2. All of its power comes from two hardware features: a privileged processor mode, and a timer interrupt that forces control back to the kernel.

3. The kernel is the privileged part. It is entered only three ways: a system call, an interrupt, or a fault. Between those it does nothing.
4. Kernel designs run from monolithic (Linux) through hybrid (Windows NT, XNU) to microkernel (MINIX 3, QNX, seL4); the 1992 Tanenbaum-Torvalds debate settled nothing and both approaches ship today.
5. Kernels are written in C for predictable code with no hidden runtime, plus assembly for entry paths, and since Linux 6.1 in December 2022, Rust.
6. Booting is a chain: power good, reset vector at 0xFFFFFFF0, firmware and POST, boot device, bootloader, kernel, initramfs, root filesystem, PID 1.
7. On the machine used here the kernel phase took 2.55 seconds, and PID 1 started at 1.17 seconds, before the real root was mounted at 2.55.
8. A process is a running program with its own address space, descriptors and state; UNIX creates one with fork then exec, Windows with CreateProcess.
9. Threads share everything except stack and registers, which makes them cheap and dangerous; the kernel schedules them individually in the 1:1 model.
10. The scheduler answers one question thousands of times a second, and every algorithm trades throughput against responsiveness; round robin loses on averages and wins on how the machine feels.
11. Linux used CFS from 2.6.23 in 2007 and replaced it with EEVDF in 6.6 in October 2023; Windows uses 32 priority levels with dynamic boosts.
12. A system call puts a number in rax, arguments in six registers, executes syscall, and lands where MSR_LSTAR points; printing “hello” took about twenty-five of them, of which one was the point.
13. The kernel allocates physical page frames lazily on first touch, overcommits deliberately, and kills the highest-scoring process when the promise fails.
14. The virtual filesystem layer dispatches through per-filesystem function tables, which is why /proc can be read like a file that does not exist.
15. Three tables sit behind every open file: the per-process descriptor table, the system-wide open file table, and the inode table.
16. UNIX permissions are owner, group, other with read, write, execute, plus setuid, setgid and sticky; Windows uses ordered access control lists.
17. Pipes, FIFOs, signals, message queues, shared memory, sockets and RPC differ in copies, message boundaries and whether they cross machines.
18. Containers are namespaces plus cgroups on a shared kernel; virtual machines add a hypervisor below the guest kernel and isolate far more strongly.
19. The lineage is one family: Multics failed, UNIX began on a PDP-7 in 1969, was rewritten in C in 1973, split into BSD and System V, was standardized as POSIX in 1988, and produced Linux in 1991, macOS via NeXT in 2001, and Android on a Linux kernel in 2008.
20. Every one of the 500 machines on the June 2026 TOP500 list runs Linux, as has been true since November 2017, while Windows NT’s 1993 design still runs most desktops and XNU runs every Apple device.

The Terminal and the Shell

19.0 What this chapter gives you

1. You will be able to say what a terminal actually was, as a physical machine, and why the word survived after the machine died.
2. You will be able to tell apart five things that people constantly mix up: terminal, terminal emulator, shell, console, and command line.
3. You will be able to explain what a pseudo-terminal is, and describe exactly what happens inside the machine when you press Ctrl-C.
4. You will be able to describe the shell's read-parse-expand-execute loop, and name the two system calls that start every program you run.
5. You will be able to use redirection and pipes correctly, including the case where `2>&1` in the wrong place does nothing useful.
6. You will be able to read an exit code, chain commands with `&&` and `||`, and say why continuous integration systems only look at that number.
7. You will be able to fix a broken PATH, and say which file to edit on macOS, on a Linux server, and why the answer differs between the two.
8. You will be able to explain the Windows registry properly, and say why a macOS or Linux developer almost never thinks about one.
9. You will be able to use around forty commands with confidence, and read a manual page to find the fortieth-first.
10. You will be able to write a real shell script with error handling that stops instead of continuing into damage.

19.1 What a terminal actually is

■ 19.1.1 PLAIN – in simple words

1. Today, a terminal is a window on your screen with text in it.
2. That is a costume. The word means something older and physical.
3. A **terminal** was a machine that sat at the end of a wire.
4. The other end of the wire went to a computer, often in another building.
5. The terminal had a keyboard and a way to show what came back.
6. It had almost no brain of its own. It sent characters out and printed characters in. Nothing more.
7. The earliest ones printed onto a paper roll, like a till receipt that never ends.
8. Later ones had a small screen instead of paper, and that was a huge saving in paper and noise.
9. When personal computers arrived, nobody needed the physical machine any more. The computer was on your desk.

10. But all the software already knew how to talk to a terminal. So we kept pretending. A program on your screen now acts like the old machine.
11. That is why the window is called a terminal, and why it still obeys commands designed for a device made in 1978.

■ 19.1.2 PLAIN – a picture in your head

1. Imagine a hotel in 1970 with one telephone switchboard in the basement.
2. Every room has a simple phone. The phone has no brain. It has a handset and a dial.
3. All the intelligence, all the routing, all the connections, live in the basement.
4. The phone in the room is the terminal. The switchboard is the computer.
5. Ten rooms can be connected at once. The basement machine handles all ten, giving each a slice of its attention.
6. Now the hotel modernizes. Every room gets its own full computer. The basement machine is thrown away.
7. But the guests learned to use the handset. So the new computers put a picture of a handset on their screens, and it works the same way.
8. That picture is the terminal emulator you use today.

Where this comparison breaks: a hotel phone carries sound as a continuous wave, while a terminal carries discrete characters, one byte at a time, with a strict alphabet. And the old terminal was not entirely brainless. It knew how to move its own cursor, clear its own screen, and pick a colour, when told to by special character sequences. That small amount of local intelligence is exactly the part that survives in your terminal window today.

■ 19.1.3 PLAIN – a worked example

1. Picture the Teletype Model 33, sold from 1963. Engineers called it the ASR-33.
2. ASR stands for Automatic Send-Receive: it could also punch and read paper tape.
3. It weighed around 25 kilograms. It printed onto a roll of paper. It was loud, like a typewriter that never stopped.
4. It ran at 110 bits per second. That is 10 characters per second.
5. Type a line of 80 characters and wait 8 seconds for the reply to finish printing. That was the normal rhythm of computing.
6. Because printing was slow, command names were made short. This is the real reason UNIX has `ls`, `cp`, `mv`, `rm` and not `list`, `copy`, `move`, `remove`.
7. There is no scrolling back on a paper terminal. What is printed is printed. To see something again, you print it again.
8. There is no clearing the screen either. There is no screen.
9. Now compare the DEC VT100, introduced in 1978. It had a glass screen showing 24 rows of 80 characters.
10. Suddenly the cursor could be moved anywhere, text could be erased, and the screen could be redrawn. Full-screen text editors became possible.
11. Below is a real capture of the bytes a modern terminal still uses to turn text red and bold. This was taken from the sandbox used to write this chapter.

```
$ printf '\033[1;31mRED BOLD\033[0m\n' | od -c
0000000 033  [  1  ;  3  1  m  R  E  D          B  0
0000020  L  D 033  [  0  m  \n
```

12. 033 is the octal number for the ESC character, decimal 27, hex 1B.

13. ESC [1 ; 3 1 m means “bold, red foreground”. ESC [0 m means “reset everything”.
14. That exact sequence was defined for terminals of the VT100 era. Your terminal in 2026 still speaks it.

■ 19.1.4 PLAIN – what is really happening inside

1. A terminal and a computer are joined by a serial line: one wire for characters going out, one for characters coming in.
2. When you press the A key, the terminal sends the single byte 65 down the wire. It does not draw an A on its own.
3. The computer receives 65, decides what to do, and usually sends 65 straight back. That is called echo.
4. Only when the byte comes back does the terminal print it. So on a slow or broken line, your typing does not appear.
5. This is why an old terminal feels like a conversation: everything you see is something the far end chose to send you.
6. Most bytes mean “print this character”. A few mean “do this action”.
7. Byte 7 rings the bell. Byte 8 moves the print head back one space. Byte 10 moves down a line. Byte 13 returns to the left margin.
8. Byte 27, ESC, means “the next few bytes are not text, they are an instruction”.
9. ESC [starts a control sequence. Numbers and a final letter follow.
10. ESC [2 J clears the screen. ESC [10 ; 5 H puts the cursor on row 10, column 5.
11. Every manufacturer invented their own sequences at first, which was chaos. A standard fixed that, and the VT100 was the terminal that made the standard popular.
12. In UNIX, every device is a file. The file that represents the connection to a terminal was named after the machine on the other end.
13. Teletype was shortened to tty. The device is /dev/tty. That is the whole origin of the name.

■ 19.1.5 TECHNICAL – the engineer’s version

1. The Teletype Corporation Model 33 was introduced as a commercial product in 1963, after an original design for the United States Navy.
2. It was one of the first products to use ASCII, first published in 1963. Over half a million Model 33 units had been built by 1975.
3. The Model 33 defined by accident two long-lived conventions: code 17 (Ctrl-Q, DC1) as XON and code 19 (Ctrl-S, DC3) as XOFF for flow control.
4. Ctrl-S still freezes many terminals today, and confused users still think their machine has crashed. It has not. Press Ctrl-Q.
5. The DEC VT52 arrived in September 1975 with a proprietary escape set. The DEC VT100 arrived in 1978 and implemented the emerging ANSI sequences.
6. The relevant standards, in order: ECMA-48 adopted 1976; ANSI X3.41-1974 and ANSI X3.64-1977 cited in the VT100 manual; the name “ANSI escape sequence” dates from ANSI’s 1979 adoption of X3.64.
7. ECMA-48 and X3.64 were merged into ISO/IEC 6429. ANSI withdrew X3.64 in 1994 in favour of the international standard. Japan adopted JIS X 0211.
8. So the correct name in 2026 is ECMA-48 or ISO/IEC 6429. “ANSI escape codes” is a convention of speech, not a live standard name.
9. The VT100 used an Intel 8080 processor, showed 24 lines of 80 columns, and supported 132-column mode. Its market success created a clone industry: Zenith Z-19 in 1979, Qume QVT-108, Televideo TVI-970, Wyse WY-99GT.

10. Because sequences still varied, UNIX grew a capability database: `termcap`, then `terminfo`. Programs ask the database, not the hardware.
11. The `TERM` environment variable names your terminal type. `tput` and `infocmp` read the database. Real output from this sandbox:

```
$ TERM=xterm-256color tput colors
256
$ TERM=xterm-256color tput setaf 1 | od -c
0000000 033  [  3  1  m
$ infocmp xterm | head -5
xterm|xterm-debian|xterm terminal emulator (X Window System),
    am, bce, km, mc5i, mir, msgr, npc, xenl,
    colors#8, cols#80, it#8, lines#24, pairs#64,
    bel=^G, blink=\E[5m, bold=\E[1m, cbt=\E[Z,
```

12. Note the honest detail: plain `xterm` claims 8 colours, `xterm-256color` claims 256. Setting `TERM` wrongly is a common cause of broken colours and broken arrow keys over SSH.

Machine	Year	Output	Speed
Teletype 33ASR	1963	paper roll	110 bit/s
DEC VT52	1975	24x80 screen	up to 19200
DEC VT100	1978	24x80 screen	up to 19200
Modern emul.	2026	window	memory speed

13. The honest version: your terminal is not emulating a VT100 exactly. It emulates a superset, usually announced as `xterm-256color`. Mouse reporting, bracketed paste, true colour and Unicode are all later additions that no VT100 ever had.

■ 19.1.6 WORDS – remember these

1. Terminal — a machine at the end of a wire for typing and reading — a character-oriented I/O device with a keyboard and display.
2. Teleprinter — a typewriter that talks over a wire — an electromechanical send-receive device using a serial character code.
3. ASR-33 — the famous noisy paper terminal — Teletype Model 33 Automatic Send-Receive, 1963, 110 bit/s, 10 characters per second.
4. Escape sequence — a few characters that mean “do something” not “print something” — a control string beginning with ESC, per ECMA-48.
5. `tty` — the name of the terminal device file — abbreviation of teletype, exposed as `/dev/tty` and `/dev/ttyN`.
6. `terminfo` — a list of what each terminal model can do — a compiled capability database consulted through `tput` and `curses`.
7. Echo — the far end sending your keystroke back so you can see it — local or remote reflection of input characters, controlled by the line discipline.

19.2 Terminal, terminal emulator, shell, console and command line

■ 19.2.1 PLAIN – in simple words

1. Five words get used as if they mean the same thing. They do not.
2. A **terminal** is the device, real or pretended, that carries characters in and out.

3. A **terminal emulator** is a program that draws a terminal on your screen. Terminal.app and iTerm2 on macOS are terminal emulators.
4. A **shell** is a different program entirely. It reads what you type and runs programs for you. bash and zsh are shells.
5. A **console** originally meant the one screen physically attached to the machine, used for its most important messages.
6. A **command line** is not a program at all. It is a style of working: you type a line, you press Enter, something happens.
7. When you open Terminal.app on a Mac, you start a terminal emulator, and it starts a shell inside itself.
8. The window is the emulator. The % or \$ prompt inside it comes from the shell.
9. If the shell crashes, the window stays but nothing responds. If the window closes, the shell dies with it.

■ 19.2.2 PLAIN – a picture in your head

1. Think of a theatre.
2. The building is the terminal emulator. It has seats, lighting, a stage, and a door to the street.
3. The play performed on the stage is the shell. It is what you actually came to watch.
4. The building does not know the plot. It only supplies a place to perform and a way for the audience to see and be heard.
5. You can put a different play in the same building. Run zsh instead of bash and nothing about the window changes.
6. You can also put the same play in a different building. Run bash under iTerm2 instead of Terminal.app and the play is identical.
7. The console is the stage manager's desk backstage: the one place that gets the emergency announcements even when the audience does not.

Where this comparison breaks: a theatre and a play are physically separate, while your emulator and shell are joined by a specific and strange piece of kernel plumbing, not by air. And the shell can run with no building at all, reading commands from a file, with nobody watching. That is a shell script.

■ 19.2.3 PLAIN – a worked example

1. Open Terminal.app on macOS. You now have four things stacked up.
2. Terminal.app itself is a normal macOS application, written with Apple's UI toolkit. It draws characters and reads your keyboard.
3. It asked the operating system for a pseudo-terminal, a fake wire.
4. It started /bin/zsh on the far end of that fake wire.
5. zsh printed a prompt into the fake wire, and Terminal.app drew it.
6. Prove it to yourself. Type `tty` and press Enter. On macOS you will see something like `/dev/ttys003`. On Linux you will see `/dev/pts/0`.
7. Real output from the Linux sandbox used to write this chapter, running a shell inside a real pseudo-terminal:

```
$ tty
/dev/pts/0
$ echo $SHELL
/bin/bash
$ ps -o pid,ppid,pgid,sid,tpgid,stat,TTY,comm
PID  PPID  PGID   SID  TPGID  STAT  TT      COMMAND
```

```

30511 30510 30511 30511 30512 Ss pts/0 bash
30512 30511 30512 30511 30512 R+ pts/0 ps

```

8. Read that table. `bash` has process ID 30511. `ps` has 30512 and its parent is 30511, so the shell started it.
9. Both are attached to terminal `pts/0`. That is the fake wire.
10. `TPGID` is 30512, meaning the terminal's foreground group is currently `ps`, not the shell. That is what "the shell is waiting" looks like from outside.

■ 19.2.4 PLAIN - what is really happening inside

1. The emulator owns one end of the fake wire. The shell owns the other.
2. Everything you type goes into the emulator, which writes those bytes into its end.
3. The operating system carries them across and hands them to the shell.
4. Everything the shell or its programs print goes the other way, and the emulator draws it.
5. The emulator has one extra job: it must understand escape sequences. When bytes `ESC [ 2 J` arrive, it clears its own drawing area.
6. The shell has one extra job: it must decide what your line of text means.
7. Neither knows much about the other. You can replace either one.
8. The console is a separate idea. On a Linux server it is where kernel messages appear, even before any emulator exists.
9. On a physical Linux machine, pressing `Ctrl-Alt-F2` gives you a real text console, `/dev/tty2`, drawn by the kernel itself with no emulator involved.
10. On macOS there is no such thing for the user. There is a boot-time verbose console and a system log, but no switchable text consoles.
11. Windows is different again. Its window is `conhost.exe` or Windows Terminal, and the thing inside is `cmd.exe` or `powershell.exe`.

■ 19.2.5 TECHNICAL - the engineer's version

Thing	What it is	Examples
Terminal	character device	<code>tty</code> , <code>pty</code> , serial
Terminal emulator	GUI application	<code>Terminal.app</code> , <code>iTerm2</code>
Shell	command language	<code>bash</code> , <code>zsh</code> , <code>fish</code>
Console	primary device	<code>/dev/console</code> , <code>tty1</code>
Command line	interaction mode	not a program

1. On Linux, virtual consoles are `/dev/tty1` through `/dev/tty63`, driven by the kernel VT subsystem, with no user-space emulator.
2. `/dev/console` is the kernel's message destination, set by the `console=` kernel command-line parameter at boot.
3. `/dev/tty` with no number is a magic alias meaning "the controlling terminal of the process reading it".
4. Pseudo-terminal slaves on Linux are `/dev/pts/N`, provided by the `devpts` filesystem. On macOS and BSD they are `/dev/ttysNNN`.
5. Common emulators, with real facts: `xterm`, first released 1984, still the compatibility reference. GNOME Terminal, KDE Konsole, Alacritty, `kitty`, `WezTerm`, `iTerm2` on macOS, Windows Terminal from Microsoft.
6. Windows Terminal reached version 1.0 on 19 May 2020. Before it, Windows had only the legacy console host.

7. macOS Terminal.app ships with macOS. It defaults TERM to xterm-256color in recent versions. iTerm2 is a third-party replacement with split panes and its own escape extensions.
8. The distinction matters operationally. Colour problems and key problems are emulator or TERM problems. Completion and prompt problems are shell problems. Never debug one by changing the other.

■ 19.2.6 WORDS – remember these

1. Terminal emulator — the window program — a user-space application that allocates a pty and renders ECMA-48 output.
2. Shell — the program that runs your commands — a command language interpreter, specified for sh by POSIX IEEE Std 1003.1.
3. Console — the machine's own primary screen — /dev/console, the kernel's message device, selected at boot.
4. Virtual console — a full-screen text session on Linux — /dev/ttyN, implemented in the kernel VT layer.
5. Controlling terminal — the terminal that owns your session — the tty a session leader has opened, reachable as /dev/tty.

19.3 The pseudo-terminal

■ 19.3.1 PLAIN – in simple words

1. There is no wire and no machine any more. So the operating system fakes one.
2. The fake is called a **pseudo-terminal**, usually shortened to PTY.
3. It comes as a pair of ends that are joined inside the kernel.
4. One end is held by the terminal emulator. Whatever it writes appears as keyboard input on the other end.
5. The other end is held by the shell. Whatever the shell prints comes out of the first end for the emulator to draw.
6. In between sits a piece of kernel code called the **line discipline**.
7. The line discipline is not a pipe. It is an active thing that changes the characters as they pass.
8. It is why backspace deletes a letter instead of printing a strange symbol.
9. It is why the shell does not see your line until you press Enter.
10. It is why Ctrl-C stops a program, instead of being delivered to it as the letter it technically is.

■ 19.3.2 PLAIN – a picture in your head

1. Imagine two people passing notes through a slot in a wall.
2. On the wall's slot sits a clerk. The clerk reads every note before passing it on.
3. When you write a letter, the clerk copies it onto a public board so you can see what you wrote. That is echo.
4. When you scribble out a letter, the clerk erases it from the board and from the note. That is backspace.
5. The clerk holds each note until you write a full stop. Only then is the note pushed through. That is line buffering.
6. If you write a special mark meaning "stop", the clerk does not pass the mark along. The clerk runs to the other room and pulls the person out of their chair. That is Ctrl-C.
7. The clerk can be told to stop doing all of this. In that mode every letter goes through instantly, unchanged, uncopied.
8. Text editors and games ask for exactly that mode, because they want to react to every single key.

Where this comparison breaks: the clerk is not slow and not optional. It is kernel code running in microseconds, and there is always one, even in the mode where it does almost nothing. Also, the clerk does not understand your notes. It only recognizes about a dozen specific characters and treats every other byte as data to be passed along.

■ 19.3.3 PLAIN – a worked example

1. Here are the special characters the line discipline is watching for, taken from a real `stty -a` run inside a pseudo-terminal on Linux.

```
$ stty -a
speed 38400 baud; rows 0; columns 0; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D;
start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; discard = ^O; min = 1; time = 0;
isig icanon iexten echo echoe echok -noflsh -tostop
```

2. Read the important ones. `intr = ^C` means Ctrl-C is the interrupt key.
3. `susp = ^Z` means Ctrl-Z suspends. `eof = ^D` means Ctrl-D signals end of input.
4. `erase = ^?` means the Delete character erases one letter. `kill = ^U` erases the whole line.
5. `werase = ^W` erases one word. `lnext = ^V` means “take the next key literally, do not treat it specially”.
6. Now the flags on the last line, which are the settings, not the keys.
7. `icanon` means canonical mode is on: input is collected into lines.
8. `echo` means the kernel prints your keystrokes back for you.
9. `isig` means the special keys generate signals. Turn this off with `stty -isig` and Ctrl-C becomes an ordinary byte.
10. Notice `speed 38400 baud`. There is no wire, so this number is fiction. It is kept only because programs still ask for it.

■ 19.3.4 PLAIN – what is really happening inside

1. Here is the whole chain, from your finger to a character on the glass.

```
your keyboard
  |
  v
[ window system: macOS AppKit / X11 / Wayland ]
  | key event
  v
[ terminal emulator process ]
  | writes bytes
  v
+-----+
| PTY MASTER (/dev/ptmx) |
+-----+
  | kernel
  v
+-----+
| LINE DISCIPLINE        |
| echo, erase, line buffer, |
| Ctrl-C -> SIGINT,        |
| Ctrl-Z -> SIGTSTP        |
+-----+
  |
  v
+-----+
```

```

| PTY SLAVE  (/dev/pts/0) |
+-----+
| read() returns a line
|
v
[ shell process: bash or zsh ]
| fork + exec
|
v
[ the command you ran ]
| writes to fd 1
|
v
back up through slave, line
discipline, master, emulator,
and onto the screen

```

2. Step by step, when you type `l` then `s` then Enter:
3. The emulator writes the byte `l` into the master.
4. The line discipline copies it back towards the master, so the emulator draws `l`. It also stores it in a small buffer, not yet given to the shell.
5. Same for `s`. The buffer now holds `ls`.
6. You press Enter, which sends carriage return, byte 13. The line discipline translates it to newline, byte 10, and now considers the line complete.
7. The shell's `read()` call, which was blocked and waiting, returns the three bytes `l`, `s`, newline.
8. Now the interesting one. You press Ctrl-C. That is byte 3.
9. The line discipline sees byte 3 matches `intr`. It does not put it in the buffer.
10. Instead it sends the signal `SIGINT` to every process in the terminal's foreground process group.
11. The default action of `SIGINT` is to end the process. So your running command stops. The shell was not in the foreground group, so it survives.
12. Ctrl-D is different. It is not a signal. It means "end of file now".
13. If the buffer has text in it, Ctrl-D pushes that text through immediately.
14. If the buffer is empty, `read()` returns zero bytes, which means end of input, and the shell exits.
15. That is why Ctrl-D on an empty line logs you out, and why pressing it in the middle of a typed line does nothing visible.
16. Ctrl-Z sends `SIGTSTP`, which suspends. The process freezes in place. The shell notices, takes back the terminal, and prints your prompt.

■ 19.3.5 TECHNICAL – the engineer's version

1. A PTY is a bidirectional character device pair. On Linux the master is obtained by opening `/dev/ptmx`, which allocates a new slave in `/dev/pts`.
2. The relevant calls are `posix_openpt`, `grantpt`, `unlockpt`, `ptsname`, then `open` on the slave. `openpty` and `forkpty` wrap all of it.
3. The slave is made the controlling terminal by the child calling `setsid` then `ioctl(TIOCSCTTY)`, or implicitly on first open by a session leader.
4. Terminal settings live in `struct termios` with four flag sets: `c_iflag` input, `c_oflag` output, `c_cflag` control, `c_lflag` local, plus the `c_cc` array of control characters.
5. `ICANON` in `c_lflag` selects canonical mode. Cleared, you are in raw mode and `VMIN` and `VTIME` in `c_cc` control read behaviour.
6. `ECHO`, `ECHOE`, `ECHOK`, `ISIG`, `IEXTEN` are the other `c_lflag` bits you will actually touch. `OPPOST` and `ONLCR` in `c_oflag` translate newline to carriage-return newline on output.
7. Signals generated by the line discipline, with their Linux numbers:

Key	Byte	Signal	Default action
Ctrl-C	0x03	SIGINT (2)	terminate
Ctrl-\	0x1C	SIGQUIT (3)	terminate, core
Ctrl-Z	0x1A	SIGTSTP (20)	stop
none	none	SIGTTIN (21)	stop background
none	none	SIGTTOU (22)	stop background

8. Signals go to the foreground process group of the controlling terminal, found with `tcgetpgrp` and set with `tcsetpgrp`. This is the field shown as TPGID by `ps`.
9. Job control was first implemented in the C shell by Jim Kulp at IASA in Austria, using features of the 4.1BSD kernel, released 1981.
10. A background process that reads from the terminal gets SIGTTIN and stops. That is why a backgrounded interactive program mysteriously freezes.
11. Window size is not part of `termios`. It is `struct winsize` set with `ioctl(TIOCSWINSZ)`, and changing it delivers SIGWINCH, signal 28, to the foreground group.
12. Real job-control evidence from a pseudo-terminal session in this sandbox:

```
demo$ sleep 300 &
[1] 30514
demo$ jobs
[1]+  Running                  sleep 300 &
demo$ ps -o pid,pgid,stat,TTY,comm --ppid $$
  PID  PGID  STAT  TT      COMMAND
 30514 30514  S     pts/0    sleep
 30515 30515  R+    pts/0     ps
demo$ kill %1
```

13. Note that `sleep` has its own PGID equal to its PID. Each job gets its own process group, which is what makes Ctrl-C hit one job and not all of them.
14. The + in R+ means “in the foreground process group”. `sleep` has no plus, because it is a background job.
15. Windows had no equivalent until ConPTY, the Windows Pseudo Console API, which first shipped in the Windows 10 October 2018 update, version 1809. Before that, tools faked it by screen-scraping the console.
16. The honest version: a PTY is not a perfect terminal. There is no real baud rate, no parity, no modem control lines. `stty` will happily accept settings for hardware that does not exist and silently ignore them.

■ 19.3.6 WORDS – remember these

1. PTY — a fake wire between a window and a shell — a pseudo-terminal device pair, master plus slave, joined in the kernel.
2. Line discipline — the kernel code that edits your typing — the tty layer implementing canonical mode, echo and signal generation.
3. Canonical mode — input is handed over one whole line at a time — ICANON set in `termios.c_lflag`, with line editing by the kernel.
4. Raw mode — every key reaches the program instantly — ICANON and ECHO cleared, VMIN and VTIME controlling read behaviour.
5. Foreground process group — the job that owns the keyboard right now — the process group returned by `tcgetpgrp` on the controlling terminal.

6. SIGINT — the polite “stop that” signal from Ctrl-C — signal 2, default action terminate, catchable and ignorable.
7. Job control — running several programs from one terminal — process groups, sessions and the SIGTSTP, SIGCONT, SIGTTIN, SIGTTOU family.

19.4 What a shell is

■ 19.4.1 PLAIN – in simple words

1. A shell is a program with a very small job description.
2. It reads a line of text. It works out what you meant. It asks the operating system to run something. It waits. It prints a prompt again.
3. That is the whole loop. Everything else is decoration on those four steps.
4. The shell is not part of the operating system. It is an ordinary program.
5. You can have several shells installed and switch between them freely.
6. The shell is also a programming language, with variables, conditions and loops. That is what makes shell scripts possible.
7. A few commands are built into the shell itself, because they must be. `cd` is the main one.
8. `cd` cannot be a separate program, because a separate program cannot change its parent’s current directory.
9. Everything else, `ls` and `grep` and `python`, is a real file on disk that the shell finds and launches.

■ 19.4.2 PLAIN – a picture in your head

1. Think of a restaurant with one waiter and a kitchen.
2. You say “two coffees and the soup”. The waiter does not cook.
3. The waiter interprets. “Two coffees” becomes two separate drink orders. “The soup” means today’s soup, so the waiter fills in the detail.
4. The waiter walks to the kitchen and passes the orders. The kitchen is the operating system.
5. The waiter then stands and waits until the food is ready, and brings it back.
6. Then the waiter returns to your table and stands ready again. That is the prompt.
7. If you ask for something not on the menu, the waiter comes back and says so. That is `command not found`.
8. The waiter can also do a few things without the kitchen: move you to another table, remember your name. Those are the built-in commands.

Where this comparison breaks: the waiter understands meaning, and will guess sensibly if you are vague. The shell does not understand anything. It applies fixed textual rules in a fixed order. If those rules turn your line into nonsense, it passes the nonsense to the kitchen without hesitation. A waiter who behaved like a shell would fetch a chair when you said “chair” in the middle of a sentence about the weather.

■ 19.4.3 PLAIN – a worked example

1. You type this and press Enter:

```
| grep -c 500 access.log
```
2. Step 1, read. The shell has the line as a string of 24 characters.
3. Step 2, split into words. It uses spaces and tabs, giving four words: `grep`, `-c`, `500`, `access.log`.
4. Step 3, expand. It checks each word for things it must rewrite: variables, wildcards, braces, tildes, backticks. Here there are none, so nothing changes.

5. Step 4, find the program. `grep` has no slash in it, so the shell searches the directories listed in `PATH`, in order.
6. It finds `/usr/bin/grep` and stops looking.
7. Step 5, run it. The shell makes a copy of itself, and the copy replaces itself with `grep`.
8. Step 6, wait. The shell sleeps until `grep` finishes.
9. `grep` prints 3 and exits with status 0.
10. Step 7, record the result. The shell stores 0 in the variable `?` and prints the prompt.
11. The real run, from this sandbox:

```
$ grep -c 500 access.log
3
$ echo $?
0
```

■ 19.4.4 PLAIN – what is really happening inside

1. The shell cannot simply “become” `grep`, because then it would be gone and you would have no shell left.
2. So it uses a two-step trick that UNIX has used since the beginning.
3. First it calls **fork**. The operating system makes a near-identical copy of the shell process. Two processes now exist, running the same code.
4. Both come back from `fork`, but with different answers. The parent gets the child’s process number. The child gets zero.
5. That difference is how each one knows who it is.
6. The child then calls **exec**. This does not create a process. It throws away the current program’s memory and loads a new program in its place.
7. The process number does not change. Open files do not change. Environment variables do not change. Only the code and data are replaced.
8. This is exactly why redirection works. Between `fork` and `exec`, the child can quietly rearrange its own files, and the new program inherits the arrangement without knowing.
9. The parent calls **wait**. It sleeps until the child ends, and collects the child’s exit number.
10. If you typed `&` at the end, the parent skips the waiting and prints the prompt immediately. That is a background job.
11. Between pressing Enter and seeing output, then: line read, words split, expansions applied, program located, `fork`, file descriptors set up, `exec`, program runs, output travels back up the pseudo-terminal, parent collects the exit status, prompt printed.
12. On a modern machine that whole sequence takes well under a millisecond, plus however long your program actually takes.

■ 19.4.5 TECHNICAL – the engineer’s version

1. The canonical loop, in C-like pseudocode, is short enough to memorize:

```
for (;;) {
    print_prompt();
    line = read_line();           /* from fd 0 */
    words = tokenize(line);
    words = expand(words);        /* order matters, see 19.12 */
    if (is_builtin(words[0])) { run_builtin(words); continue; }
    pid = fork();
    if (pid == 0) {
        setup_redirections();    /* dup2 on 0,1,2 */
        execvp(words[0], words); /* only returns on failure */
        _exit(127);              /* command not found */
    }
}
```

```
    }  
    waitpid(pid, &status, 0);  
    last_status = WEXITSTATUS(status);  
}
```

2. `fork` is defined in POSIX. Linux implements it with `clone`. Modern implementations use copy-on-write, so the copy is cheap: page tables are duplicated, physical pages are shared until written.
3. `vfork` and `posix_spawn` exist as cheaper variants. Real shells often use `fork` anyway because the child needs to run arbitrary setup code.
4. `execvp` is the variant that searches `PATH` and takes an argument vector. The `e` variants take an explicit environment; without them the current `environ` is inherited.
5. A successful `exec` never returns. Any code after it runs only on failure, which is why the example calls `_exit(127)`.
6. Exit status is packed into an integer by `wait`. `WEXITSTATUS` extracts the low 8 bits. `WIFSIGNALED` and `WTERMSIG` cover death by signal.
7. Builtins are required for anything that must change shell state: `cd`, `export`, `exec`, `set`, `unset`, `shift`, `trap`, `read`, `wait`, `eval`, `source`, `umask`, and the job control commands.
8. Some builtins exist only for speed. `echo`, `test`, `l`, `pwd`, `kill` and `printf` all exist as real binaries in `/usr/bin` as well.
9. You can see the duplication:

```
$ type -a echo  
echo is a shell builtin  
echo is /usr/bin/echo  
echo is /bin/echo  
$ type cd  
cd is a shell builtin
```

10. This matters. `/usr/bin/echo -e` and the bash builtin `echo -e` behave differently, and scripts that assume one get the other under `sh`. Use `printf` when it matters. That advice is a convention, not a standard, but it is close to universal among careful script authors.
11. Instrument the whole thing with `strace -f -e trace=execve,clone,wait4` on Linux, or `dtruss` on macOS, which needs elevated privileges and System Integrity Protection considerations.

■ 19.4.6 WORDS – remember these

1. Shell — the program that runs your typed commands — a command language interpreter and scripting language.
2. Built-in — a command the shell performs itself — a function inside the shell binary, not a file in `PATH`.
3. `fork` — make a copy of the running program — the POSIX system call that duplicates a process, returning 0 to the child.
4. `exec` — replace this program with another — the `execve` family, which overwrites the address space and keeps the PID.
5. `wait` — pause until the child is finished — `waitpid`, which reaps the child and returns its termination status.
6. Prompt — the text asking you for a command — a shell-generated string from `PS1`, printed to the terminal before each read.

19.5 The shells themselves

■ 19.5.1 PLAIN – in simple words

1. There have been many shells. A handful matter.
2. The first UNIX shell was written by Ken Thompson in 1971. It could run programs and redirect their input and output. It could not do much else.
3. Stephen Bourne wrote a much better one at Bell Labs, and it shipped as the default in Version 7 UNIX. That is the **Bourne shell**, the program `sh`.
4. It added variables, loops, conditions and functions. It made shell scripting a real thing. Everything since is measured against it.
5. Bill Joy wrote the **C shell** at Berkeley, with a syntax that looked more like the C language, and with job control and command history.
6. David Korn at Bell Labs wrote the **Korn shell**, taking the good ideas from the C shell and putting them into a Bourne-compatible shell.
7. Brian Fox wrote **bash** for the GNU project, free of licence restrictions, and it became the shell of Linux.
8. Paul Falstad wrote **zsh**, which is Bourne-compatible but with far better completion and matching. Apple made it the macOS default in 2019.
9. **fish** broke compatibility on purpose, to be friendly out of the box.
10. **PowerShell** from Microsoft is not in this family at all. It moves objects between commands instead of text.

■ 19.5.2 PLAIN – a picture in your head

1. Think of English and its descendants over centuries.
2. Old English is the Thompson shell: recognizable, but you cannot read a newspaper with it.
3. Middle English is the Bourne shell: the grammar settles, and texts written in it are still readable today with effort.
4. Then two dialects grow in different towns. One is the Korn shell, which keeps the old grammar and adds new words. The other is the C shell, which changes the grammar and confuses travellers.
5. `bash` is the modern standard dialect, taught in schools, understood everywhere, slightly dull.
6. `zsh` is the same language spoken by people who care about pronunciation, with a large vocabulary of convenience.
7. `fish` is a constructed language: cleaner and easier, but nobody else's documents work in it.
8. **PowerShell** is not a dialect of English. It is a different language family entirely, from a different continent.

Where this comparison breaks: languages drift by accident, but shells were designed on purpose, by named people, with written justifications. And unlike English, there is a formal written standard for shell grammar: POSIX. A script written to that standard genuinely runs on all of the Bourne-family shells, which is not something you can say about human dialects.

■ 19.5.3 PLAIN – a worked example

1. The same task, count files ending in `.md`, in four shells.

```
# sh, bash, zsh, ksh: the Bourne family
n=$(ls *.md | wc -l)
echo "there are $n files"

# csh and tcsh: different assignment syntax entirely
set n = `ls *.md | wc -l`
```

```
echo "there are $n files"

# fish: no dollar on assignment, no equals sign
set n (ls *.md | wc -l)
echo "there are $n files"

# PowerShell: no text at all, real objects
$n = (Get-ChildItem *.md).Count
"there are $n files"
```

2. Notice that the Bourne family line is identical across four shells. That is the value of POSIX compatibility.
3. Notice that the C shell needs spaces around = and uses set. Scripts do not port between the families.
4. Notice that PowerShell never counted lines of text. Get-ChildItem returned a list of file objects, and .Count asked the list how long it was.
5. That last point is the whole design difference. In UNIX shells, the thing passing between commands is bytes. In PowerShell it is typed objects with properties.

■ 19.5.4 PLAIN – what is really happening inside

1. Why did so many shells appear? Because the shell is small, personal, and used constantly. Small annoyances are worth fixing.
2. The C shell's improvements were real: job control, history with !!, aliases, directory stacks.
3. Its scripting was genuinely bad. A widely circulated essay by Tom Christiansen, "Csh Programming Considered Harmful", listed the reasons, and the argument was largely won.
4. The Korn shell showed you could have both: Bourne syntax plus the interactive features. It became the commercial UNIX default.
5. bash exists for a legal reason as much as a technical one. GNU needed a shell it could ship freely, without AT&T code.
6. zsh went further on interactive quality: completion that understands the command you are typing, spelling correction, shared history, better globbing.
7. Apple's move to zsh in 2019 was also partly legal. bash version 4 changed to the GPL version 3 licence, which Apple will not ship, so macOS was frozen on bash 3.2 from 2007.
8. fish decided that compatibility was the thing holding shells back. It has syntax highlighting and autosuggestions with no configuration at all.
9. PowerShell came from a different problem. Windows configuration is not text files, so parsing text was useless. Passing objects was the natural answer on that system.

■ 19.5.5 TECHNICAL – the engineer's version

Shell	Year	Author	Note
sh (V6)	1971	Ken Thompson	first UNIX shell
sh	1979	Stephen Bourne	shipped in V7 UNIX
csh	1978	Bill Joy	job control, history
ksh	1983	David Korn	Bourne plus csh ideas
bash	1989	Brian Fox	GNU, Linux default
zsh	1990	Paul Falstad	macOS default 2019
fish	2005	A. Liljencrantz	not POSIX by design
PwrShell	2006	Jeffrey Snover	object pipeline

1. Precise dates worth knowing. Bash 1.0 was released on 8 June 1989 by Brian Fox at the Free Software Foundation. The name is Richard Stallman's pun: Bourne-again shell.
2. zsh 1.0 was released in 1990 by Paul Falstad, then a sophomore at Princeton University. The name comes from the login ID of a Princeton teaching assistant, Zhong Shao.
3. fish 1.0 was released on 13 February 2005 by Axel Liljencrantz. The name is "friendly interactive shell".
4. PowerShell's design was published in the Monad Manifesto by Jeffrey Snover in August 2002. It was demonstrated in October 2003, renamed PowerShell on 25 April 2006, and version 1.0 shipped on 14 November 2006.
5. PowerShell Core 6.0, released January 2018, made it cross-platform and open source. It runs on Linux and macOS today.
6. macOS 10.15 Catalina, released 7 October 2019, made zsh the default login shell for new accounts. Existing accounts kept bash. Apple ships `/bin/bash` as version 3.2.57 for licence reasons and it is not updated.
7. The POSIX shell is specified in IEEE Std 1003.1, the Shell and Utilities volume. `sh` on Debian and Ubuntu is dash, not bash, which is a common source of "works on my machine" script failures.
8. Verified in this sandbox:

```
$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 Mar 31 2024 /bin/sh -> dash
$ bash --version | head -1
GNU bash, version 5.2.21(1)-release (x86_64-pc-linux-gnu)
```
9. Practical advice, and this is opinion held by most working engineers rather than a rule: write scripts for `sh` or `bash`, use whatever you like interactively. Never write scripts in `csh`. Never assume `#!/bin/sh` gives you `bash`.
10. Where experts disagree: some argue that fish and zsh's improvements should be adopted in scripts too, and that POSIX compatibility is a museum concern. Others point out that servers, containers and rescue images often contain only `sh`, so portable scripts still pay. Both are right in their own context.

■ 19.5.6 WORDS - remember these

1. Bourne shell — the original serious shell — `sh`, from Version 7 UNIX 1979, ancestor of the POSIX shell grammar.
2. POSIX shell — the written standard for shell syntax — IEEE Std 1003.1 Shell and Utilities volume.
3. `bash` — the common Linux shell — Bourne-again shell, GNU project, first released June 1989.
4. `zsh` — the macOS default since 2019 — Z shell, Bourne-compatible with advanced completion and globbing.
5. `dash` — a small fast `sh` — Debian Almquist shell, POSIX-only, `/bin/sh` on Debian and Ubuntu.
6. Object pipeline — passing structured data instead of text — PowerShell's model, where commands emit and consume .NET objects.

19.6 stdin, stdout and stderr

■ 19.6.1 PLAIN - in simple words

1. Every program starts life with three channels already open.
2. Channel 0 is **standard input**. It is where the program reads from. By default, your keyboard.
3. Channel 1 is **standard output**. It is where results go. By default, your screen.
4. Channel 2 is **standard error**. It is where complaints go. Also your screen, by default.

5. Output and errors are separate on purpose. This is one of the best design decisions in UNIX.
6. If they were mixed, saving a program's results to a file would also save its error messages into the same file, ruining the data.
7. Because they are separate, you can capture the results and still see the errors on your screen, live.
8. The shell lets you point any of these channels somewhere else before the program starts.
9. That is called **redirection**, and the program never knows it happened.

■ 19.6.2 PLAIN – a picture in your head

1. Think of a factory machine with three pipes attached.
2. One pipe brings raw material in at the top.
3. One pipe sends finished product out of the front.
4. One pipe sends scrap and warning notes out of the side.
5. The machine does not know or care where the pipes go. It just pushes things into them.
6. Before switching the machine on, you can move any pipe. Put the product pipe into a barrel. Leave the scrap pipe pointing at the floor so you notice it.
7. Or connect the product pipe of one machine to the input pipe of the next machine. Now you have a production line.
8. Nothing inside either machine changed. Only the plumbing.

Where this comparison breaks: real pipes have no memory, but these have a small buffer, so a fast machine can run ahead of a slow one for a while. And the scrap pipe is not really for scrap. Progress messages, prompts and warnings all come out of it, including from programs that are working perfectly.

■ 19.6.3 PLAIN – a worked example

1. Here is a real run. The directory has `a.txt` and does not have `nope.txt`.
2. First, no redirection at all. Both streams reach the screen and get mixed:

```
$ ls a.txt nope.txt
ls: cannot access 'nope.txt': No such file or directory
a.txt
```

3. Now send only standard output to a file:

```
$ ls a.txt nope.txt > out.txt
ls: cannot access 'nope.txt': No such file or directory
$ cat out.txt
a.txt
```

4. The error still appeared live on the screen. The file has only the good result. That is exactly what you want.
5. Now send only standard error to a file:

```
$ ls a.txt nope.txt 2> err.txt
a.txt
$ cat err.txt
ls: cannot access 'nope.txt': No such file or directory
```

6. Now send both to the same file, correctly:

```
$ ls a.txt nope.txt > both.txt 2>&1
$ cat both.txt
ls: cannot access 'nope.txt': No such file or directory
a.txt
```


7. Now the same two pieces in the wrong order. Watch what happens:

```
$ ls a.txt nope.txt 2>&1 > wrong.txt
ls: cannot access 'nope.txt': No such file or directory
$ cat wrong.txt
a.txt
```

8. The error went to the screen, not the file. The redirection did not work.
9. Why: `2>&1` means “make channel 2 point wherever channel 1 points right now”. At that moment channel 1 still points at the screen.
10. Only afterwards did `> wrong.txt` move channel 1 to the file. Channel 2 was already aimed at the screen and stayed there.
11. The rule to remember: redirections are applied left to right, and `2>&1` copies the current destination, not a promise to follow.
12. So `> file 2>&1` is right and `2>&1 > file` is almost always a mistake.

■ 19.6.4 PLAIN – what is really happening inside

1. The three channels are numbers, not names. The number is a **file descriptor**: an index into a table the kernel keeps for your process.
2. Entry 0, 1 and 2 are filled in before your program starts. They are not special to the kernel. They are special only by agreement.
3. When you run a program from a terminal, all three entries point at the same pseudo-terminal device. That is why output and errors both land on screen.
4. When you write `> out.txt`, the shell does this between fork and exec:
5. It opens `out.txt` for writing, creating or emptying it. The kernel gives back the lowest free number, say 3.
6. It calls `dup2(3, 1)`. That copies entry 3 over entry 1. Entry 1 now points at the file.
7. It closes 3, which is no longer needed. Then it calls `exec`.
8. The new program starts with entry 1 already pointing at the file. It writes to 1 as usual, with no idea anything is different.
9. `2>&1` is just `dup2(1, 2)`. Copy whatever entry 1 currently holds into entry 2.
10. That is the entire mechanism. There is no magic and no cooperation from the program.
11. You can see the table on Linux. Every process has one under `/proc`:

```
$ ls -l /proc/self/fd
lr-x----- 0 -> /dev/null
l-wx----- 1 -> pipe:[159625]
l-wx----- 2 -> /dev/null
lr-x----- 3 -> /proc/10189/fd
```

12. In that real capture, the process was reading nothing, writing into a pipe, and throwing errors away.

■ 19.6.5 TECHNICAL – the engineer’s version

Syntax	Meaning	Underlying call
<code>> f</code>	stdout to f, truncate	<code>open O_TRUNC, dup2</code>
<code>>> f</code>	stdout to f, append	<code>open O_APPEND</code>
<code>< f</code>	stdin from f	<code>open O_RDONLY</code>
<code>2> f</code>	stderr to f	<code>dup2 to fd 2</code>
<code>2>&1</code>	stderr to current stdout	<code>dup2(1, 2)</code>

Syntax	Meaning	Underlying call
&> f	both to f (bash, zsh)	two dup2 calls
<<< str	here-string as stdin	temp file or pipe
<< EOF	here-document as stdin	temp file or pipe
2>/dev/null	discard errors	open the null dev
>&-	close the descriptor	close(1)

1. &> and &>> are bash and zsh extensions. The portable POSIX spelling is `> file 2>&1`. Use the portable form in scripts with `#!/bin/sh`.
2. |& in bash is shorthand for `2>&1 |`. It is also not POSIX.
3. Buffering is a libc behaviour, not a kernel one, and it surprises people. The C standard library uses line buffering for stdout when it is a terminal and full buffering, typically 4096 or 8192 bytes, when it is a pipe or file.
4. stderr is unbuffered by default. That is why error messages can appear before the output that logically came first.
5. Fix it when it matters with `stdbuf -o0 -e0` command, or `unbuffer` from the expect package, or in Python with `python3 -u`.
6. /dev/null is the discard device, major 1 minor 3 on Linux. Writes succeed and vanish, reads return end of file immediately.
7. /dev/stdout, /dev/stderr and /dev/fd/N exist on Linux and macOS and let you name descriptors as paths. Useful with tools that only accept a filename.
8. Descriptors above 2 are yours to use. `exec 3< file` opens a private read channel that survives across commands until `exec 3<&-` closes it.
9. Inspect a live process's descriptors with `lsuf -p PID`, or on Linux read `/proc/PID/fd`. Real `lsuf` output from this sandbox:

```
$ lsuf -i -P -n | head -3
COMMAND PID USER FD  TYPE NAME
claude   462 root 11u IPv4 TCP 192.0.2.2:39400->160.79.104.10:443
claude   462 root 13u IPv4 TCP 127.0.0.1:43279 (LISTEN)
```

10. The honest version: “everything is a file” is a slogan, not a fact. Descriptors point to file descriptions, which may be regular files, pipes, sockets, devices, epoll instances, timers or signal queues. They all support read and write but not equally: you cannot seek in a pipe, and write to a socket can partially succeed.

■ 19.6.6 WORDS – remember these

1. File descriptor — the number a program uses to name an open channel — a small non-negative integer indexing the process file descriptor table.
2. stdin — where a program reads from — file descriptor 0, `FILE *stdin` in C.
3. stdout — where results go — file descriptor 1, line buffered to a terminal, block buffered otherwise.
4. stderr — where complaints go — file descriptor 2, unbuffered by default, kept separate so results stay clean.
5. Redirection — pointing a channel somewhere else — open plus `dup2` performed by the shell between `fork` and `exec`.
6. /dev/null — the bin — the null device, discards writes, returns EOF on read.
7. Here-document — inline text fed to a program as input — the `<< WORD` construct, terminated by `WORD` on its own line.

19.7 Pipes

■ 19.7.1 PLAIN – in simple words

1. A pipe connects the output of one program straight to the input of the next.
2. You write it with the vertical bar: `a | b`.
3. Nothing is saved on disk. The data goes from one program to the other through memory.
4. Both programs run at the same time. The second does not wait for the first to finish.
5. This means you can process something enormous without ever holding all of it at once.
6. It also means each program can be small, do one job, and know nothing about the others.
7. That combination is generally considered the single best idea in UNIX.
8. Doug McIlroy proposed it at Bell Labs, and it was added to UNIX in 1973.
9. Before pipes, you saved to a temporary file and read it back. Everyone did it. Nobody enjoyed it.

■ 19.7.2 PLAIN – a picture in your head

1. Think of a bucket chain at a fire, before fire engines existed.
2. Ten people stand in a line from the well to the fire.
3. The first person does not fill a thousand buckets and then start passing them. They pass each bucket as soon as it is full.
4. Water starts arriving at the fire within seconds, not hours.
5. Nobody in the chain knows where the water comes from or where it goes. Each person knows only “take from the left, give to the right”.
6. If a person in the middle is slow, the people behind them naturally wait. Nobody needs to coordinate this. The line just backs up.
7. If the fire is put out, the person at the end walks away. The next person tries to hand over a bucket, finds nobody there, and stops too.
8. That last case is important, and it has a name in UNIX. We come back to it.

Where this comparison breaks: real people can hold one bucket, while a pipe holds a fixed amount of data, 65536 bytes on Linux by default. And the writer does not “see” that the reader has gone. It gets told, forcefully, by a signal.

■ 19.7.3 PLAIN – a worked example

1. Here is a real log file, eight lines, from this sandbox. We will find which client caused the most server errors.

```
10.0.0.7 ... "GET /index.html HTTP/1.1" 200 5120
10.0.0.9 ... "GET /style.css HTTP/1.1" 200 812
10.0.0.7 ... "GET /missing HTTP/1.1" 404 152
10.0.0.12 ... "POST /api/login HTTP/1.1" 500 90
10.0.0.7 ... "GET /api/user HTTP/1.1" 500 90
10.0.0.9 ... "GET /index.html HTTP/1.1" 200 5120
10.0.0.12 ... "GET /api/user HTTP/1.1" 500 90
10.0.0.3 ... "GET /index.html HTTP/1.1" 200 5120
```

2. Stage 1, keep only the server errors. Real output:

```
$ grep " 500 " access.log
10.0.0.12 ... "POST /api/login HTTP/1.1" 500 90
10.0.0.7 ... "GET /api/user HTTP/1.1" 500 90
10.0.0.12 ... "GET /api/user HTTP/1.1" 500 90
```

3. Stage 2, keep only the first field, which is the client address:

```
$ grep " 500 " access.log | cut -d' ' -f1
10.0.0.12
10.0.0.7
10.0.0.12
```

4. Stage 3, sort, so identical lines sit together:

```
$ ... | sort
10.0.0.12
10.0.0.12
10.0.0.7
```

5. Stage 4, collapse the runs and count them:

```
$ ... | uniq -c
  2 10.0.0.12
  1 10.0.0.7
```

6. Stage 5, sort by that count, largest first:

```
$ ... | sort -rn
  2 10.0.0.12
  1 10.0.0.7
```

7. Stage 6, keep the top two:

```
$ grep " 500 " access.log | cut -d' ' -f1 | sort | uniq -c \
| sort -rn | head -2
  2 10.0.0.12
  1 10.0.0.7
```

8. Six programs. None of them knows about logs. Each does one small thing.
9. `uniq -c` only collapses adjacent duplicates, which is exactly why `sort` must come before it. This trips up everyone once.
10. The same pipeline works unchanged on a log of eight lines or eight hundred million, and uses the same tiny amount of memory either way.

■ 19.7.4 PLAIN – what is really happening inside

1. When the shell sees `a | b`, it asks the kernel for a pipe.
2. The kernel creates a small buffer in memory and gives back two file descriptors: one you can read from, one you can write to.
3. The shell forks twice, once for `a` and once for `b`.
4. In the child that will run `a`, it points descriptor 1 at the write end.
5. In the child that will run `b`, it points descriptor 0 at the read end.
6. Both children close the ends they do not need. This matters enormously.
7. Then both children exec. Both are now running at the same time.
8. `a` writes as usual to descriptor 1, thinking it is writing to the screen.
9. `b` reads as usual from descriptor 0, thinking it is reading a keyboard.
10. When the buffer is full, `a`'s next write simply blocks. The kernel puts it to sleep until `b` takes some data out.
11. When the buffer is empty, `b`'s read blocks until `a` puts something in.
12. This automatic waiting is called back pressure, and it is why a pipeline cannot run out of memory no matter how much data flows through.
13. Here is real proof from this sandbox that a pipe streams rather than waits:

```
$ time (seq 1 50000000 | head -1)
1
real    0m0.002s
```

14. Producing fifty million numbers would take seconds. It finished in two thousandths of a second, because head stopped after one line and seq was killed.
15. Now the ending case. When b exits, the read end is closed.
16. The next time a writes, the kernel sees there is no reader left and sends a the signal SIGPIPE.
17. The default action of SIGPIPE is to kill the process silently. That is by design: it stops a producer from filling the world with data nobody wants.
18. This is why yes | head -3 ends instead of running forever.

■ 19.7.5 TECHNICAL – the engineer’s version

1. The system call is pipe(int fd[2]), giving fd[0] for reading and fd[1] for writing. pipe2 adds flags such as O_CLOEXEC.
2. On Linux the default pipe capacity is 65536 bytes, which is sixteen pages of 4096 bytes. Verified here:

```
$ python3 -c 'import os, fcntl
> r, w = os.pipe()
> print(fcntl.fcntl(w, 1032))'
65536
$ cat /proc/sys/fs/pipe-max-size
1048576
```

3. PIPE_BUF, which is 4096 on Linux and 512 minimum by POSIX, is a different number. It is the largest write guaranteed to be atomic when several writers share one pipe.
4. Capacity can be changed per pipe with fcntl(fd, F_SETPIPE_SZ, n) up to /proc/sys/fs/pipe-max-size. On macOS the pipe buffer starts at 16384 bytes and the kernel may grow it; there is no F_SETPIPE_SZ.
5. Closing unused ends is mandatory. If the shell left the write end open in the reader, the reader would never see end of file and the pipeline would hang forever. This is the classic pipe bug.
6. SIGPIPE is signal 13. The shell reports death by signal N as exit status 128 plus N, so a SIGPIPE death shows as 141.

```
$ yes | head -3 >/dev/null; echo "${PIPESTATUS[@]}"
141 0
$ seq 1 100000000 | head -2 >/dev/null; echo "${PIPESTATUS[@]}"
141 0
```

7. PIPESTATUS is a bash array holding every stage’s status. zsh calls it pipestatus. The plain \$? gives only the last stage, which is why a failing first stage is invisible by default.
8. set -o pipefail makes the pipeline’s status the rightmost non-zero status. Verified:

```
$ set -o pipefail; yes | head -3 >/dev/null; echo $?
141
```

9. Servers and daemons usually call signal(SIGPIPE, SIG_IGN) and check for EPIPE from write instead, because a killed web server is worse than a dropped client.
10. Doug McIlroy proposed pipes at Bell Labs; Ken Thompson implemented them in Version 3 UNIX in 1973. The vertical bar notation and the tee command date from that period.

11. Named pipes, or FIFOs, are the same buffer with a filesystem name, created with `mkfifo`. Opening one for writing blocks until a reader appears, which is a real behaviour, demonstrated in this sandbox: a writer sat blocked for a full second until `cat` opened the other end.
12. Process substitution, `<(command)`, is `bash` and `zsh` only. It creates a FIFO or a `/dev/fd` entry and substitutes its path, letting you feed a command's output to a tool that demands a filename.

■ 19.7.6 WORDS – remember these

1. Pipe — a direct connection from one program's output to another's input — an anonymous unidirectional kernel buffer with two file descriptors.
2. Back pressure — a fast producer waiting for a slow consumer — blocking of `write` when the pipe buffer is full.
3. `SIGPIPE` — the “nobody is listening” signal — signal 13, sent to a writer whose read end has closed, default action terminate.
4. `PIPESTATUS` — the list of results from every stage — a `bash` array of exit statuses for the last foreground pipeline.
5. FIFO — a pipe with a name on disk — a named pipe created by `mkfifo`, type `p` in `ls -l`.
6. `tee` — split the flow so you can see it and save it — a filter that copies `stdin` to `stdout` and to named files.

19.8 Exit codes

■ 19.8.1 PLAIN – in simple words

1. Every program, when it finishes, hands back a single number.
2. That number is called the **exit code** or exit status.
3. It has nothing to do with what the program printed. It is a separate, private answer to one question: did this work?
4. Zero means success. Anything else means failure.
5. That feels backwards. It is not. There is one way to succeed and many ways to fail, so zero is the single success and the rest label the failure.
6. The number is small: 0 to 255 only. It cannot carry a message, only a code.
7. You see the last one with `echo $?`.
8. The shell uses it to decide whether to run the next command when you chain commands together.
9. Automated build systems test only this number. Not your output. Not your colours. The number.

■ 19.8.2 PLAIN – a picture in your head

1. Think of a factory quality inspector at the end of a line.
2. The inspector does not describe the product. There is a label for that.
3. The inspector stamps one thing on the crate: a code.
4. Code 0 means “passed”. Any other code means “rejected”, and different codes say which test it failed.
5. The next station on the line reads only the stamp. It never opens the crate.
6. If the stamp is 0, the crate moves on. If not, the line stops and an alarm sounds.
7. A crate can contain a beautiful description of a disaster and still be stamped 0, if the inspector was careless. Then the disaster moves down the line unnoticed.

Where this comparison breaks: the stamp is applied by the program itself, and a badly written program can stamp 0 on anything. This happens constantly and is the single most common cause of a

build system reporting success on a broken build. The check is only as honest as the program doing the checking.

■ 19.8.3 PLAIN – a worked example

1. Real runs from this sandbox. Each shows the command, the message, and the code.

```
$ ls /nonexistent
ls: cannot access '/nonexistent': No such file or directory
$ echo $?
2

$ true; echo $?      -> 0
$ false; echo $?     -> 1

$ grep -q zzz /etc/hostname; echo $?
1

$ bash -c 'exit 42'; echo $?
42

$ /etc/hostname; echo $?      # a file that is not executable
126

$ nosuchcommand123; echo $?
127

$ bash -c 'kill -INT $$'; echo $?
130

$ bash -c 'kill -TERM $$'; echo $?
143
```

2. Read the meanings. `grep` returning 1 is not an error. It means “I searched correctly and found nothing”.
3. That distinction is why `grep -q pattern file && do_something` works cleanly. It is also why `grep` returning 2 means a real error, like an unreadable file.
4. 126 means “found it, but could not run it”. Usually a missing execute permission.
5. 127 means “could not find it at all”. If you see 127 in a build log, look at `PATH` first.
6. 130 is 128 plus 2, and signal 2 is `SIGINT`. Somebody pressed Ctrl-C.
7. 143 is 128 plus 15, and signal 15 is `SIGTERM`. Something asked it to stop, usually a timeout or an orchestrator.
8. 137 is 128 plus 9, `SIGKILL`. In a container that almost always means the memory limit was hit and the kernel killed it.
9. Now chaining. Real output:

```
$ true && echo "ran because true"
ran because true
$ false && echo "not printed"
$ false || echo "ran because false"
ran because false
$ true || echo "not printed"
$ false ; echo "semicolon always runs"
semicolon always runs
```

10. A `&&` B runs B only if A succeeded. A `||` B runs B only if A failed.
11. A `;` B runs B either way. This is the one that hides failures.

■ 19.8.4 PLAIN – what is really happening inside

1. When a program calls `exit(3)`, the number 3 goes to the kernel.
2. The process dies, but a small record stays behind holding that number.
3. The parent calls `wait`, collects the record, and the record is freed. A process whose record has not yet been collected is a **zombie**.
4. The kernel packs more than the exit code into that record. It also stores whether the process was killed by a signal, and which one.
5. The shell unpacks it. If the process exited normally, `$?` is the code. If it was killed by signal `N`, the shell reports 128 plus `N`.
6. That 128 rule is a shell convention, not a kernel fact. The kernel keeps the two cases genuinely separate.
7. Why the limit of 255: the exit status field is eight bits wide in the traditional `wait` encoding.
8. So `exit 256` becomes 0 and `exit -1` becomes 255. Both are silent traps.
9. In a pipeline, `$?` reports only the last command. The others are lost unless you ask for them.
10. `set -e` makes the shell exit as soon as any command returns non-zero. It is how you stop a script from carrying on after a disaster.

■ 19.8.5 TECHNICAL – the engineer’s version

Code	Meaning	Typical source
0	success	everything
1	general failure	most tools
2	misuse or real error	grep, ls, bash
64–78	sysexits.h categories	BSD-style tools
126	found but not executable	shell
127	command not found	shell
128+N	killed by signal N	shell reporting
130	Ctrl-C, SIGINT	interactive use
137	SIGKILL, often out of memory	containers
143	SIGTERM	timeouts, systemd
255	out of range or -1	buggy exit calls

1. The kernel encoding is defined by POSIX macros: `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WCOREDUMP`, `WIFSTOPPED`.
2. `WEXITSTATUS` is only valid when `WIFEXITED` is true. Reading it otherwise returns rubbish, and this is a real bug in real code.
3. The 64 to 78 range comes from `sysexits.h`, introduced in 4.3BSD. `EX_USAGE` is 64, `EX_DATAERR` 65, `EX_NOINPUT` 66, `EX_UNAVAILABLE` 69, `EX_SOFTWARE` 70, `EX_CONFIG` 78. It is a convention, widely ignored, but useful if you follow it consistently within one project.
4. `grep` documents its own contract in its manual page, and the exact wording from the real page in this sandbox is worth reading:

EXIT STATUS

Normally the exit status is 0 if a line is selected, 1 if no lines were selected, and 2 if an error occurred.

5. `diff` uses 0 for identical, 1 for different, 2 for trouble. `curl` has about 100 documented codes; 6 is “could not resolve host”, 7 “failed to connect”, 28 “operation timed out”, 56 “failure receiving data”. A real failure captured here returned 56.

6. Continuous integration works exactly like this. A GitHub Actions step fails when the process exits non-zero. Nothing else is inspected.
7. Consequences that bite in practice. A step that ends with a pipeline reports only the last stage, so `run_tests | tee log.txt` reports the status of `tee`, which basically always succeeds.
8. The fix is `set -o pipefail`, or checking `PIPESTATUS`, or restructuring so the important command is last.
9. `set -e` has genuine sharp edges. It does not trigger inside a condition, inside `&&` or `||` chains except the last element, or inside a command whose status is being tested. Experts disagree about whether it is safe.
10. Both sides of that argument: the “always use it” camp says most scripts are short and unhandled failure is worse than surprising exits. The “never use it” camp says the exceptions are too subtle to remember and explicit `if ! cmd; then` checks are honest. The compromise most teams settle on is `set -euo pipefail` plus explicit handling around known-noisy commands.

■ 19.8.6 WORDS – remember these

1. Exit code — the single number a program leaves behind — the 8-bit exit status collected by `wait`.
2. `$?` — the shell variable holding the last one — expands to the exit status of the most recently completed foreground command.
3. Zombie — a finished process whose result nobody collected — a process in state Z, holding only its exit record.
4. `&&` — run the next one only if this worked — the AND list operator, short circuits on non-zero status.
5. `||` — run the next one only if this failed — the OR list operator, short circuits on zero status.
6. `set -e` — stop the script at the first failure — `errexit`, which exits on any untested non-zero status.
7. `sysexits` — an agreed table of failure categories — the 64 to 78 range from the BSD `sysexits.h` header.

19.9 Environment variables

■ 19.9.1 PLAIN – in simple words

1. Every running program carries a small list of name and value pairs.
2. That list is the **environment**. It is just text, copied into the program when it starts.
3. `HOME` says where your home directory is. `PATH` says where to look for commands. `LANG` says which language and character set to use.
4. When a program starts another program, the child gets a copy of the list.
5. It is a copy, not a link. The child can change its own copy and the parent never notices.
6. There is no way for a child to change its parent’s environment. This is a hard rule, and it explains many confusing situations.
7. Inside the shell you can also have plain variables that are not in the environment. They exist only in that shell.
8. To move one into the environment you `export` it. That is the whole difference.
9. Where these get set is a separate mess, and it is why your `PATH` can be right in one window and wrong in another.

■ 19.9.2 PLAIN – a picture in your head

1. Think of a person leaving home for work with a small notebook in their pocket.
2. The notebook lists their address, their language, and the list of shops they are allowed to visit.
3. When they hire an assistant, they photocopy the notebook and hand the copy over.
4. The assistant can scribble in their copy. The original is untouched.
5. When the assistant hires their own assistant, another photocopy is made, including any scribbles.
6. So changes flow downwards only, and only to people hired after the change.
7. If you edit the master notebook at home, everybody already out working still has the old copy.
8. That is exactly why changing a setting file does not affect terminal windows you already have open.

Where this comparison breaks: a notebook can hold anything, but the environment holds only text, with a size limit, and no structure. There are no lists, no numbers and no nesting. Everything is a string, and any structure you think you see, like the colons in `PATH`, is a convention agreed by the programs reading it.

■ 19.9.3 PLAIN – a worked example

1. Real run showing the difference between a shell variable and an exported one:

```
$ MYVAR=hello
$ echo "in this shell: $MYVAR"
in this shell: hello
$ bash -c 'echo child sees: [$MYVAR]'
child sees: []
$ export MYVAR
$ bash -c 'echo child sees: [$MYVAR]'
child sees: [hello]
```

2. Before export, the child saw nothing. After export, it saw the value. Nothing else changed.
3. Now `PATH`. Here is the real one from this sandbox, shortened:

```
$ echo $PATH
/home/claude/.npm-global/bin:/root/.local/bin:/root/.cargo/bin:
/usr/local/go/bin:/opt/node22/bin:/usr/local/sbin:/usr/local/bin:
/usr/sbin:/usr/bin:/sbin:/bin
```

4. It is one string. Directories separated by colons. Searched strictly left to right, first match wins.
5. Find out which one wins:

```
$ which ls
/usr/bin/ls
$ type ls
ls is /usr/bin/ls
$ command -v grep
/usr/bin/grep
```

6. Order matters enormously. If `/usr/local/bin` comes before `/usr/bin` and both contain `python3`, you get the one in `/usr/local/bin`.
7. This is the whole explanation for “it works in my terminal but not in the cron job”. Different `PATH`, different program.

■ 19.9.4 PLAIN – what is really happening inside

1. The environment is passed to `execve` as an array of strings, each shaped `NAME=value`, ending with a null pointer.
2. The kernel copies that array onto the new program's stack. The C library makes it available as the global `environ`.
3. Nothing validates it. Any string with an `=` is accepted.
4. When you type a command with no slash in it, the shell splits `PATH` on colons and tries each directory in turn.
5. For each, it builds a full path and tries to execute it. First success wins. If none work, you get 127.
6. An empty entry in `PATH`, such as a leading colon or two colons together, means the current directory. That is a security hazard and should be avoided.
7. The current directory is not in `PATH` by default on any sane system. That is deliberate.
8. If it were, someone could leave a file called `ls` in a shared directory, and you would run it by accident.
9. So to run a program in the directory you are standing in, you must say `./program`. The slash tells the shell not to search at all.
10. Searching `PATH` for every command would be slow, so shells remember. This is called **hashing**.

```
$ hash -r          # forget everything
$ ls > /dev/null
$ grep --version > /dev/null
$ hash
hits    command
  1     /usr/bin/grep
  1     /usr/bin/ls
```

11. The cache is why installing a new program sometimes does not take effect until you run `hash -r` or open a new shell. `bash` and `zsh` both do this.

■ 19.9.5 TECHNICAL – the engineer's version

Variable	Holds	Typical value
<code>PATH</code>	command search list	<code>/usr/local/bin:/usr/bin</code>
<code>HOME</code>	your home directory	<code>/Users/name, /home/name</code>
<code>PWD</code>	current directory	maintained by the shell
<code>OLDPWD</code>	previous directory	used by <code>cd -</code>
<code>SHELL</code>	your login shell	<code>/bin/zsh</code>
<code>USER</code>	your login name	from the <code>passwd</code> entry
<code>TERM</code>	terminal capability name	<code>xterm-256color</code>
<code>LANG</code>	locale for everything	<code>en_US.UTF-8</code>
<code>EDITOR</code>	preferred text editor	<code>vim, nano, code -w</code>
<code>TMPDIR</code>	scratch directory	<code>/tmp</code> , or per-user
<code>LD_LIBRARY_PATH</code>	extra library dirs	avoid unless forced

1. `PWD` is maintained by the shell, not the kernel. The kernel truth is `getcwd`. They can disagree when symbolic links are involved, which is what `cd -P` and `pwd -P` exist to resolve.
2. `SHELL` records your login shell from the password database. It does not tell you which shell is currently running. To find that, check `$0`, or `$BASH_VERSION` and `$ZSH_VERSION`.
3. `LANG` and the `LC_*` family change program behaviour in ways people do not expect. Sort order is the classic case. Real measurement from this sandbox:

```
$ printf 'b\nA\na\nB\n' > letters.txt
$ LC_ALL=C sort letters.txt          -> A B a b
$ LC_ALL=en_US.UTF-8 sort letters.txt -> a A b B
```

4. That is the same data, the same command, and a different answer. Scripts that must be reproducible set `LC_ALL=C` explicitly.
5. Startup file order is the part everyone gets wrong. It depends on two independent questions: is this a login shell, and is it interactive.

Shell	Login shell reads	Interactive non-login
bash	/etc/profile, then the first of ~/.bash_profile, ~/.bash_login, ~/.profile	~/.bashrc only
zsh	/etc/zprofile, ~/.zprofile, then /etc/zshrc, ~/.zshrc, /etc/zlogin, ~/.zlogin	/etc/zshrc, ~/.zshrc
sh	/etc/profile, ~/.profile	ENV file if set

6. bash reads only one of ~/.bash_profile, ~/.bash_login, ~/.profile, in that order, and stops at the first that exists. If you have both a .bash_profile and a .profile, the .profile is silently ignored.
7. zsh always reads ~/.zshrc for interactive shells, login or not. That is why zsh setup is simpler and why macOS advice usually says “put it in .zshrc”.
8. This explains the classic complaint. An SSH login is a login shell. A new tab in Terminal.app on macOS is also a login shell, by Apple’s choice, which differs from most Linux terminal emulators where a new tab is an interactive non-login shell.
9. So a PATH line added to ~/.bashrc on a Linux server works in new tabs but not over `ssh host` command, which is non-interactive and reads neither.
10. Real check of which files exist, from this sandbox:

```
$ for f in /etc/profile ~/.bash_profile ~/.bashrc \
>         ~/.profile ~/.zshrc
> do
>   [ -e $f ] && s=EXISTS || s=missing
>   printf '%-20s %s\n' "$f" "$s"
> done
/etc/profile           EXISTS
/root/.bash_profile    missing
/root/.bashrc          EXISTS
/root/.profile         EXISTS
/root/.zshrc           EXISTS
```

11. GUI applications on macOS do not read your shell startup files at all. They inherit the environment from `launchd`. This is why a program launched from the Dock cannot find a tool that works fine in Terminal.
12. Environment size is limited. On Linux the combined size of arguments and environment is capped by `MAX_ARG_STRLEN` at 128 KiB per string, and the total by the stack limit, typically a quarter of `ulimit -s`. Exceeding it gives `E2BIG`, seen as “Argument list too long”.
13. Never put secrets in environment variables on a shared machine. On Linux `/proc/PID/environ` is readable by the owner, and process listings can leak command-line arguments to everyone.

■ 19.9.6 WORDS – remember these

1. Environment — the list of settings a program inherits — a null-terminated array of NAME=value strings passed through `execve`.
2. Export — move a shell variable into the environment — mark it for inclusion in the child’s environment on the next `exec`.
3. PATH — where the shell looks for commands — a colon-separated directory list, searched left to right, first match wins.
4. Hashing — remembering where a command was found — the shell’s command location cache, cleared with `hash -r`.
5. Login shell — the shell started when you sign in — a shell whose `argv[0]` begins with `-`, reading the profile files.
6. Locale — language and formatting rules — the `LANG` and `LC_*` variables controlling collation, case, dates and messages.

19.10 Configuration on each system

■ 19.10.1 PLAIN – in simple words

1. Every operating system needs somewhere to keep settings.
2. Windows keeps almost all of them in one giant database called the **registry**.
3. macOS and Linux keep them in ordinary files scattered around the disk.
4. The Windows way is one place, one format, one tool. It is fast to read and hard to inspect by eye.
5. The UNIX way is many places, many formats, and any text editor works.
6. On Linux, system-wide settings live in `/etc`, and your personal settings live in files in your home directory whose names begin with a dot.
7. A leading dot means “hidden” by convention. `ls` will not show it unless you ask with `-a`.
8. These are called **dotfiles**, and people keep them in version control and copy them between machines.
9. macOS does both. It has UNIX dotfiles and `/etc` underneath, and Apple’s own settings system on top, using files called property lists.
10. That is why a Mac developer edits `~/ .zshrc` for the shell and never thinks about a registry, even though macOS has a settings database of its own.

■ 19.10.2 PLAIN – a picture in your head

1. Imagine two libraries.
2. The first library has one enormous card catalogue in the entrance hall. Every fact about every book is in a drawer somewhere in it.
3. Finding anything is fast, if you know the drawer. Browsing is impossible. The drawers are labelled in code.
4. If the catalogue is damaged, the whole library stops working, because no book can be located.
5. The second library writes each subject’s notes on a sheet of paper and pins it to the shelf that subject lives on.
6. Finding a note means walking to the right shelf. Slower, but you can read it with your eyes, and you can photocopy one shelf’s notes and carry them elsewhere.
7. If one sheet is torn, only that shelf is affected.
8. Library one is the Windows registry. Library two is `/etc` and dotfiles.

Where this comparison breaks: the registry is far better engineered than a card catalogue. It is transactional, it supports permissions per entry, and it can be changed by policy across ten thousand

machines at once. The scattered-files approach has no transactions at all: a half-written config file is simply a broken config file.

■ 19.10.3 PLAIN – a worked example

1. Say you want to change the shell prompt colour and the editor a tool opens.
2. On Linux or macOS, you edit one text file:

```
# in ~/.zshrc or ~/.bashrc
export EDITOR=vim
export PS1='%n@%m %1~ %# '
```

3. You then run `source ~/.zshrc` to apply it to the current shell, or open a new window.
4. You can email that file to a colleague. It is 20 lines of text.
5. On macOS, an application setting is different. To make Finder show hidden files:

```
defaults write com.apple.finder AppleShowAllFiles -bool true
killall Finder
```

6. That wrote into a property list file at `~/Library/Preferences/com.apple.finder.plist`.
7. On Windows, the same class of change is a registry edit. In `regedit` you would navigate to a path such as:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion
\Explorer\Advanced
```

8. Then set the value `Hidden` to the number 1.
9. Or from PowerShell, without the graphical tool:

```
Set-ItemProperty -Path 'HKCU:\Software\Microsoft\Windows\
CurrentVersion\Explorer\Advanced' -Name Hidden -Value 1
```

10. Three systems, one idea, three completely different mechanisms.

■ 19.10.4 PLAIN – what is really happening inside

1. The registry is a tree, like a filesystem, but stored in a few binary files.
2. The top-level branches are called **hives**. The two you meet are:
3. `HKEY_LOCAL_MACHINE`, shortened to `HKLM`. Settings for the whole computer. Changing it needs administrator rights.
4. `HKEY_CURRENT_USER`, shortened to `HKCU`. Settings for the person signed in right now. You can change your own.
5. Inside a hive are **keys**, which are like folders, and **values**, which are like files.
6. A value has a name, a type and data. Types include string, expandable string, 32-bit number, 64-bit number, binary blob and multi-string.
7. `regedit.exe` is the graphical browser. It is a plain tree view with the values on the right.
8. Why it exists: before it, Windows kept settings in hundreds of `.ini` text files. There was no locking, no types, no permissions and no network management.
9. Why it became a problem: it is a single shared namespace with no ownership rules. Any program can write anywhere it has rights to.
10. Uninstalling a program often leaves its keys behind. Over years the registry accumulates dead entries, and a whole industry of “registry cleaner” products grew up around this, most of them useless or harmful.
11. On macOS, the equivalent store is thousands of separate property list files, most in `~/Library/Preferences` and `/Library/Preferences`.

12. Each is named after a reverse domain identifier, such as `com.apple.finder.plist`, so applications cannot collide by accident.
13. They are usually stored in a compact binary format, not readable text. That is a storage choice, not a philosophy choice.
14. On Linux, there is no central store at all. Each program picks its own file and its own format, guided loosely by an agreed set of directories.

■ 19.10.5 TECHNICAL – the engineer’s version

System	Store	Tool to read
Windows	registry hives	regedit, reg.exe
macOS	plist files	defaults, plutil
Linux	/etc and dotfiles	any text editor
Modern	XDG directories	any text editor

1. Registry history. It first appeared in Windows 3.1, released 1992, as a single file `REG.DAT` limited to 64 KB, holding only OLE registration and file associations.
2. Windows NT 3.1 in 1993 and Windows 95 in 1995 expanded it into the full hive structure with `HKEY_LOCAL_MACHINE` and `HKEY_CURRENT_USER`, and moved registry access into kernel mode.
3. On modern Windows the hive files live in `C:\Windows\System32\config` for machine hives, and `NTUSER.DAT` in each user profile for HKCU. They are not editable as text.
4. The five root keys are HKLM, HKCU, HKCR (classes, a merged view), HKU (all loaded user hives) and HKCC (current hardware profile). Only HKLM and HKU are truly stored; the others are views.
5. Value types are named `REG_SZ`, `REG_EXPAND_SZ`, `REG_DWORD`, `REG_QWORD`, `REG_BINARY`, `REG_MULTI_SZ`.
6. Windows also has Group Policy, which writes into the registry under `Software\Policies` and is enforced centrally. That central-management ability is the strongest argument for the registry’s existence.
7. macOS plists conform to Apple’s property list format. The XML variant has a DOCTYPE from Apple; the binary variant starts with the eight bytes `bplist00`.
8. `defaults read com.apple.finder` prints a domain. `defaults write` sets a value. `plutil -convert xml1 file.plist` makes a binary file readable, and `plutil -lint` validates one.
9. There is a caching daemon, `cfprefsd`. Editing a plist file by hand while an application is running can be overwritten by the cache. Use `defaults` or quit the application first. This is a real and frequent trap.
10. Linux and modern macOS command-line tools increasingly follow the XDG Base Directory Specification, version 0.8, published 8 May 2021 by freedesktop.org.

XDG variable	Default	For
<code>XDG_CONFIG_HOME</code>	<code>\$HOME/.config</code>	settings
<code>XDG_DATA_HOME</code>	<code>\$HOME/.local/share</code>	app data
<code>XDG_STATE_HOME</code>	<code>\$HOME/.local/state</code>	logs, history
<code>XDG_CACHE_HOME</code>	<code>\$HOME/.cache</code>	rebuildable data
<code>XDG_CONFIG_DIRS</code>	<code>/etc/xdg</code>	system settings
<code>XDG_DATA_DIRS</code>	<code>/usr/local/share/</code>	system data

11. The point of XDG is that a home directory used to fill with dozens of top-level dotfiles. Now well-behaved tools use `~/.config/toolname/`.
12. Adoption is partial. `git` supports `~/.config/git/config`. `ssh` still insists on `~/.ssh`. `bash` still uses `~/.bashrc`. This is a convention with incomplete uptake, not a standard anyone enforces.
13. System configuration on Linux lives in `/etc`, a hierarchy defined loosely by the Filesystem Hierarchy Standard, current version 3.0 from 2015. Examples: `/etc/passwd`, `/etc/hosts`, `/etc/ssh/ssh_config`, `/etc/fstab`, `/etc/systemd/system`.
14. Why a macOS or Linux developer never thinks about a registry: everything they touch daily, the shell, the editor, the compiler, `git`, `ssh`, `docker`, reads a text file at a documented path. Nothing they use has a central database, so the concept never comes up.
15. The honest version: macOS does have a registry-like system, and its problems are the same as Windows'. Stale preference domains accumulate, `cfprefsd` caching causes lost edits, and there is no reliable uninstall. Developers avoid it only because their tools are UNIX tools, not Mac applications.

■ 19.10.6 WORDS – remember these

1. Registry — Windows' one big settings database — a hierarchical binary store of keys and typed values, held in hive files.
2. Hive — a top-level branch of the registry — a discrete file-backed subtree such as `HKLM` or a user's `NTUSER.DAT`.
3. `HKLM` and `HKCU` — machine settings and my settings — `HKEY_LOCAL_MACHINE` requires elevation, `HKEY_CURRENT_USER` does not.
4. `plist` — a macOS settings file — an Apple property list, XML or binary, keyed by reverse-domain identifier.
5. Dotfile — a hidden settings file in your home directory — a file whose name starts with `.`, omitted by `ls` unless `-a` is given.
6. XDG base directories — the agreed places for config, data and cache — the freedesktop.org specification defining `XDG_CONFIG_HOME` and friends.
7. `/etc` — where system settings live on UNIX — the machine-wide configuration hierarchy described by the Filesystem Hierarchy Standard.

19.11 The essential commands

■ 19.11.1 PLAIN – in simple words

1. There are thousands of commands. About forty carry almost all daily work.
2. They fall into groups, and learning them by group is far easier than learning them alphabetically.
3. Navigation: where am I, what is here, move somewhere else.
4. Files: create, copy, move, delete.
5. Viewing: show me the contents, the top, the bottom, one page at a time.
6. Searching: find files by name, find text inside files.
7. Text processing: cut columns, replace text, sort, count.
8. Permissions: who is allowed to do what.
9. Processes: what is running, stop it.
10. Disk: how much space, what is using it.
11. Network: is it reachable, fetch it, log in to it.
12. Archives: bundle a folder into one file, unpack it again.
13. Help: what does this command do, where does it live, what did I type before.

■ 19.11.2 PLAIN – a picture in your head

1. Think of a workshop with a pegboard of hand tools.
2. You do not learn a pegboard by reading every label. You learn it by doing three or four jobs.
3. The saw, the hammer and the tape measure get used in every job. Those are `ls`, `cd` and `cat`.
4. Some tools look alike but are for different materials. `grep` searches inside files; `find` searches for files. Beginners reach for the wrong one constantly.
5. Some tools are power tools with a manual. `sed` and `awk` are those. You will use ten percent of them forever, and that is fine.
6. Every tool has flags, which are like the settings on a drill. Two or three settings per tool are worth memorizing. The rest you look up.

Where this comparison breaks: hand tools are shaped so you cannot use them wrongly. Command flags are one letter long and unforgiving. `rm -rf /` and `rm -rf ./` differ by one character and by everything else.

■ 19.11.3 PLAIN – a worked example

1. Navigation and files, run for real in this sandbox:

```
$ pwd
/tmp/shdemo
$ ls -l
-rw-r--r-- 1 root root 610 Aug 13 01:55 access.log
prw-r--r-- 1 root root  0 Aug 13 01:55 f
-rw-r--r-- 1 root root  0 Aug 13 01:56 note1.txt
```

2. Read a long listing left to right. The first character is the type: `-` for a regular file, `d` for a directory, `l` for a symbolic link, `p` for a named pipe, which is what that `f` is.
3. The next nine characters are permissions in three groups of three: owner, group, everyone else. `r` read, `w` write, `x` execute.
4. Then the link count, the owner, the group, the size in bytes, the modification time and the name.
5. Permissions changing for real:

```
$ echo 'echo hi' > s.sh
$ ls -l s.sh
-rw-r--r-- 1 root root 8 Aug 13 01:56 s.sh
$ chmod +x s.sh
$ ls -l s.sh
-rwxr-xr-x 1 root root 8 Aug 13 01:56 s.sh
$ chmod 640 s.sh
$ stat -c '%a %A %n' s.sh
640 -rw-r----- s.sh
```

6. The number form is three octal digits. Read 4 for read, 2 for write, 1 for execute, added together. So 6 is read plus write, 4 is read only, 0 is nothing. 640 means owner read and write, group read, others nothing.
7. Archives for real:

```
$ tar -czf proj.tar.gz proj
$ tar -tzf proj.tar.gz
proj/
proj/src/
proj/src/main.py
proj/README.md
$ tar -xzf proj.tar.gz -C out
```

8. Remember the letters: c create, t list, x extract, z gzip, f the filename follows, v be chatty. Always put f last, because the filename comes straight after it.
9. Networking for real, using the reader's own recorded session on their home connection in India:

```
$ dig +short github.com
20.207.73.82
$ curl -v https://github.com
* Trying 20.207.73.82:443...
... times out after 15 seconds, no response at all
```

10. That address is in a Microsoft-owned range, because GitHub is owned by Microsoft, acquired 2018. The name resolved fine. The connection did not complete. Those are two different failures and the commands separate them.
11. A trace from the same session showed the path leaving the home router at 192.168.0.1, crossing private ISP addresses, reaching Microsoft's network at hop 7, and then falling silent.

■ 19.11.4 PLAIN – what is really happening inside

1. Almost every one of these commands is a small program in /usr/bin or /bin. The shell finds it in PATH and runs it.
2. Most read from standard input when given no filename, which is why they all work in pipes without any special support.
3. Most write results to standard output and complaints to standard error, so redirection works uniformly.
4. Most return 0 on success and non-zero on failure, so && chains work.
5. These four properties are the entire reason the toolkit composes. Nothing else is shared between them.
6. cd is the exception. It is a shell builtin, because changing directory must affect the shell itself.
7. ls sorts alphabetically because it chooses to, not because directories are sorted. On disk, directory entries are in whatever order the filesystem likes.
8. rm does not erase data. It removes a name. If another name points at the same data, the data survives. Only when the last name goes and no process has the file open is the space freed.
9. mv within one filesystem does not move data either. It just changes which directory holds the name. Across filesystems it must copy and then delete, which is why it is slow there.
10. kill does not necessarily kill. It sends a signal. The default is SIGTERM, which is a polite request the program may handle or ignore. kill -9 sends SIGKILL, which the program cannot refuse.

■ 19.11.5 TECHNICAL – the engineer's version

1. Navigation and inspection.

Command	Plain job	Flags worth knowing
pwd	print current path	-P resolve symlinks
cd	change directory	- go back, no flag
ls	list directory	-l -a -h -t -r -S
tree	show nested structure	-L depth, -a

2. Real note: ls -lt sorts newest first, ls -ltr reverses so newest is at the bottom, which is what you want on a long log directory.
3. Files.

Command	Plain job	Flags worth knowing
mkdir	make a directory	-p make parents
touch	create or update time	-t set a time
cp	copy	-r -a -i -v
mv	move or rename	-i -n -v
rm	delete	-r -f -i
ln	make another name	-s symbolic link

4. `cp -a` means archive: recursive, preserve permissions, times and links. `rm -i` asks before each delete and is worth aliasing on shared machines.
5. There is no undelete. `rm` is final. The only protection is backups and care.
6. Viewing.

Command	Plain job	Flags worth knowing
cat	print whole file	-n number lines
less	page through a file	-N -S, / to search
head	first lines	-n N, -c bytes
tail	last lines	-n N, -f follow
wc	count lines and words	-l -w -c

7. `tail -f logfile` is the standard way to watch a log grow. `tail -F` also survives the file being rotated and recreated.
8. Inside `less`: /text searches forward, n next match, G end, g start, q quit. It is also the default pager for `man`.
9. Searching.

Command	Plain job	Flags worth knowing
grep	find text in files	-i -n -r -v -c -E
find	find files by attribute	-name -type -mtime
which	show which file will run	-a show all matches
locate	fast name search	needs an index

10. Real `grep` and `find` runs from this sandbox:

```
$ grep -c "500" access.log
3
$ grep -n "404" access.log
3:10.0.0.7 ... "GET /missing HTTP/1.1" 404 152
$ grep -v "200" access.log | wc -l
4
$ find /home/claude/book/chapters -name 'ch1*.md' -size +100k \
  -printf '%s %p\n' | sort -rn | head -3
131233 /home/claude/book/chapters/ch14.md
124198 /home/claude/book/chapters/ch17.md
123297 /home/claude/book/chapters/ch16.md
```

11. `find` syntax is unusual: path first, then tests, then actions. `-exec cmd {} \;` runs once per file; `-exec cmd {} +` batches them and is much faster. `-print0` with `xargs -0` handles filenames with spaces.

12. Text processing.

Command	Plain job	Flags worth knowing
cut	take columns	-d delim, -f fields
sort	order lines	-n -r -u -k -t
uniq	collapse adjacent dups	-c -d -u
tr	swap or delete chars	-d delete, -s squash
sed	edit a stream	-n, s///, -i
awk	field-aware processing	-F, patterns, END

13. Real runs:

```
$ cut -d: -f1,7 /etc/passwd | head -3
root:/bin/bash
daemon:/usr/sbin/nologin
bin:/usr/sbin/nologin
$ printf '10\n9\n100\n2\n' | sort      -> 10 100 2 9
$ printf '10\n9\n100\n2\n' | sort -n  -> 2 9 10 100
$ echo "Hello World" | tr 'a-z' 'A-Z'
HELLO WORLD
$ sed 's/GET/FETCH/' access.log | head -1
10.0.0.7 ... "FETCH /index.html HTTP/1.1" 200 5120
$ awk '{bytes += $10} END {print "total:", bytes, "lines:", NR}' \
    access.log
total: 16594 lines: 8
$ awk '$9 == 500 {print $1}' access.log
10.0.0.12
10.0.0.7
10.0.0.12
```

14. Note sort without -n puts 100 before 2, because it is comparing text. That single default has produced more wrong reports than any other flag in UNIX.
15. sed -i edits in place. GNU sed takes -i alone; BSD and macOS sed require an argument, so sed -i '' 's/a/b/' f on macOS and sed -i 's/a/b/' f on Linux. This difference breaks scripts crossing the two systems constantly.
16. Permissions and ownership.

Command	Plain job	Flags worth knowing
chmod	change permissions	-R, +x, octal 644
chown	change owner and group	-R, user:group
umask	default for new files	022 typical
id	who am I, which groups	-u -g -G
sudo	run as another user	-u, -i, -l

17. Real: umask printed 0022 here, meaning new files get 644 and new directories 755, because the mask removes write from group and others.
18. chown needs root for changing the owner. Changing only the group is allowed if you belong to the target group.
19. Processes.

Command	Plain job	Flags worth knowing
ps	snapshot of processes	aux, -ef, -o fmt
top	live process view	-b -n batch mode
kill	send a signal	-TERM -KILL -HUP
kill	signal by name	-f match full line
jobs	this shell's jobs	%l refers to job l
nohup	survive terminal close	with &

20. Real, from this sandbox:

```
$ ps aux --sort=-%mem | head -3
USER  PID  %CPU %MEM    VSZ   RSS  STAT  TIME  COMMAND
root  462   3.8   9.0 6045508 746736 Rsl    3:46  claude
root  473   0.3   0.4 1800580 33860  Sl     0:21  environment-manager
$ ps -o pid,ppid,stat,etime,comm -p 1
  PID  PPID  STAT      ELAPSED  COMMAND
   1     0  Sll       01:38:18  process_api
```

21. `ps aux` is BSD-style, `ps -ef` is System V style. Both work on Linux and macOS. RSS is resident memory in kilobytes, the number that actually matters. VSZ is address space reserved and is usually meaningless.

22. Escalation order for stopping something: `kill PID` first, wait a few seconds, then `kill -9 PID`. Going straight to `-9` skips the program's chance to flush data and clean up.

23. Disk.

Command	Plain job	Flags worth knowing
df	free space per mount	-h -i inodes
du	space used by a tree	-sh, -dl, -h
ncdu	interactive disk usage	not always installed

24. Real:

```
$ df -h | head -4
Filesystem      Size  Used Avail Use% Mounted on
/dev/vda         252G   12G   30G   29% /
/dev/vdc         327M   295M   26M   93% /opt/claude-code
$ du -sh /home/claude/book/chapters
1.6M  /home/claude/book/chapters
```

25. If `df` shows space free but writes still fail, check `df -i`. You may have run out of inodes, which are the records that name files, not the bytes.

26. Network.

Command	Plain job	Flags worth knowing
curl	fetch a URL	-I -L -o -s -v
wget	download a file	-c continue, -O
ssh	log in to another host	-p port, -i key, -v
scp	copy files over ssh	-r, -P port
ping	is the host answering	-c count
dig	ask DNS a question	+short, @server

Command	Plain job	Flags worth knowing
tracert	show the path there	-m max hops
netstat	sockets and listeners	-tlnp
ss	modern netstat	-tlnp
lsf	what has this file open	-i, -p PID

27. Real runs from this sandbox:

```
$ dig +short github.com
140.82.112.4
$ dig @1.1.1.1 +short pypi.org
151.101.64.223
151.101.192.223
$ curl -s -o /dev/null \
  -w 'code=%{http_code} time=%{time_total}s ip=%{remote_ip}\n' \
  pypi.org
code=301 time=0.039756s ip=151.101.192.223
$ netstat -tlnp | head -3
Proto Local Address      State  PID/Program name
tcp    127.0.0.1:43279    LISTEN 462/claude
tcp    0.0.0.0:2024      LISTEN -
$ ssh -V
OpenSSH_9.6p1 Ubuntu-3ubuntu13.18, OpenSSL 3.0.13 30 Jan 2024
```

28. netstat is deprecated on Linux in favour of ss from the iproute2 package, but it is still present on macOS and on many servers. Learn both spellings of the same idea.
29. ping needs ICMP to be allowed. Many cloud hosts drop ICMP by policy, so a failed ping proves nothing on its own. Prove reachability with a TCP connection instead: `curl -v` or `nc -vz host port`.
30. Archives and transfer.

Command	Plain job	Flags worth knowing
tar	bundle a directory	-czf -xzf -tzf
gzip	compress one file	-d decompress, -9
zip	Windows-friendly bundle	-r recursive
unzip	unpack a zip	-l list, -d dir
rsync	sync trees efficiently	-av -delete -n

31. tar bundles then compresses the whole bundle, so it compresses better but cannot extract one file without reading the stream. zip compresses each file separately, so it is worse at compression but supports random access. Real measurement here: the same tiny tree gave 213 bytes as `.tar.gz` and 640 bytes as `.zip`.
32. Help and history.

Command	Plain job	Flags worth knowing
man	read the manual	-k search, section
which	path of the command	-a all matches
type	what kind of thing is it	-a all definitions
history	what did I type before	Ctrl-R to search

Command	Plain job	Flags worth knowing
<code>apropos</code>	search manual summaries	same as <code>man -k</code>

33. Real history behaviour, captured in a pseudo-terminal here:

```
demo$ history
  1  echo alpha
  2  ls notel.txt
  3  pwd
  4  history
demo$ !2
ls notel.txt
notel.txt
```

34. `!!` repeats the previous command. `sudo !!` is the standard recovery after forgetting `sudo`. `!$` is the last argument of the previous line.

■ 19.11.6 WORDS – remember these

1. Flag — a short option that changes a command’s behaviour — a single-letter or long option parsed by `getopt`.
2. Recursive — apply to everything underneath — the `-r` or `-R` option, descending the directory tree.
3. Inode — the record that describes a file — the filesystem structure holding permissions, times and block pointers, not the name.
4. Signal — a message sent to a running process — an asynchronous notification such as `SIGTERM 15` or `SIGKILL 9`.
5. Pager — a program that shows text one screen at a time — `less` or `more`, the target of the `PAGER` variable.
6. Symbolic link — a file that points at another path — created with `ln -s`, shown as type `l` by `ls -l`.

19.12 Globbing, quoting and expansion

■ 19.12.1 PLAIN – in simple words

1. The shell rewrites your line before any program sees it.
2. By the time `ls *.txt` reaches `ls`, the star is gone. `ls` receives a list of real filenames.
3. This surprises people. `ls` has no wildcard support at all. It never needed any.
4. The rewriting has several kinds, and they happen in a fixed order.
5. `~` becomes your home directory.
6. `{a,b}` becomes two separate words.
7. `$NAME` becomes the value of a variable.
8. `$(command)` becomes whatever that command printed.
9. `*` and `?` become matching filenames.
10. Quotes switch parts of this off. That is their only job.

■ 19.12.2 PLAIN – a picture in your head

1. Think of a mail room that opens every letter before delivery.
2. Wherever the letter says “the boss”, the clerk writes in the boss’s actual name. Wherever it says “all the branches”, the clerk writes out every branch address in full.
3. The recipient never sees the shorthand. They see a fully spelled-out letter.

4. Quotation marks are an instruction to the clerk: leave this part exactly as written.
5. Single quotes mean “change nothing at all inside here”.
6. Double quotes mean “you may still fill in names, but do not split the text into pieces or expand wildcards”.

Where this comparison breaks: the clerk works in a strict order and cannot go back. Once a variable has been replaced by text containing a space, the shell may split that text into two words, and there is no way to un-split it afterwards. That one-way order is the cause of nearly every quoting bug.

■ 19.12.3 PLAIN – a worked example

1. All real output from this sandbox. The directory holds `note1.txt`, `note2.txt`, `notes.md`, `report.csv` and `my file.txt`.

```
$ echo *.txt
my file.txt note1.txt note2.txt
$ echo note?.txt
note1.txt note2.txt
$ echo note[12].txt
note1.txt note2.txt
$ echo nomatch*.zzz
nomatch*.zzz
$ echo file{1..4}.log
file1.log file2.log file3.log file4.log
$ echo {a,b,c}.txt
a.txt b.txt c.txt
$ echo ~
/root
$ echo "there are $(ls | wc -l) entries"
there are 7 entries
$ echo $((3 + 4 * 2))
11
```

2. Note the fourth one. When a wildcard matches nothing, bash leaves it unchanged. The program then receives a literal star and usually complains. zsh instead reports an error and runs nothing, which is arguably safer.
3. Note that brace expansion produced filenames that do not exist. Braces are pure text generation; they never look at the disk.
4. Quoting, for real:

```
$ NAME=world
$ echo "double: $NAME"
double: world
$ echo 'single: $NAME'
single: $NAME
$ echo backslash: \ $NAME
backslash: $NAME
```

5. Now the spaces problem, which is the one that destroys data:

```
$ f="my file.txt"
$ printf "[%s]\n" $f
[my]
[file.txt]
$ printf "[%s]\n" "$f"
[my file.txt]
$ rm $f
```

```
rm: cannot remove 'my': No such file or directory
rm: cannot remove 'file.txt': No such file or directory
```

- Unquoted, one filename became two arguments. With `rm` in a directory that happened to contain a file called `my`, that command would have deleted the wrong file silently.
- The rule is simple and absolute: put double quotes around every variable expansion unless you have a specific reason not to.

■ 19.12.4 PLAIN – what is really happening inside

- You can watch the rewriting happen. `set -x` makes bash print each line after expansion. Real output:

```
$ bash -xc 'x=5; ls note*.txt
>          echo "x is $x, dir ${basename /tmp/shdemo}"'
+ x=5
+ ls note1.txt note2.txt
++ basename /tmp/shdemo
+ echo 'x is 5, dir shdemo'
```

- Look at line two. `ls note*.txt` became `ls note1.txt note2.txt` before running. The wildcard is gone.
- Look at the double plus. That is a nested expansion: the command substitution ran first, at one level deeper.
- The order the shell uses is fixed and worth memorizing:
- Brace expansion first. Purely textual, no filesystem involved.
- Then tilde expansion.
- Then, all at the same time and left to right: parameter expansion (`$VAR`), command substitution (`$(...)`) and arithmetic expansion (`$(...)`).
- Then word splitting, which cuts the results on spaces, tabs and newlines.
- Then filename expansion, the wildcards.
- Finally quote removal, which strips the quote characters themselves so the program never sees them.
- Two consequences follow directly. Word splitting happens after variable expansion, which is why unquoted variables split.
- And filename expansion happens after word splitting, which is why a variable holding a star can still expand to filenames unless quoted.

■ 19.12.5 TECHNICAL – the engineer's version

Form	Name	Consults disk
<code>{a,b}</code>	brace expansion	no
<code>~user</code>	tilde expansion	passwd file
<code>\$VAR</code>	parameter expansion	no
<code>\$(cmd)</code>	command substitution	runs a program
<code>\$(x+1)</code>	arithmetic expansion	no
<code>* ? []</code>	pathname expansion	yes
<code><(cmd)</code>	process substitution	creates a fifo

- The controlling variable for word splitting is `IFS`, the internal field separator. Its default is space, tab and newline. Setting `IFS=$'\n'` restricts splitting to line boundaries.

2. Glob characters are `*` any string including empty, `?` any single character, `[abc]` one of a set, `[!abc]` or `^abc]` none of a set. Character classes such as `[[:digit:]]` are POSIX.
3. A leading dot is never matched by `*` by default. `shopt -s dotglob` in bash changes that. This is why `rm *` does not remove `.git`.
4. `shopt -s nullglob` makes a non-matching pattern expand to nothing instead of itself. `shopt -s failglob` makes it an error, matching zsh's default.
5. `shopt -s globstar` enables `**` for recursive matching in bash 4 and later. macOS's bundled bash 3.2 does not have it; zsh has it always.
6. Parameter expansion has a large sub-language worth learning: `${VAR:-default}` use a default, `${VAR:?message}` fail if unset, `${VAR#prefix}` and `${VAR%suffix}` strip, `${VAR//old/new}` replace all, `${#VAR}` length, `${VAR:2:3}` substring.
7. Backticks are the old command substitution syntax. `$(...)` is POSIX, nests cleanly, and is what you should write. Backticks are a legacy convention.
8. Single quotes protect everything, including backslashes. There is no way to put a single quote inside single quotes; you must close, escape, and reopen: `'it\'\'s'`.
9. Double quotes still allow `$`, backtick and backslash. They suppress word splitting and globbing.
10. `"$@"` expands to each argument as a separate quoted word. `"$*"` joins them into one word. This distinction matters in every wrapper script ever written.
11. For filenames from `find`, use `-print0` with `xargs -0` or `while IFS= read -r -d ''`. Newlines are legal in UNIX filenames, and any line-based loop over filenames is broken in principle.
12. `shellcheck` is a static analyser that catches unquoted expansions and dozens of similar faults. Running it on every script is the cheapest quality improvement available in shell programming.

■ 19.12.6 WORDS – remember these

1. Glob — a wildcard filename pattern — pathname expansion performed by the shell, not by the command.
2. Expansion — the shell rewriting your line before running it — the ordered sequence from brace expansion to quote removal.
3. Word splitting — cutting expanded text into separate arguments — splitting on the characters in IFS, after parameter expansion.
4. IFS — the characters that separate words — the internal field separator, defaulting to space, tab and newline.
5. Command substitution — replacing a command with its output — `$(cmd)`, with trailing newlines removed.
6. Quote removal — the final step that deletes the quote marks — performed after all expansions, so programs never see quotes.

19.13 Shell scripting

■ 19.13.1 PLAIN – in simple words

1. A shell script is a file containing the same commands you would type.
2. Put them in a file, mark the file as runnable, and you can run the whole sequence with one word.
3. The first line should say which program interprets the file. That line starts with `#!` and is called the **shebang**.
4. `#!/usr/bin/env bash` is the usual choice, because it finds bash wherever it is installed.
5. Inside, you get variables, `if` for decisions, `for` and `while` for repetition, and functions for naming a block of steps.

6. Arguments arrive as \$1, \$2, and so on. \$@ is all of them. \$# is how many.
7. `exit N` ends the script with the code N, which the caller can test.
8. One line near the top prevents most disasters: `set -euo pipefail`.
9. It means: stop on the first failure, stop if I use a variable I never set, and notice failures inside pipelines.

■ 19.13.2 PLAIN – a picture in your head

1. Think of a recipe card taped to the kitchen wall.
2. Typing commands is cooking from memory. A script is writing the recipe down so tomorrow's version is identical.
3. Without `set -e`, the recipe says "if the oven fails to heat, carry on anyway", which produces a raw cake presented as finished.
4. With `set -e`, the cook stops and tells you the oven failed.
5. Functions are named sub-recipes: "make the sauce" written once, used three times.
6. Arguments are the parts you leave blank: how many people, which flavour.

Where this comparison breaks: a human cook applies judgement and notices when something looks wrong. A script notices nothing at all. It will pour salt for an hour if you tell it to, and its only safety comes from checks you wrote in advance.

■ 19.13.3 PLAIN – a worked example

1. Here is the real difference `set -euo pipefail` makes. Both scripts were run in this sandbox.

```
#!/usr/bin/env bash
cd /nonexistent-dir
echo "still running, and about to work in $(pwd)"
echo "value is: $UNDEFINED_VAR"
```

2. Real output:

```
./noset.sh: line 2: cd: /nonexistent-dir: No such file or directory
still running, and about to work in /tmp/shdemo
value is:
exit=0
```

3. Read that carefully. The `cd` failed. The script continued in the wrong directory. It used an undefined variable as an empty string. And it reported success.
4. If line 3 had been `rm -rf ./*`, it would have deleted the wrong directory and told the caller everything was fine.
5. Now the same script with the guard:

```
#!/usr/bin/env bash
set -euo pipefail
cd /nonexistent-dir
echo "this line never runs"
```

6. Real output:

```
./withset.sh: line 3: cd: /nonexistent-dir: No such file
or directory
exit=1
```

7. It stopped at the failure and reported failure. That is the whole argument for the line.

■ 19.13.4 PLAIN – what is really happening inside

1. The `#!` line is read by the kernel, not the shell. When you execute a file, the kernel looks at its first two bytes.
2. If they are `#!`, the kernel reads the rest of the line, runs that program, and hands it the script's path as an argument.
3. So `./script.sh` really becomes `/usr/bin/env bash ./script.sh`.
4. `env` then searches `PATH` for `bash` and executes it. That indirection is why the `env` form works on machines where `bash` is not in `/bin`.
5. Without the execute permission bit, the kernel refuses, and you get exit code 126.
6. `set -e` sets the `errexit` shell option. After every simple command the shell checks the status and exits if it is non-zero and untested.
7. `set -u` sets `nounset`. Expanding an unset variable becomes an error instead of an empty string.
8. `set -o pipefail` changes a pipeline's status from "the last command" to "the rightmost command that failed".
9. Functions are not separate processes. They run inside the same shell, so they can change its variables and its directory.
10. A subshell, written `( ... )`, is a forked copy. Changes inside it are thrown away when it ends. That is why `cd` inside `( )` does not move you.

■ 19.13.5 TECHNICAL – the engineer's version

1. Here is a complete, working script, run for real in this sandbox. It reads a web server log and reports error counts, with a meaningful exit code.

```
#!/usr/bin/env bash
set -euo pipefail

usage() {
    echo "usage: $(basename "$0") LOGFILE [MIN_ERRORS]" >&2
    exit 64
}

[ $# -ge 1 ] || usage
logfile=$1
min_errors=${2:-1}

if [ ! -r "$logfile" ]; then
    echo "error: cannot read $logfile" >&2
    exit 66
fi

total=$(wc -l < "$logfile")
echo "file: $logfile"
echo "lines: $total"

echo "status counts:"
awk '{print $9}' "$logfile" | sort | uniq -c | sort -rn

echo "clients with at least $min_errors error(s):"
found=0
while read -r count ip; do
    if [ "$count" -ge "$min_errors" ]; then
        printf ' %-12s %s\n' "$ip" "$count"
        found=$((found + 1))
    fi
done
```



```
done < <(awk '$9 >= 400 {print $1}' "$logfile" \
    | sort | uniq -c | sort -rn)

if [ "$found" -eq 0 ]; then
    echo "  none"
    exit 0
fi
exit 1
```

2. Line by line. `#!/usr/bin/env bash` selects bash via PATH.
3. `set -euo pipefail` turns on the three guards described above.
4. `usage()` defines a function. `>&2` sends its message to standard error, so it is not captured by a caller collecting output. `exit 64` is `EX_USAGE`.
5. `[ $# -ge 1 ] || usage` is an idiom: if there is not at least one argument, run usage. `$#` is the argument count.
6. `min_errors=${2:-1}` uses parameter expansion to default the second argument to 1 when it is absent. Without this, `set -u` would abort.
7. `[ ! -r "$logfile" ]` tests readability. The quotes are essential; a path with a space would otherwise become two arguments and break the test.
8. `wc -l < "$logfile"` uses redirection rather than passing the filename, so `wc` prints only the number without the filename after it.
9. The `while read -r count ip` loop reads two fields per line. `-r` stops backslashes being interpreted, and should always be present.
10. `done < <(...)` is process substitution. The pipeline runs in a separate process and its output is fed to the loop's standard input.
11. This matters: a plain pipeline `| while read` puts the loop in a subshell, so `found` would be lost when the loop ends. Process substitution keeps the loop in the main shell.
12. `$((found + 1))` is arithmetic expansion. No external `expr` is needed.
13. The script exits 0 when nothing was found and 1 when errors were found, which lets a monitoring system use it directly in an `if`.
14. Real runs:

```
$ ./logsum.sh access.log 2
file: access.log
lines: 8
status counts:
    4 200
    3 500
    1 404
clients with at least 2 error(s):
10.0.0.7      2
10.0.0.12     2
exit=1

$ ./logsum.sh
usage: logsum.sh LOGFILE [MIN_ERRORS]
exit=64

$ ./logsum.sh /nope.log
error: cannot read /nope.log
exit=66
```

15. Portability notes. `[ [ ] ]` is a bash and zsh feature with better quoting behaviour; `[ ]` is POSIX and

is really the program test. Process substitution and arrays are bash and zsh only, not sh.

16. `trap 'rm -f "$tmp"' EXIT` is the standard way to clean up on any exit path, including errors. Create temporary files with `mktemp`, never with a fixed name in `/tmp`.
17. Where experts disagree: some hold that anything over roughly 100 lines should be rewritten in Python. Others keep shell scripts of a thousand lines running happily. The practical dividing line most teams use is data structures: the moment you want a list of records with fields, leave shell.

■ 19.13.6 WORDS – remember these

1. Shebang — the first line naming the interpreter — the `#!` magic number read by the kernel's `execve`.
2. `set -e` — stop at the first failure — the `errexit` option, with documented exceptions inside conditions.
3. `set -u` — treat unset variables as errors — the `nounset` option.
4. `pipefail` — a pipeline fails if any stage fails — a bash and zsh option, not in POSIX.
5. Subshell — a forked copy of the shell — created by `( )`, pipelines and command substitution; its variable changes do not escape.
6. Process substitution — treat a command's output as a file — `<(cmd)`, implemented with a FIFO or `/dev/fd`, bash and zsh only.
7. `trap` — run cleanup code when the script ends — a handler registered for a signal or the pseudo-signal `EXIT`.

19.14 Getting comfortable

■ 19.14.1 PLAIN – in simple words

1. Speed at the command line comes from about a dozen habits, not from typing faster.
2. Tab completion: type the first few letters and press Tab. The shell finishes the name.
3. History search: press Ctrl-R and type part of an old command. It appears. Press Enter to run it.
4. Line editing: Ctrl-A jumps to the start of the line, Ctrl-E to the end, Ctrl-U deletes to the start, Ctrl-W deletes the last word.
5. Aliases: give a long command a short name you choose.
6. Sessions that survive: `tmux` keeps your work running when your connection drops.
7. Reading manuals properly: every manual page has the same sections, and you only ever need three of them.

■ 19.14.2 PLAIN – a picture in your head

1. Think of learning a musical instrument.
2. Nobody plays fast by moving their fingers faster. They play fast because common patterns became automatic.
3. Tab completion is like a scale you no longer think about.
4. Ctrl-R is like remembering a phrase you played last week instead of re-inventing it.
5. `tmux` is like leaving your instrument set up on its stand, tuned, so tomorrow you sit down and continue mid-piece.

Where this comparison breaks: an instrument gives immediate feedback when you get it wrong. A shell often gives no feedback at all, so bad habits survive for years. The only cure is deliberately learning the correct form once.

■ 19.14.3 PLAIN – a worked example

1. Keyboard shortcuts worth memorizing. These come from the readline library, used by bash, and zsh has the same set in emacs mode.

Keys	What it does	Keys	What it does
Ctrl-A	start of line	Ctrl-E	end of line
Ctrl-U	delete to line start	Ctrl-K	delete to end
Ctrl-W	delete previous word	Ctrl-Y	paste last delete
Ctrl-R	search history back	Ctrl-G	cancel the search
Ctrl-L	clear the screen	Ctrl-C	cancel this line
Ctrl-D	end of input, log out	Ctrl-Z	suspend the job
Alt-B	back one word	Alt-F	forward one word
Alt-.	last argument of prev	Ctrl-_	undo the edit

2. On macOS, Alt is the Option key, and Terminal.app needs “Use Option as Meta key” enabled in its settings before Alt-B and Alt-F work.
3. Aliases, real output from this sandbox:

```
$ alias ll='ls -lah'
$ alias
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -lah'
alias ls='ls --color=auto'
$ type ll
ll is aliased to `ls -lah`
```

4. Aliases go in ~/.zshrc or ~/.bashrc so they exist in every new shell.
5. An alias is text substitution on the first word only. For anything needing arguments in the middle, write a function instead.

■ 19.14.4 PLAIN – what is really happening inside

1. Tab completion is a shell feature, not a terminal feature. The terminal just sends byte 9.
2. The shell’s line editor intercepts it, looks at what you have typed, and works out the candidates.
3. bash uses the GNU readline library for this. zsh has its own editor, ZLE.
4. Modern completion is programmable. When you type `git che` and press Tab, the shell runs a completion function shipped with git, which knows about branches.
5. History is kept in memory during the session and written to a file when the shell exits. bash uses ~/.bash_history; zsh uses ~/.zsh_history.
6. That end-of-session write is why history from one window can be missing in another. `setopt INC_APPEND_HISTORY` and `setopt SHARE_HISTORY` in zsh, or `shopt -s histappend` with `PROMPT_COMMAND` in bash, fix it.
7. tmux works by putting your shell inside a pseudo-terminal that tmux itself owns, rather than the one your emulator made.
8. When your SSH connection dies, your emulator’s pty is destroyed, but tmux’s is not, because the tmux server process is still running on the far machine.
9. Reconnect, run `tmux attach`, and tmux re-draws its saved screen contents into your new terminal.
10. Without tmux, losing the connection sends SIGHUP to your shell, which normally kills everything it started.

■ 19.14.5 TECHNICAL – the engineer’s version

1. tmux, real version and behaviour from this sandbox:

```
$ tmux -V
tmux 3.4
$ tmux new-session -d -s demo 'sleep 60'
$ tmux ls
demo: 1 windows (created Thu Aug 13 02:00:38 2026)
$ tmux kill-session -t demo
```

2. The commands worth knowing: `tmux new -s name`, `tmux ls`, `tmux attach -t name`, `tmux kill-session -t name`. Inside, the prefix key is Ctrl-B by default: Ctrl-B d detach, Ctrl-B c new window, Ctrl-B " split horizontally, Ctrl-B % split vertically, Ctrl-B [scrollback mode.
3. GNU Screen is the older equivalent, first released 1987, prefix Ctrl-A. tmux, first released 2007 by Nicholas Marriott, is the more actively developed choice today.
4. `nohup cmd &` and `disown` are lighter alternatives: they detach a single job from the terminal’s hangup signal but give you no way to reattach and see it.
5. Manual page structure is standardized. The sections, in the order they always appear, and confirmed against the real `grep` page in this sandbox:

GREP(1)	User Commands	GREP(1)
NAME		
SYNOPSIS		
DESCRIPTION		
OPTIONS		
REGULAR EXPRESSIONS		
EXIT STATUS		
ENVIRONMENT		
NOTES		
COPYRIGHT		
BUGS		
EXAMPLE		
SEE ALSO		

6. How to read one properly: read NAME to confirm it is the right tool, read SYNOPSIS to learn the argument shape, then jump straight to EXAMPLES at the bottom. Read DESCRIPTION only when the examples are not enough.
7. Inside less, which pages the manual: /word searches, n repeats, q quits, G goes to the end where the examples are.
8. Manual sections are numbered. 1 user commands, 2 system calls, 3 library functions, 4 devices, 5 file formats, 7 miscellaneous and overviews, 8 administration commands.
9. That numbering matters. `man 1 printf` is the command; `man 3 printf` is the C function. `man 5 passwd` is the file format; `man 1 passwd` is the tool.
10. `man -k word` or `apropos word` searches all the NAME lines when you do not know the command’s name.
11. GNU tools often have fuller documentation in `info` rather than `man`. `curl --help all` and `git help -a` are other real sources.
12. `tldr`, a community project, gives example-first summaries and is worth installing when a manual page is 900 lines long.

■ 19.14.6 WORDS – remember these

1. `readline` — the library giving bash its line editing — GNU `readline`, configured through `~/inputrc`.

2. Tab completion — the shell finishing a word for you — programmable completion driven by shell functions.
3. Reverse search — finding an old command by fragment — Ctrl-R, an incremental backward search through the history list.
4. Alias — a short name for a longer command — first-word text substitution performed before expansion.
5. tmux — keeps sessions alive across disconnection — a terminal multiplexer holding its own ptys in a persistent server process.
6. SIGHUP — the “your terminal has gone” signal — signal 1, sent to the foreground group when the controlling terminal is lost.

19.15 Package managers

■ 19.15.1 PLAIN – in simple words

1. A package manager installs software for you and keeps track of what it installed.
2. It knows which files belong to which program, so it can remove them cleanly.
3. It knows which programs depend on which other programs, and installs those too.
4. It gets the software from a **repository**: a server holding a catalogue and the files.
5. It checks a cryptographic signature, so you can tell the files really came from the people who claim to have made them.
6. Linux distributions each have one. Debian and Ubuntu use apt. Fedora and Red Hat use dnf. Arch uses pacman.
7. macOS has no official one for developer tools, so the community built **Homebrew**.
8. Windows now has winget, built by Microsoft.
9. The pattern is always: update the catalogue, then install by name.

■ 19.15.2 PLAIN – a picture in your head

1. Think of a library with a strict lending system.
2. You ask for one book. The librarian knows that book refers constantly to two others, so brings all three.
3. Every book has a stamp proving it came from a real publisher, not a forgery somebody left on a shelf.
4. When you return the first book, the librarian checks whether anyone else is using the other two before shelving them.
5. Installing software by downloading it yourself is walking into the building and taking a book without telling anyone. Nothing tracks it and nothing removes it later.

Where this comparison breaks: books do not conflict. Software versions do. Two programs may need incompatible versions of the same library, and no librarian can satisfy both from one shelf. That problem is called dependency hell, and it is the reason containers and per-language package managers exist.

■ 19.15.3 PLAIN – a worked example

1. Real apt output from this sandbox, showing what a package manager actually knows:

```
$ apt-cache policy curl
curl:
  Installed: 8.5.0-2ubuntu10.9
  Candidate: 8.5.0-2ubuntu10.11
```

```

Version table:
 8.5.0-2ubuntu10.11 500
      500 archive.ubuntu.com/ubuntu noble-updates/main
*** 8.5.0-2ubuntu10.9 100

$ apt-cache depends curl
curl
  Depends: libc6
  Depends: libcurl4t64
  Depends: zlib1g

$ dpkg -L curl | head -4
/usr
/usr/bin
/usr/bin/curl
/usr/share

```

2. Three separate facts there: which version is installed, which is available, what it needs, and exactly which files it owns.
3. The repositories it trusts are listed in `/etc/apt/sources.list.d/`, and the signing keys in `/etc/apt/trusted.gpg.d/`. In this sandbox those held `ubuntu.sources`, `docker.list`, and the Ubuntu archive keyring files.
4. Equivalent commands across systems:

Task	apt	dnf / pacman
refresh	apt update	dnf check-update
install	apt install curl	dnf install curl
remove	apt remove curl	dnf remove curl
search	apt search curl	dnf search curl
upgrade	apt upgrade	dnf upgrade

5. On Arch the same five are `pacman -Sy`, `pacman -S curl`, `pacman -R curl`, `pacman -Ss curl`, `pacman -Syu`.
6. On macOS: `brew update`, `brew install curl`, `brew uninstall curl`, `brew search curl`, `brew upgrade`.
7. On Windows: `winget search curl`, `winget install curl`, `winget upgrade --all`.

■ 19.15.4 PLAIN – what is really happening inside

1. `apt update` downloads an index file listing every package, its version, its dependencies and a hash of its contents.
2. That index is signed. The manager verifies the signature against keys you already trust, and refuses to proceed if it fails.
3. `apt install X` reads the index and solves a puzzle: pick versions of `X` and everything it needs so that all constraints hold at once.
4. This is genuinely a hard computational problem. Real solvers use SAT solving techniques, and they can fail with a message about held broken packages.
5. It then downloads each chosen package, verifies each hash, and unpacks the files to their recorded locations.
6. It runs the package's own install scripts, and writes down which files were placed, so removal is exact.

7. Homebrew works differently. It downloads pre-built binaries, called bottles, into its own directory tree and symlinks them into place, so it never touches Apple's system files.
8. Now the warning. You will see installation instructions of this shape:


```
| curl -fsSL some-site.example/install.sh | sh
```
9. That downloads a script and runs it immediately, with your permissions, with no chance to read it.
10. There is no signature check, no record of what it installed, and no way to uninstall it cleanly.
11. Worse, a hostile server can send different content to curl than it sends to a browser, so reading the page first proves nothing.
12. The safer form is two steps: download to a file, read it, then run it. That is not paranoia, it is the minimum you would apply to any other code you were about to give full access to your account.

■ 19.15.5 TECHNICAL – the engineer's version

Manager	Year	System	Format
dpkg	1994	Debian	.deb
RPM	1995	Red Hat 2.0	.rpm
APT	1998	Debian	wraps dpkg
pacman	2002	Arch Linux	.pkg.tar
YUM	2002	RPM systems	wraps rpm
Homebrew	2009	macOS	formulae
DNF	2015	Fedora 22	wraps rpm
winget	2020	Windows	manifests

1. Precise history. Ian Murdock created dpkg for Debian in January 1994. Red Hat Linux 2.0 shipped RPM on 20 September 1995. APT 0.0.1 was released by Scott K. Ellis in 1998; APT 1.0 came on 1 April 2014.
2. YUM was created by Seth Vidal and Michael Stenner at Duke University on 7 June 2002. DNF replaced it as Fedora's default in May 2015 with Fedora 22.
3. Judd Vinet created pacman alongside the launch of Arch Linux in March 2002.
4. Max Howell created Homebrew on 21 May 2009. Version 1.0 arrived on 21 September 2016.
5. Microsoft released winget in preview at Build on 19 May 2020, and version 1.0 on 27 May 2021. Chocolatey, the community predecessor, released 0.6.0 on 23 March 2011.
6. Two distinct layers exist and confusing them causes real problems. System package managers own /usr and system libraries. Language package managers, pip, npm, cargo, gem, own their own trees.
7. Installing a Python library with the system manager and another with pip into the same interpreter is a known way to break a machine. Use virtual environments, or pipx, or containers.
8. Signature mechanics. Debian signs the Release file with OpenPGP; the file contains hashes of the index files, which contain hashes of the packages. So one signature check covers everything by chaining hashes.
9. RPM signs individual packages as well. Both models are documented, both are sound, and both fail if you add an untrusted key without thinking.
10. Homebrew does not sign bottles with OpenPGP. It relies on HTTPS transport and on checksums recorded in the formula in a public git repository. That is weaker than Debian's model, and it is a fair criticism.

11. Reproducibility is active work rather than settled fact. The Reproducible Builds project, running since 2013, aims to make a package's binary byte-for-byte derivable from its source. Debian reports high but not complete coverage. Treat any claim of full reproducibility as a target, not a delivered guarantee.
12. Supply chain attacks are the reason all of this matters. Real incidents include the event-stream npm package compromise in 2018 and the xz-utils backdoor discovered in March 2024, which reached Debian and Fedora testing branches before being caught.

■ 19.15.6 WORDS - remember these

1. Package — one installable unit of software — an archive plus metadata describing version, dependencies and file list.
2. Repository — the server holding the catalogue — a signed index plus the package files it describes.
3. Dependency resolution — working out what else must be installed — constraint solving over the version graph, often SAT-based.
4. Signature — proof the files came from who they claim — an OpenPGP signature over the index or the package.
5. Bottle — a pre-built Homebrew package — a relocatable binary archive built by Homebrew's own infrastructure.
6. Supply chain attack — hostile code inserted upstream — compromise of a package, its maintainer or its build system, rather than of your machine.

19.98 Common wrong ideas

1. Wrong: the terminal and the shell are the same program. Right: the terminal emulator draws the window; the shell runs inside it and can be swapped for another without changing the window.
2. Wrong: `ls` understands the `*` wildcard. Right: the shell expands `*` into a list of filenames first; `ls` only ever receives real names.
3. Wrong: `Ctrl-C` sends the letter `C` to the program. Right: the kernel's line discipline sees byte 3 and sends `SIGINT` to the foreground process group; the byte itself is never delivered.
4. Wrong: `2>&1 > file` sends both streams to the file. Right: redirections are applied left to right, so `stderr` is aimed at the screen first and stays there. Write `> file 2>&1`.
5. Wrong: a pipeline runs the first command fully, then the second. Right: all stages run at the same time, connected by a kernel buffer, with the writer blocking when it is full.
6. Wrong: exit code 1 always means something crashed. Right: many tools use 1 for a normal negative answer. `grep` returns 1 when it searched correctly and found nothing.
7. Wrong: a child process can change its parent's environment variables. Right: the child gets a copy at `exec` time. Nothing it does can reach back. That is why `cd` must be a shell builtin.
8. Wrong: editing `~/.zshrc` changes shells that are already open. Right: it is read at startup. Existing shells keep the old values until you `source` the file or open a new window.
9. Wrong: `kill -9` is the normal way to stop a program. Right: `kill` sends `SIGTERM`, which lets the program flush data and clean up. `SIGKILL` cannot be handled and should be the second attempt, not the first.
10. Wrong: piping a downloaded script into a shell is fine because you trust the website. Right: there is no signature, no record and no clean removal, and a server can serve different bytes to `curl` than to a browser.

19.99 Chapter summary in 20 lines

1. A terminal was a physical machine at the end of a wire, with a keyboard and a printer or screen, and almost no intelligence of its own.
2. The Teletype Model 33 of 1963 ran at 110 bits per second, printed on paper, and is why UNIX commands have two-letter names.
3. The DEC VT100 of 1978 popularized the escape sequences, standardized as ECMA-48 and ISO/IEC 6429, that still control your terminal today.
4. The device file is called `tty` because the machine on the other end was a teletype.
5. Terminal, terminal emulator, shell, console and command line are five different things; you use an emulator containing a shell.
6. A pseudo-terminal is a kernel-made fake wire with a master end for the emulator and a slave end for the shell.
7. The line discipline sits between them, doing echo, line editing and buffering, and turning Ctrl-C into SIGINT for the foreground process group.
8. A shell reads a line, expands it, forks, execs and waits. That loop is the whole job.
9. The shell family runs from Thompson 1971 and Bourne 1979 through bash 1989 to zsh 1990, which macOS made the default in Catalina in 2019.
10. PowerShell, from 2006, is a different design entirely: it passes typed objects between commands instead of text.
11. Every process starts with descriptors 0, 1 and 2 for input, output and errors, kept separate so results stay clean.
12. Redirection is open plus `dup2` performed between fork and exec, applied strictly left to right, which is why `2>&1` order matters.
13. A pipe is a kernel buffer, 65536 bytes on Linux, joining two processes that run at the same time, with blocking as natural back pressure.
14. When a reader exits early the writer gets SIGPIPE, signal 13, reported by the shell as exit status 141.
15. Exit codes are 0 for success and 1 to 255 for failure, with 126 not executable, 127 not found, and 128 plus N for death by signal N.
16. `&&` runs the next command only on success, `||` only on failure, and `;` always. Build systems test only these numbers.
17. The environment is a copied list of strings; PATH is searched left to right, the result is cached, and `./` is required for the current directory.
18. Windows keeps settings in one registry of hives, keys and typed values; macOS and Linux use plists, `/etc` and dotfiles, increasingly under the XDG directories.
19. The shell rewrites your line in a fixed order, and word splitting happens after variable expansion, which is why you quote every expansion.
20. `set -euo pipefail` at the top of a script turns silent wrong behaviour into a loud stop, which is the single highest-value line you can write.

Files, Formats, Extensions and the .exe

20.0 What this chapter gives you

1. You will be able to say exactly what a file is, and why there is no such thing as a “picture file” at the level of the disk.
2. You will be able to explain what a file extension really does on Windows, on macOS and on Linux, and why the three answers are different.
3. You will be able to read the first bytes of any file and name the format, using a table of real magic numbers you will have memorized parts of.
4. You will be able to explain why a text file written on Windows shows strange marks on Linux, and fix it.
5. You will be able to open a DOCX, an XLSX, a JAR and an APK with a zip tool, because they are all zip files, and show the layout inside.
6. You will be able to describe what an .exe actually contains, section by section, including why it still says “This program cannot be run in DOS mode”.
7. You will be able to trace, in thirty numbered steps, everything that happens between your double-click and the first line of your program.
8. You will be able to explain why a .py or .sh file also runs, and what the two characters #! do at the level of the kernel.
9. You will be able to identify a file of an unknown format using only `file`, a hex dump, and `strings`.
10. You will be able to name what an installer does, why unsigned software warns you, and what a photo silently tells the world about where you were.

20.1 What a file actually is

■ 20.1.1 PLAIN – in simple words

1. A file is a run of numbered boxes. Each box holds one byte, a number from 0 to 255.
2. That is the whole of it. A file is a sequence of bytes with a length.
3. The file also has a name, so that a human and a program can find it again.
4. Around that sequence, the system keeps a few extra facts: how big it is, when it was last changed, who owns it, who is allowed to read it.
5. Those extra facts are called **metadata**, which just means “data about the data”.
6. Nothing in the bytes says “I am a picture”. Nothing in the metadata says it either, on most systems.
7. A file becomes a picture only when a program chooses to read those bytes using picture rules.
8. If a different program reads the same bytes using sound rules, it will produce noise. Nothing was broken. The bytes never claimed anything.
9. So a file is not a thing with a nature. It is raw material plus an agreement about how to read it.

■ 20.1.2 PLAIN – a picture in your head

1. Think of a long shelf of numbered pigeonholes in a post room.
2. Each pigeonhole holds one small card with a number written on it, from 0 to 255.
3. The shelf has a label on the front: “note.txt”. That is the file name.
4. A clipboard hangs beside the shelf. It records: 26 cards, filled on 13 August at 01:57, owned by the office manager, readable by everyone.
5. That clipboard is the metadata. The pigeonholes are the file contents.
6. Now, one clerk has been told “these cards spell out English letters”. Another clerk has been told “these cards are pixel colours”.
7. Both clerks read the exact same cards. They produce completely different results. Neither clerk is cheating.

Where this comparison breaks: real pigeonholes are physically next to each other, and a real file usually is not. The bytes of one file can be scattered across thousands of separate places on a disk, and the file system quietly stitches them into one continuous-looking sequence for you. The shelf is a fiction the file system maintains. Also, a real clipboard is attached to the shelf, whereas on most file systems the metadata lives somewhere else entirely, in a small record called an inode, and the file name is not even part of it.

■ 20.1.3 PLAIN – a worked example

1. Here is a file with 26 bytes in it, viewed as raw numbers and then as characters.

```
$ stat note.txt
  File: note.txt
  Size: 26          Blocks: 8          IO Block: 4096   regular file
Device: 254,0      Inode: 794747         Links: 1
Access: (0644/-rw-r--r--)  Uid: (0/root)   Gid: (0/root)
Access: 2026-08-13 01:57:42.022877781 +0000
Modify: 2026-08-13 01:57:42.020159532 +0000
Change: 2026-08-13 01:57:42.020159532 +0000
 Birth: 2026-08-13 01:57:42.016211454 +0000
```

2. Size: 26 is the only fact about the contents. Twenty-six bytes. That is all the system knows about what is inside.
3. Inode: 794747 is the number of the little record that holds this metadata. The name note.txt is not stored there. It is stored in the folder.
4. Blocks: 8 means eight 512-byte units are actually reserved on disk, so 4096 bytes are used to store 26. Storage is handed out in whole blocks.
5. There are four timestamps, not one, and they mean different things:

Field	Plain meaning	Changed by
Access	last time it was read	reading the file
Modify	last time contents changed	writing bytes
Change	last time metadata changed	chmod, rename
Birth	when it was created	creation only

6. Notice what is absent from every line above: any statement about what kind of file this is. The word “text” appears nowhere.

■ 20.1.4 PLAIN – what is really happening inside

1. When you save a file, the file system does three separate jobs.
2. Job one: find free blocks on the storage device and write your bytes into them.
3. Job two: create or update a small fixed-size record holding the size, the owner, the permissions, the timestamps, and a list of which blocks hold the data. On Linux file systems this record is the **inode**.
4. Job three: add an entry to a directory. A directory is itself just a file whose contents are a list of pairs: a name, and an inode number.
5. This is why the name is not part of the file. The name lives in the folder, pointing at the file.
6. This is also why one file can have two names. Two directory entries can point at the same inode. That is a **hard link**.
7. Deleting a name does not delete the bytes. It removes one directory entry and reduces a counter. The bytes go only when the counter reaches zero.
8. The Links: 1 line in the output above is exactly that counter.
9. None of these three jobs asks or records what the bytes mean.

■ 20.1.5 TECHNICAL – the engineer's version

1. A regular file in POSIX terms is an unstructured byte stream addressed by offset, with no record structure imposed by the kernel.
2. The metadata returned by the `stat(2)` system call is defined in POSIX.1 and is filled from the file system's own inode structure.
3. On ext4 the inode is 256 bytes by default and holds mode, uid, gid, size, four timestamps with nanosecond precision, link count, and either extents or block pointers.
4. `st_size` is the logical length in bytes. `st_blocks` is the allocated storage in 512-byte units and can be smaller than `st_size` for a sparse file or larger because of block rounding.
5. Directory entries on ext4 are stored in `linux_dirent` structures containing an inode number, a record length, a name length, a one-byte file type hint, and the name. There is no type field beyond regular, directory, symlink, FIFO, socket, block device and character device.
6. NTFS differs. Its Master File Table record holds a set of named attributes, and small files are stored resident inside the MFT record itself. NTFS also supports multiple named data streams per file, which is covered in 20.13.
7. Apple's APFS, shipped in 2017 with macOS 10.13 High Sierra, stores extended attributes in the same B-tree as the file record.
8. Real figures for common file systems:

File system	Max file size	Max name length
ext4	16 TiB	255 bytes
NTFS	8 PiB (Win11)	255 UTF-16 code units
APFS	8 EiB	255 UTF-8 characters
exFAT	128 PiB	255 UTF-16 code units

9. Tools that observe this layer: `stat`, `ls -li`, `df -i`, `debugfs` on ext4, `fsutil` file layout on Windows, `diskutil` and `stat -f` on macOS.

■ 20.1.6 WORDS – remember these

1. File — a named run of bytes — an unstructured byte stream addressed by offset, with an associated metadata record.

2. Metadata — facts about a file that are not its contents — the `struct stat` fields: `mode`, `uid`, `gid`, `size`, `timestamps`, `link count`.
3. Inode — the small record holding a file's facts — an on-disk structure indexed by number, containing everything except the name.
4. Directory — a folder — a file whose contents map names to inode numbers.
5. Hard link — a second name for the same file — an additional directory entry referencing the same inode, incrementing `st_nlink`.
6. Block — the smallest chunk of disk given out — the allocation unit, typically 4096 bytes on `ext4` and `NTFS`.

20.2 File extensions: a convention, not a rule

■ 20.2.1 PLAIN – in simple words

1. The bit after the last dot in a file name is the **extension**: `.txt`, `.jpg`, `.exe`.
2. Say this once and remember it: on almost every system, the extension is part of the name and nothing else.
3. It is a **convention**. Everyone agreed to use it. It is not enforced by the disk, and mostly not enforced by the kernel either.
4. You can rename `holiday.jpg` to `holiday.txt`. Not one byte inside the file changes.
5. What changes is the guess that programs make about the file when they see the name.
6. On Windows, that guess is very strong. The extension decides which program opens the file, full stop.
7. On macOS, the extension matters too, but the system also keeps a richer type idea behind the scenes.
8. On Linux, the usual tools ignore the name and look at the bytes instead.
9. So the same rename does three different things on three systems, and none of them altered the file.

■ 20.2.2 PLAIN – a picture in your head

1. Think of a hospital where every patient wears a wristband with a written label: “diabetes”, “fracture”, “asthma”.
2. The wristband is fast and convenient. A nurse can glance and act.
3. But the wristband is written by hand. It is not the patient. Swap two wristbands and the patients are unchanged.
4. A careful doctor does not trust the wristband. She examines the patient. That takes longer and is always right.
5. Windows is the nurse acting on the wristband. Linux is the doctor examining the patient.
6. macOS is a hospital that reads the wristband first but also keeps a proper file in the records office.

Where this comparison breaks: a real patient has an objective illness that exists whether or not anyone examines it. A file does not have an objective format in the same way. Its format only exists as an agreement between whoever wrote it and whoever reads it. Two different agreements can both fit the same bytes, which is why a valid file can genuinely be two formats at once, a fact attackers use.

■ 20.2.3 PLAIN – a worked example

1. Here is a real one-pixel PNG image. We copy it to a `.txt` name and ask the system what it is.

```
$ cp tiny.png tiny.txt
$ file tiny.txt
```

```
tiny.txt: PNG image data, 1 x 1, 8-bit/color RGB, non-interlaced

$ file --mime-type tiny.txt tiny.png
tiny.txt:  image/png
tiny.png:  image/png

$ sha256sum tiny.png tiny.txt
b1ff9c8ea3a780bad09b346c423d2d0e46815926879b18e841d928376a946640
  tiny.png
b1ff9c8ea3a780bad09b346c423d2d0e46815926879b18e841d928376a946640
  tiny.txt
```

2. The two checksums are identical to the last character. The contents are the same file.
3. The Linux file command did not care about the name at all. It read the bytes and answered PNG both times.
4. On Windows, double-clicking tiny.txt would open Notepad, which would show a screenful of rubbish and then possibly corrupt the file if you saved it.
5. Nothing about the file was ever wrong. The name simply stopped matching the contents, and one operating system trusts names.

■ 20.2.4 PLAIN – what is really happening inside

1. Windows keeps a large lookup table called the **registry**. One part of it, HKEY_CLASSES_ROOT, maps extensions to type identifiers.
2. .txt maps to a name like txtfile. txtfile then has a shell\open\command entry giving the exact command line to run.
3. Double-clicking runs ShellExecute, which walks that chain: extension, then program identifier, then command. The file's contents are never consulted.
4. macOS uses **Uniform Type Identifiers**, reverse-domain strings such as public.png or com.adobe.pdf. Every application declares which identifiers it can open.
5. macOS decides a file's identifier mostly from its extension, then registers the answer in a database called Launch Services.
6. Linux desktops use the freedesktop.org shared MIME database, which combines name patterns with content tests, and content tests usually win.
7. Command-line Linux tools such as file use a library called **libmagic**, which holds thousands of rules of the form "at offset X, expect these bytes, therefore this format".
8. So there are two strategies in the world: trust the name, or read the bytes. Reading the bytes is slower and needs an open file. Trusting the name is instant and needs nothing.
9. That speed difference is the whole reason the extension convention survives.

■ 20.2.5 TECHNICAL – the engineer's version

1. The 8.3 filename with a three-character extension comes from CP/M in 1974 and was carried into MS-DOS in 1981 and into the FAT file system.
2. In FAT, the name and extension were stored as two fixed fields, 8 bytes and 3 bytes, with no dot stored on disk. The dot was purely a display convention.
3. Modern Windows stores the whole name including dots as one UTF-16 string. The extension is defined as the text after the final dot.
4. Windows Explorer hides extensions for registered types by default. This setting is HideFileExt under HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced.
5. That default is a documented security weakness. A file named invoice.pdf.exe displays as invoice.pdf with a PDF-looking icon supplied by the executable's own resource section.

6. The ILOVEYOU worm of May 2000 used exactly this. Its attachment was named LOVE-LETTER-FOR-YOU.TXT.vbs and displayed as a .TXT file. It caused damage estimated in the billions of dollars.
7. Windows also supports right-to-left override characters in file names, which can display exe.txt when the real name ends in .exe. Modern mail gateways strip these.
8. Apple introduced Uniform Type Identifiers in Mac OS X 10.4 Tiger in 2005, and replaced the older four-character type and creator codes inherited from the 1984 Macintosh. A modern Swift framework, UniformTypeIdentifiers, arrived in macOS 11 in 2020.
9. The file command was written by Ian Darwin in 1986 and rewritten and maintained by Christos Zoulas since 1993. Its rule database is compiled into magic.mgc.
10. Behaviour by platform:

Platform	Primary decision	Fallback
Windows	extension plus registry	none
macOS	extension to UTI	type metadata
Linux CLI	content sniffing	extension
Linux desktop	shared MIME db	glob patterns

11. Web browsers are a fourth case. They obey the HTTP Content-Type header, with a defined sniffing algorithm in the WHATWG MIME Sniffing Standard, which servers can disable with X-Content-Type-Options: nosniff.

■ 20.2.6 WORDS – remember these

1. Extension — the letters after the last dot — a filename suffix used as a type hint, not enforced by the file system.
2. Convention — everyone does it this way — behaviour that is customary but not specified or enforced.
3. Registry — the Windows settings database — a hive-structured key-value store; HKEY_CLASSES_ROOT maps extensions to ProgIDs.
4. UTI — Apple’s name for a file kind — Uniform Type Identifier, a reverse-DNS string in a declared inheritance hierarchy.
5. libmagic — the library that guesses a file’s type from its contents — a rules engine matching byte patterns at given offsets.
6. Double extension — a name with two dots used to deceive — an attack relying on HideFileExt and on icon resources embedded in the executable.

20.3 Magic numbers: the real identity of a file

■ 20.3.1 PLAIN – in simple words

1. Most file formats begin with a short, fixed pattern of bytes that says which format it is.
2. That pattern is called a **magic number**, or a signature.
3. It is at the very start, so a program can read a few bytes, decide, and stop if it is wrong.
4. A PNG image always begins with the byte 137 and then the letters P, N, G.
5. A ZIP archive always begins with the letters P and K, the initials of Phil Katz who invented the format.
6. A Linux program always begins with the byte 127 and then the letters E, L, F.

7. A Windows program always begins with the letters M and Z, the initials of Mark Zbikowski who designed the old DOS executable header.
8. This is the closest thing a file has to a real identity, and it is still only a convention, because nothing stops you writing those bytes yourself.
9. But it is a far better guess than the name, because the bytes were written by the program that made the file.

■ 20.3.2 PLAIN – a picture in your head

1. Think of the first line of a formal letter.
2. “Dear Sir” tells you it is a letter. “Once upon a time” tells you it is a story. “Ingredients:” tells you it is a recipe.
3. You do not need to read the rest. Three or four words settle it.
4. Better still, some documents begin with a printed crest or a watermark, which is harder to produce by accident.
5. A magic number is that crest. It is at the top, it is short, and it is very unlikely to appear at the top of something else by chance.

Where this comparison breaks: a letter’s opening is meant for humans and can be argued about. A magic number is exact. Either the bytes are 89 50 4E 47 or they are not. There is no partial match and no interpretation. Also, unlike a crest, a magic number carries no proof of who wrote it. Anyone can type the same four bytes at the start of any file, which is exactly how polyglot files, files valid as two formats at once, are built.

■ 20.3.3 PLAIN – a worked example

1. Here is the first sixteen bytes of a real PNG file we made, shown as hex on the left and as printable characters on the right.

```
$ xxd -l 16 tiny.png
00000000: 8950 4e47 0d0a 1a0a 0000 000d 4948 4452 .PNG.....IHDR
```

2. Read it byte by byte. 89 is the number 137, chosen deliberately to be above 127 so that a program treating the file as text notices immediately.
3. 50 4e 47 is P, N, G in ASCII.
4. 0d 0a is carriage return and line feed. If a broken file transfer converted line endings, this pair would be damaged and the file would fail its own check.
5. 1a is the old DOS end-of-file character. If you type this file on DOS, it stops printing right there instead of spewing binary.
6. 0a is a lone line feed, catching the opposite conversion.
7. Then 00 00 00 0d, which is the number 13 in big-endian order: the length of the first chunk.
8. Then 49 48 44 52, which is IHDR, the image header chunk.
9. The whole eight-byte PNG signature was designed in the mid-1990s to survive damage by naive file transfer software, and it works.

■ 20.3.4 PLAIN – what is really happening inside

1. A program that must identify a file opens it and reads the first block, usually the first 512 or 4096 bytes.
2. It then runs down a list of rules. Each rule says: at this offset, compare these bytes, and if they match, report this type.

- Rules can be nested. Matching RIFF at offset 0 leads to a second test at offset 8, which decides between WAV, AVI and WebP.
- Rules can also be negative. A ZIP that begins with the entry name mimetype is treated as an OpenDocument file rather than a plain archive.
- Some formats have no fixed signature at all. Plain text, CSV and raw MP3 audio are examples, so the tool falls back to statistics: are all bytes printable, are there consistent separators.
- When nothing matches, the answer is data, which honestly means “I do not know”.

■ 20.3.5 TECHNICAL - the engineer's version

- The following signatures were all verified by hex-dumping real files on the machine used to write this chapter, except where the row says otherwise.

Format	First bytes (hex)	As text
PNG	89 50 4E 47 0D 0A 1A 0A	.PNG....
JPEG/JFIF	FF D8 FF E0	
GIF87a	47 49 46 38 37 61	GIF87a
GIF89a	47 49 46 38 39 61	GIF89a
PDF	25 50 44 46 2D	%PDF-
ZIP local	50 4B 03 04	PK..
ZIP central	50 4B 01 02	PK..
ZIP end record	50 4B 05 06	PK..
GZIP	1F 8B 08	...
ELF	7F 45 4C 46	.ELF
PE / MZ	4D 5A	MZ
Mach-O 64-bit	CF FA ED FE	
Java class	CA FE BA BE	
WebAssembly	00 61 73 6D 01 00 00 00	.asm....
RIFF / WAV	52 49 46 46	RIFF
MP3 with ID3	49 44 33	ID3
SQLite 3	53 51 4C 69 74 65 20 66	SQLite f
7-Zip	37 7A BC AF 27 1C	7z..'
BZIP2	42 5A 68	BZh
XZ	FD 37 7A 58 5A 00	.7zXZ.

- Notes on the trickier rows. Mach-O stores its magic as a 32-bit little-endian integer, so MH_MAGIC_64, whose value is 0xFEEDFACF, appears on disk as CF FA ED FE. The 32-bit form is 0xFEEDFACE.
- A macOS universal binary starts with 0xCAFEBAFE in big-endian order, which is byte-for-byte identical to a Java class file. Tools disambiguate by checking the next field: Java stores a version number below 100, a fat Mach-O stores an architecture count. This collision is real and long-standing.
- Mach-O was not produced in this sandbox, which is Linux. Those two rows come from the loader.h header in Apple's open-source XNU sources, not from a local dump.
- Java class files carry a major version after the magic. Version 52 is Java 8, 55 is Java 11, 61 is Java 17, 65 is Java 21. A real dump taken here:

```
$ unzip -p commons-lang3-3.14.0.jar \
  org/apache/commons/lang3/AnnotationUtils\$.class | xxd -l 16
00000000: cafe babe 0000 0034 00bc 0a00 0200 0307
```

0x34 is 52, so this class targets Java 8. 6. WebAssembly's magic is the four bytes `\0asm` followed by a 32-bit version, currently 1. WebAssembly 1.0 became a W3C Recommendation in December 2019. 7. Raw MP3 has no header signature. An MPEG audio frame starts with an 11-bit sync pattern of all ones, so bytes such as `FF FB`. Most real files start with an ID3v2 tag instead, whose first three bytes are `ID3`. 8. The file command in action on the machine used here:

```
$ file sample.wav sample.db sample.wasm hello hello.exe
sample.wav: RIFF (little-endian) data, WAVE audio,
             Microsoft PCM, 16 bit, mono 8000 Hz
sample.db:  SQLite 3.x database, last written using
             SQLite version 3045001, database pages 2
sample.wasm: WebAssembly (wasm) binary module version
             0x1 (MVP)
hello:      ELF 64-bit LSB pie executable, x86-64,
             dynamically linked, interpreter
             /lib64/ld-linux-x86-64.so.2, not stripped
hello.exe:  PE32+ executable (console) x86-64,
             for MS Windows, 19 sections
```

9. Note how much file extracted beyond the type: image dimensions, sample rate, SQLite version, section count, interpreter path. Those come from deeper rules in the same database.
10. The magic database ships as text in `/usr/share/file/magic/` and is compiled into `magic.mgc`. You can add your own rules and point file at them with the `-m` option.

■ 20.3.6 WORDS – remember these

1. Magic number — a fixed pattern at the start of a file naming its format — a signature matched at a specified offset by a type-detection rule.
2. Signature — another word for magic number — same thing; the term “signature” is also used for cryptographic signing, so context matters.
3. Polyglot file — one file that is validly two formats — a file crafted so that two parsers each find a complete valid structure.
4. Endianness — which end of a number comes first — byte order; little-endian stores the least significant byte at the lowest address.
5. Sniffing — guessing a type by reading contents — content-based type detection, as opposed to name-based.
6. libmagic rule — one line in the guessing database — an offset, a type, a test value and a message, optionally nested by indentation depth.

20.4 Text files versus binary files

■ 20.4.1 PLAIN – in simple words

1. People say a file is either “text” or “binary”. That split is useful but it is not a real property of the file.
2. Every file is binary. Every file is bytes. There is no text mode on the disk.
3. What people mean by “text” is: if you show each byte as a character, you get something a human can read.
4. So “text” is a statement about how the bytes happen to look, not about how they are stored.
5. Text files also have one extra habit: they are split into lines.
6. And here the trouble starts, because the world never agreed on how to mark the end of a line.
7. Linux and macOS end a line with one invisible byte. Windows ends it with two. Very old Macs used one different byte.

8. That is why a file written on Windows sometimes shows odd marks when opened on Linux, and why a Linux file sometimes appears as one giant line on Windows.
9. On top of that, there is the question of which byte means which letter, which is called the **encoding**, and that is a second, larger mess.

■ 20.4.2 PLAIN – a picture in your head

1. Imagine an old mechanical typewriter with a paper carriage.
2. To start a new line you must do two separate physical things.
3. First, push the carriage back to the left edge. That is the **carriage return**.
4. Second, roll the paper up by one line. That is the **line feed**.
5. Two actions, two separate control codes, because the machine really did have two separate mechanisms.
6. Windows still sends both codes, faithfully imitating the typewriter.
7. Unix decided one code should mean “do both”, and sends only the line feed.
8. The old Macintosh decided the opposite, and sent only the carriage return.
9. Three reasonable choices, made in different decades, and we live with all three forever.

Where this comparison breaks: on a typewriter you could send carriage return without line feed on purpose, to type over a line and make bold text. Software no longer does that, so the two codes have collapsed into one meaning, “new line”, and only the spelling differs. Also, a typewriter had no concept of encoding. Which key made which shape was fixed by the metal. In a file, the mapping from byte to character is a separate choice on top of the line-ending choice.

■ 20.4.3 PLAIN – a worked example

1. We made the same two lines of text in three different line-ending styles and dumped the raw bytes.

```
$ file unix.txt dos.txt mac9.txt
unix.txt: ASCII text
dos.txt:  ASCII text, with CRLF line terminators
mac9.txt: ASCII text, with CR line terminators

$ od -c unix.txt
l i n e   o n e \n l i n e   t w o \n

$ od -c dos.txt
l i n e   o n e \r \n l i n e   t w o \r \n

$ od -c mac9.txt
l i n e   o n e \r l i n e   t w o \r

$ wc -l unix.txt dos.txt mac9.txt
 2 unix.txt
 2 dos.txt
 0 mac9.txt
```

2. \n is byte 10, the line feed. \r is byte 13, the carriage return.
3. Notice the last line. The tool counted two lines in the Unix file, two in the Windows file, and zero in the old Mac file.
4. It is not broken. `wc -l` counts line feed bytes, and the old Mac file contains none at all.
5. This is exactly how a real file can be “empty” to one tool and full of text to another.
6. When a Windows-made file is read by a Linux tool that only strips \n, the \r stays on the end of every value. A version number 1.2.3 becomes 1.2.3\r, and comparisons silently fail.

■ 20.4.4 PLAIN – what is really happening inside

1. There are three different layers where line endings can be changed, and confusing them causes most of the pain.
2. Layer one: the program that wrote the file chose what to put there.
3. Layer two: a transfer tool may translate on the way. Old FTP had an explicit text mode that rewrote line endings. Git can do the same on checkout.
4. Layer three: the program reading the file may quietly accept both, or may not.
5. Encoding is a separate question: which byte value stands for which character.
6. ASCII fixed 128 characters in the 1960s using values 0 to 127. That covers English and nothing else.
7. Everyone then invented their own meaning for values 128 to 255, producing dozens of incompatible sets, and the same bytes rendered differently in different countries.
8. Unicode solved the character question by giving every character in every script a number, and UTF-8 solved the byte question by encoding those numbers in one to four bytes.
9. UTF-8 was designed so that all the old ASCII values keep their old meaning. An English-only file is identical in ASCII and UTF-8.
10. Some tools put three special bytes at the very start of a UTF-8 file to announce the encoding. That is the **byte order mark**, and it causes its own set of problems.

■ 20.4.5 TECHNICAL – the engineer's version

1. Carriage return is US-ASCII 0x0D, decimal 13. Line feed is 0x0A, decimal 10. Both are in the C0 control set standardized in ASCII, published in 1963 and revised in 1967.
2. The two-code sequence comes from teleprinters such as the Teletype Model 33, introduced in 1963, which needed roughly 200 milliseconds to return the carriage. Sending two codes gave the mechanism time to finish.
3. Multics and then Unix reduced this to a single 0x0A in the file and let the terminal driver expand it, which is why the C standard library still has the notion of translating in text mode.
4. Line-ending conventions by system:

System	Bytes	Escape	Era
Unix, Linux, macOS X+	0A	\n	1970 onward
Windows, DOS, HTTP	0D 0A	\r\n	1981 onward
Classic Mac OS 1-9	0D	\r	1984 to 2001

5. Several network protocols mandate CRLF regardless of platform, including HTTP/1.1 in RFC 9112, SMTP in RFC 5321, and FTP in RFC 959. This is a standard, not a convention, and sending a bare LF is a protocol violation.
6. Encodings and their byte order marks, all dumped from real files created for this chapter:

```
$ xxd -l 12 bom_utf8.txt
00000000: efbb bf68 656c 6c6f 0a                ...hello.

$ xxd -l 12 utf16.txt
00000000: fffe 6800 6500 6c00 6c00 6f00        ..h.e.l.l.o.

$ xxd -l 6 latin1.txt
00000000: 6361 66e9 0a                        caf..

$ xxd -l 6 utf8.txt
00000000: 6361 66c3 a90a                      caf...
```

7. Read those carefully. The word `café` is five bytes in Latin-1 and six bytes in UTF-8, because the accented `é` needs one byte in the first and two in the second.
8. The UTF-8 byte order mark is `EF BB BF`. It encodes `U+FEFF`, a character that originally meant zero-width non-breaking space.
9. UTF-8 has no byte order to mark, so the mark is purely a flag. The Unicode standard permits it but does not recommend it, and POSIX tools do not expect it.
10. A UTF-8 byte order mark breaks a shebang line, breaks a shell script, breaks a JSON parser that follows RFC 8259 strictly, and appears as `ï»¿` when a file is misread as Latin-1. Microsoft tools add it by default; most others do not.
11. UTF-16 requires a mark or external knowledge, because `fffe` and `feff` mean opposite byte orders. The dump above shows `fffe`, which is little-endian.
12. Useful commands: `file -i`, `iconv -f X -t Y`, `dos2unix`, `unix2dos`, `sed -i 's/\r$//'`, and in Git, `core.autocrlf` and a `.gitattributes` file with `* text=auto`.
13. The honest version: `file` reporting “ASCII text” is a guess based on statistics. It checks whether all bytes fall in printable and common control ranges. A binary file made only of printable bytes will be called `text`, and a UTF-8 file truncated mid-character will be called `data`.

■ 20.4.6 WORDS – remember these

1. Text file — a file whose bytes read as readable characters — a byte stream interpretable under a character encoding and divided by line terminators.
2. CR — carriage return — US-ASCII `0x0D`, historically returning the print head to column zero.
3. LF — line feed — US-ASCII `0x0A`, historically advancing the paper one line.
4. CRLF — the Windows and network line ending — the two-byte sequence `0x0D 0x0A`, mandated by HTTP, SMTP and FTP.
5. Encoding — the rule mapping bytes to characters — a character encoding scheme such as US-ASCII, ISO-8859-1 or UTF-8.
6. BOM — three or two marker bytes at the start of a text file — the byte order mark, `U+FEFF`, encoded as `EF BB BF` in UTF-8.
7. Mojibake — text that has turned into garbage symbols — the result of decoding bytes with a different encoding than they were written in.

20.5 Common formats and how they are built

■ 20.5.1 PLAIN – in simple words

1. Most real formats are built from two separate ideas: a **container** and the things inside it.
2. The container decides how to store several pieces, where each one starts, and what each one is called.
3. The pieces inside can each be squeezed or encoded in their own way.
4. A ZIP file is the most successful container ever made. It holds a list of named items, each optionally compressed.
5. That is why so many things you use every day are secretly ZIP files.
6. A Word document, an Excel sheet, a PowerPoint deck, a Java program and an Android app are all ZIP files with agreed contents inside.
7. A PDF is a different shape. It is a numbered list of objects with a directory at the end telling you where each object lives.
8. Then there are the small human-readable data formats, JSON, XML, YAML, CSV and TOML, each with a different set of traps.

■ 20.5.2 PLAIN – a picture in your head

1. Think of a filing cabinet with a card index drawer at the bottom.
2. Each folder in the cabinet has its own label card stapled to the front, and the contents may be folded up tightly to save space.
3. The card index at the bottom lists every folder and says exactly how far in it sits.
4. To find one folder, you read the index, walk straight to that spot, and pull it out. You never touch the others.
5. To add a folder, you push it in at the end, then write a new index and put it after everything. The old index is simply ignored.
6. That is a ZIP file exactly: local labels on each item, and a central directory at the very end.

Where this comparison breaks: with a real cabinet you would throw the old index away. A ZIP file keeps it, sitting in the middle of the file, unreferenced. That leftover is not harmless: some tools read the file forwards and see the old listing, while correct tools read the central directory at the end and see the new one. Attackers have used this exact disagreement to smuggle content past scanners. It is known in the Android world as one of the master key bugs.

■ 20.5.3 PLAIN – a worked example

1. A Microsoft Word document is a ZIP file. Here is a real one, created with a library, then opened with a plain unzip tool.

```
$ file report.docx
report.docx: Microsoft Word 2007+

$ xxd -l 16 report.docx
00000000: 504b 0304 1400 0000 0800 2e0f 0d5d ad52  PK.....].R

$ unzip -l report.docx
  Length      Date    Time    Name
  -----  -
      1738   2026-08-13  01:57   [Content_Types].xml
        734   2026-08-13  01:57   _rels/.rels
        721   2026-08-13  01:57   docProps/core.xml
       1132   2026-08-13  01:57   docProps/app.xml
       1693   2026-08-13  01:57   word/document.xml
       1227   2026-08-13  01:57   word/_rels/document.xml.rels
     349458   2026-08-13  01:57   word/styles.xml
     438131   2026-08-13  01:57   word/stylesWithEffects.xml
       2535   2026-08-13  01:57   word/settings.xml
       2811   2026-08-13  01:57   word/fontTable.xml
      10939   2026-08-13  01:57   word/theme/theme1.xml
       5513   2026-08-13  01:57   word/numbering.xml
       8324   2026-08-13  01:57   docProps/thumbnail.jpeg
  -----
     826305
                   17 files
```

2. The first four bytes are 504b 0304, the ZIP local file header signature. It is a ZIP and nothing else.
3. Your actual words live in word/document.xml. Everything else is styles, relationships, fonts, and a preview image.
4. Look at the sizes. The text of a two-line document is 1693 bytes. The style definitions are 349458 bytes. That is why an empty Word file is not empty.
5. The docProps/thumbnail.jpeg entry is a real JPEG, which is why a document can leak a picture of an earlier version of itself.

6. You can extract, edit the XML with a text editor, and re-zip, and Word will open the result. This is how many document-generation tools work.

■ 20.5.4 PLAIN – what is really happening inside

1. A ZIP file is stored as three kinds of record, in this order in the file.
2. First, for every item: a **local file header** giving the name, the compression method, the sizes and a checksum, immediately followed by the item's bytes.
3. Then, after all the items: a **central directory**, one record per item, repeating the same information plus the exact offset where that item's local header starts.
4. Last, a small **end of central directory record** saying how many entries there are and where the central directory begins.
5. A correct reader starts at the end, finds the end record, jumps to the central directory, and reads the list. It never scans forwards.
6. That design has three consequences worth knowing.
7. You can pull out one file from a five-gigabyte archive without reading the rest, because you know the exact offset.
8. You can add data before the archive without breaking it, because all offsets are found through the end record, which self-corrects. That is how self-extracting archives work: a program at the front, a ZIP at the back.
9. You can append junk after the archive and many tools still open it, because they search backwards for the end record.

■ 20.5.5 TECHNICAL – the engineer's version

1. ZIP was created by Phil Katz in 1989 for PKZIP. The specification is Katz's APPNOTE.TXT, maintained by PKWARE, not by a standards body. It is a de facto standard.
2. Signature values, all confirmed by parsing a real archive built for this chapter: local header 0x04034B50, central directory 0x02014B50, end of central directory 0x06054B50. Stored little-endian, so PK\3\4 on disk.
3. The structure of a real three-entry archive, parsed with a short Python script written for this chapter:

```
total size: 556 bytes
LOCAL  @    0 name=note.txt  method=0 csize= 26 usize= 26
LOCAL  @   92 name=data.csv  method=0 csize= 12 usize= 12
LOCAL  @  170 name=tiny.png  method=8 csize= 64 usize= 69
EOCD   @  534 entries=3 cd_size=234 cd_offset=300
CENTRAL @  300 name=note.txt  local_hdr_at=  0
CENTRAL @  378 name=data.csv  local_hdr_at=  92
CENTRAL @  456 name=tiny.png  local_hdr_at= 170
```

4. Method 0 is Stored, meaning no compression. Method 8 is Deflate. The tool chose Stored for the tiny text files because compressing them made them larger.
5. Deflate is specified in RFC 1951, published in 1996 by Peter Deutsch. It combines LZ77 matching over a 32 KiB window with Huffman coding.
6. Appending to an archive was tested directly:

```
$ cp demo.zip appended.zip
$ printf 'EXTRA_TRAILING_DATA' >> appended.zip
$ unzip -l appended.zip
      26  note.txt
      12  data.csv
      69  tiny.png
```

The archive still lists correctly with 19 extra bytes glued on the end. 7. Prepending was also tested. Concatenating a Linux executable and a ZIP produced a file that file calls an ELF executable and unzip reads as a valid archive. Both tools are correct. 8. The original ZIP format uses 32-bit sizes and offsets and a 16-bit entry count, capping it at 4 GiB and 65535 entries. Zip64 extensions, added to APPNOTE around 2001, raise those to 64-bit fields. 9. ZIP-based formats and what identifies each:

Extension	Contents marker	Standard
.docx .xlsx .pptx	[Content_Types].xml	ECMA-376, ISO 29500
.odt .ods .odp	mimetype entry first	ISO/IEC 26300
.jar .war	META-INF/MANIFEST.MF	Java SE spec
.apk	AndroidManifest.xml	Android platform
.epub	mimetype entry first	EPUB 3, W3C
.xpi	manifest.json	Mozilla add-on

10. OOXML, the DOCX family, was standardized as ECMA-376 in 2006 and ISO/IEC 29500 in 2008. OpenDocument was standardized by OASIS in 2005 and as ISO/IEC 26300 in 2006.
11. PDF has a completely different shape. Adobe released PDF 1.0 in 1993. It became an open ISO standard, ISO 32000-1, in 2008, and ISO 32000-2, known as PDF 2.0, in 2017.
12. A PDF is a header, then a body of numbered indirect objects, then a cross-reference table listing the byte offset of every object, then a trailer. Real tail of a 33-page PDF found on this machine:

```
0000683320 00000 n
0000683847 00000 n
trailer
<<
/Size 302
/Root 301 0 R
/Info 300 0 R
>>
startxref
683904
%%EOF
```

13. startxref gives the byte offset of the cross-reference table, and %%EOF ends the file. A reader also starts at the end here, exactly like ZIP.
14. /Root 301 0 R is an indirect reference to object 301, which is the document catalogue. The 0 is a generation number used by incremental updates.
15. PDF supports incremental update: append new objects and a new cross-reference table, leaving the old ones in place. This is why a redacted PDF can still contain the unredacted original.
16. The small data formats compared, with the trap that catches people:

Format	Year, spec	Main trap
JSON	RFC 8259, 2017	no comments, no trailing comma
XML	W3C Rec, 1998	entity expansion attacks
YAML	1.2, 2009	indentation and type guessing
CSV	RFC 4180, 2005	quoting, commas, encodings
TOML	1.0.0, 2021	deep nesting is awkward

17. Detail on each trap. JSON forbids comments and trailing commas by design, and RFC 8259 requires UTF-8 for network interchange. JSON numbers have no defined precision limit, so a 64-bit integer can lose accuracy in a parser that uses doubles.

18. XML supports entity definitions. A document defining nested entities that each expand ten times can expand to gigabytes from a few kilobytes. That is the billion laughs attack, first publicized in 2003. Parsers must disable external entity resolution, or an XML file can read local files off a server.
19. YAML 1.1 interpreted unquoted NO, ON, OFF and YES as booleans. In a country list, NO for Norway became false. YAML 1.2's core schema fixed this in 2009, but many libraries still default to 1.1 behaviour. Check your library, not the specification.
20. CSV is not one format. RFC 4180 is informational and describes common practice: comma separator, CRLF line endings, double quotes doubled inside quoted fields. Real files use semicolons in European locales, tabs, and embedded newlines inside quoted fields.
21. TOML was created by Tom Preston-Werner in 2013 and reached version 1.0.0 in January 2021. It is used by Rust's Cargo and by Python packaging in `pyproject.toml`.
22. Containers versus codecs, restated: a .mp4 file is a container defined by ISO/IEC 14496-12. It says nothing about how the video is encoded. Inside it can be H.264, H.265, AV1 or several others, plus AAC or Opus audio. "It is an MP4" tells you nothing about whether a device can play it.

■ 20.5.6 WORDS - remember these

1. Container — a format that holds other pieces — a structure defining framing, naming and offsets for enclosed streams.
2. Codec — the rule for squeezing one stream — coder-decoder, the algorithm encoding a media stream inside a container.
3. Central directory — the index at the end of a ZIP — the authoritative list of entries with their local header offsets.
4. Deflate — the usual ZIP compression method — LZ77 with a 32 KiB window plus Huffman coding, specified in RFC 1951.
5. Indirect object — a numbered item in a PDF — an object addressed as "number generation R" and located through the cross-reference table.
6. Incremental update — adding to a PDF without rewriting it — appending new objects and a new xref section pointing back to the previous one.
7. Billion laughs — an XML file that explodes in size — a recursive entity expansion denial-of-service attack.

20.6 What an executable file really is

■ 20.6.1 PLAIN - in simple words

1. There is nothing magical about a program file. It is a file, like a photo or a letter.
2. What makes it special is what the bytes are, and a small label at the front that explains how to arrange them.
3. Most of the bytes are **machine instructions**: the exact numeric codes that the processor knows how to obey.
4. The rest is data the program needs: text it will print, tables it will read, space it will want.
5. The label at the front is called a **header**. It says things like "the code starts at this offset and should be placed at this address" and "start running at this point".
6. The part of the operating system that reads that header and does what it says is called the **loader**.
7. So running a program is: read a header, copy or map the pieces into memory in the right places, then jump to the starting point.

8. Three families of headers dominate the world, one per operating system, and they do the same job in different words.
9. Windows uses PE. Linux uses ELF. macOS uses Mach-O.

■ 20.6.2 PLAIN – a picture in your head

1. Think of a flat-pack wardrobe delivered in a box.
2. The box holds planks, screws and panels. None of it is a wardrobe yet.
3. On top sits an instruction sheet: put panel A here, panel B there, this side faces the wall, start with step one.
4. The planks are the machine instructions and data. The instruction sheet is the header.
5. The person following the sheet is the loader. It never invents anything. It only does what the sheet says.
6. A wardrobe from a different manufacturer comes with a different style of sheet. The planks might be almost identical. The sheet is not.
7. Give a Swedish sheet to a fitter trained on a different system and nothing gets built, even though the wood is fine.

Where this comparison breaks: a wardrobe is assembled once and then stands on its own. A program is re-assembled from the file every single time you run it, and usually most of the planks are never actually moved. The loader mostly just says “pretend these pages of the file are at these addresses” and the pages are read from disk only when first touched. It is less like building a wardrobe and more like handing someone a catalogue and promising to fetch each plank the moment they reach for it.

■ 20.6.3 PLAIN – a worked example

1. We compiled the same six-line C program twice on the same Linux machine, once for Linux and once for Windows, and asked what each file is.

```
$ gcc -O2 -o hello hello.c
$ x86_64-w64-mingw32-gcc -O2 -o hello.exe hello.c

$ file hello
hello: ELF 64-bit LSB pie executable, x86-64, version 1
      (SYSV), dynamically linked, interpreter
      /lib64/ld-linux-x86-64.so.2, not stripped

$ file hello.exe
hello.exe: PE32+ executable (console) x86-64,
           for MS Windows, 19 sections

$ ls -l hello hello.exe
-rwxr-xr-x 1 root root 15960 hello
-rwxr-xr-x 1 root root 249671 hello.exe
```

2. Same source code. Same processor family, x86-64. Same instructions inside for the parts we wrote.
3. Yet the Linux file will not run on Windows and the Windows file will not run on Linux without a compatibility layer.
4. The reason is not the instructions. It is the header format and, more importantly, which library functions the file expects to find.
5. The Windows file is fifteen times larger here because the cross-compiler statically included more of its runtime and debug information.
6. Here is the first line of each file, side by side:

```
hello      7f45 4c46 0201 0100 .ELF....
hello.exe  4d5a 9000 0300 0000 MZ.....
```

7. Four bytes and two bytes. That is the visible difference between an entire operating system's idea of a program and another's.

■ 20.6.4 PLAIN – what is really happening inside

1. A processor does not know about files. It knows about memory addresses and instruction bytes.
2. Every program file therefore has to answer four questions for the loader.
3. Question one: which processor are these instructions for? An ARM chip cannot run x86 codes, so the header names the architecture.
4. Question two: which pieces of the file go where in memory? The header lists chunks, each with a file offset, a memory address, a length, and permissions.
5. Question three: what permissions does each chunk need? Code needs to be readable and executable but not writable. Data needs to be readable and writable but not executable.
6. Question four: where do I start? The header holds one address, the **entry point**.
7. There is a fifth question for most modern programs: which shared libraries do I need? The header lists them by name.
8. The loader answers these in order, builds a fresh empty memory space for the new program, maps in the chunks, and jumps to the entry point.
9. That is the entire trick. Everything else in an executable format is bookkeeping around those five answers.

■ 20.6.5 TECHNICAL – the engineer's version

1. The three formats and their lineage:

Format	Origin	Used by
PE/COFF	Windows NT 3.1, 1993	Windows, UEFI
ELF	System V R4, circa 1989	Linux, BSD, Solaris
Mach-O	Mach at CMU, 1980s	macOS, iOS

2. PE stands for Portable Executable. It extends COFF, the Common Object File Format from AT&T Unix System V, and was introduced with Windows NT 3.1 in 1993. The specification is published by Microsoft as the PE Format document.
3. ELF stands for Executable and Linkable Format. It was defined by Unix System Laboratories for System V Release 4 and published in the System V Application Binary Interface. Linux moved from the older a.out format to ELF during 1995 and 1996.
4. Mach-O comes from the Mach microkernel project at Carnegie Mellon University, was used by NeXTSTEP from 1989, and was inherited by Mac OS X in 2001.
5. A structural comparison:

Concept	PE name	ELF name
Shared library	.dll	.so
Static library	.lib	.a
Loadable unit	section	segment
Import list	Import Directory	DT_NEEDED entries
Start address	AddressOfEntryPoint	e_entry

Concept	PE name	ELF name
Base address	ImageBase	p_vaddr of first LOAD
Runtime linker	ntdll loader	ld.so

6. Mach-O uses `.dylib` for shared libraries, `.a` for static archives, load commands instead of a fixed table, `LC_MAIN` for the entry point and `/usr/lib/dyld` as the dynamic linker.
7. All three support position-independent code so that the operating system can place the image at a randomized base address. This is Address Space Layout Randomization, standard on Windows since Vista in 2007, on Linux since the mid-2000s, and on macOS since 10.5 in 2007.
8. UEFI firmware executables also use PE/COFF, with subsystem values 10 to 13. This is why a Linux bootloader signed for Secure Boot is a PE file even though Linux itself uses ELF.
9. There is genuine disagreement among engineers about whether a modern executable format should keep separate section and segment tables at all. ELF keeps both, one for the linker and one for the loader. Some newer formats such as WebAssembly use a single section list. The ELF side argues the split allows stripping without touching load behaviour; the critics argue it duplicates information and has caused parser confusion bugs.

■ 20.6.6 WORDS - remember these

1. Executable — a file the system can run — an image containing machine code plus a header describing its memory layout and entry point.
2. Header — the label at the front of a program file — a fixed structure giving architecture, layout tables and the entry point.
3. Loader — the part of the system that sets a program up — kernel code that parses the image, creates an address space and maps segments.
4. Entry point — where execution begins — the virtual address stored in `e_entry` on ELF or `AddressOfEntryPoint` on PE.
5. Machine code — the numbers a processor obeys directly — encoded instructions for one instruction set architecture.
6. ASLR — placing a program at a random address each run — Address Space Layout Randomization, requiring position-independent code and relocations.

20.7 Anatomy of a PE file

■ 20.7.1 PLAIN - in simple words

1. Open any Windows `.exe` in a hex viewer and the first two letters are `MZ`.
2. Those are the initials of Mark Zbikowski, a Microsoft engineer who designed the executable header for MS-DOS in the early 1980s.
3. Right after that sits a tiny complete DOS program. Its only job is to print “This program cannot be run in DOS mode” and quit.
4. That stub is still there in 2026, in every Windows program you own, even though no one has run DOS in decades.
5. About 128 bytes in comes the letters `PE` followed by two zero bytes. That is where the modern file really starts.
6. After that come three tables: one describing the machine and how many pieces there are, one describing the whole image, and one listing the pieces.
7. Each piece is called a **section**, and each has a short name beginning with a dot: `.text` for code, `.data` for changeable data, `.rdata` for read-only data.

8. There is also a list of functions the program needs from other files, and if it is a library, a list of what it offers.
9. And one small number decides whether Windows opens a black console window for the program or not.

■ 20.7.2 PLAIN – a picture in your head

1. Think of a very old building that has been refurbished many times.
2. At the front door there is still a Victorian gas lamp bracket. It has not carried gas since 1930. It is bolted to the wall and nobody removes it.
3. It costs almost nothing to leave, and taking it off risks damaging the wall.
4. The DOS stub is that gas lamp bracket. Sixty-four bytes of dead code plus a sentence.
5. Once you walk past it, the building inside is completely modern: a directory board in the lobby listing every floor, and every floor labelled by purpose.
6. The directory board is the section table. Each floor is a section.

Where this comparison breaks: the gas bracket really is useless, whereas the DOS stub has one live job left. The last four bytes before it, at offset 0x3C, hold the offset of the real PE header. Every Windows loader reads that number. So the stub region is not dead space; it contains the pointer that makes the whole format work. Removing the stub without fixing that pointer breaks the file.

■ 20.7.3 PLAIN – a worked example

1. Here are the first 128 bytes of a real Windows executable we compiled for this chapter.

```
$ xxd -l 128 hello.exe
00000000: 4d5a 9000 0300 0000 0400 0000 ffff 0000  MZ.....
00000010: b800 0000 0000 0000 4000 0000 0000 0000  .....@.....
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000030: 0000 0000 0000 0000 0000 0000 8000 0000  .....
00000040: 0e1f ba0e 00b4 09cd 21b8 014c cd21 5468  ....!...L.!Th
00000050: 6973 2070 726f 6772 616d 2063 616e 6e6f  is program canno
00000060: 7420 6265 2072 756e 2069 6e20 444f 5320  t be run in DOS
00000070: 6d6f 6465 2e0d 0d0a 2400 0000 0000 0000  mode....$. ....
```

2. Bytes 0 and 1 are 4d 5a, the letters MZ. That is the DOS header magic.
3. Look at offset 0x3c, which is on the fourth line: the four bytes 8000 0000. Little-endian, that is the number 0x00000080, decimal 128.
4. That is the promise: the real header begins at byte 128 of this file.
5. Between offset 0x40 and 0x78 sits the stub. The bytes 0e 1f ba 0e 00 b4 09 cd 21 are real 16-bit DOS instructions.
6. Decoded: push cs, pop ds, load the address of the message into dx, put 9 in ah, then int 21h which is the DOS system call to print a string.
7. Then b8 01 4c cd 21 is: put 0x4C01 in ax and call DOS again, which means terminate with exit code 1.
8. Then the message itself, ending with 0d 0d 0a 24. The 24 is a dollar sign, which is how DOS marked the end of a string.
9. Now jump to byte 128 and look:

```
$ xxd -s 128 -l 32 hello.exe
00000080: 5045 0000 6486 1300 5e24 7d6a 0032 0300  PE..d...^$}j.2..
00000090: 5307 0000 f000 2600 0b02 0229 006c 0000  S....&....).l..
```

10. 5045 0000 is PE\0\0. The modern file starts here, exactly where the pointer said.

■ 20.7.4 PLAIN – what is really happening inside

1. Read the bytes after PE\0\0 in order and every field tells you something useful.
2. 6486 little-endian is 0x8664, the code for the x86-64 processor family. An ARM Windows binary would say 0xAA64.
3. 1300 is 0x0013, decimal 19. The file has nineteen sections.
4. 5e24 7d6a is 0x6A7D245E, a Unix timestamp. Converted, that is 13 August 2026 at 01:56:46 UTC, which is when we compiled it.
5. f000 is 0x00F0, decimal 240: the size of the next header.
6. 0b02 is 0x020B, the marker meaning this is a 64-bit image. A 32-bit one would say 0x010B.
7. The 240-byte header that follows holds the entry point, the preferred load address, the alignment rules, the stack size, and a table of sixteen directories pointing at imports, exports, resources and relocations.
8. After that comes the section table: nineteen fixed-size records, each with a name, a size, a memory address, and a file offset.
9. The loader reads exactly this and nothing more before it starts mapping.

■ 20.7.5 TECHNICAL – the engineer's version

1. Real dump of the optional header of the file compiled for this chapter, trimmed to fit the page:

```
$ objdump -x hello.exe
architecture: i386:x86-64
start address 0x0000000140001410
Characteristics 0x26
  executable, line numbers stripped, large address aware
Time/Date      Thu Aug 13 01:56:46 2026
Magic          020b      (PE32+)
SizeOfCode     0000000000006c00
SizeOfInitializedData 0000000000009600
AddressOfEntryPoint 0000000000001410
BaseOfCode     0000000000001000
ImageBase      0000000140000000
SectionAlignment 00001000
FileAlignment  00000200
MajorSubsystemVersion 5
SizeOfImage    0003f000
Checksum       00041982
Subsystem      00000003      (Windows CUI)
DllCharacteristics 00000160
  HIGH_ENTROPY_VA, DYNAMIC_BASE, NX_COMPAT
SizeOfStackReserve 000000000200000
SizeOfHeapReserve 000000000100000
NumberOfRvaAndSizes 00000010
```

2. Subsystem 3 is IMAGE_SUBSYSTEM_WINDOWS_CUI, a console program. Value 2 is IMAGE_SUBSYSTEM_WINDOWS_GUI. This single 16-bit field is the entire difference between a program that opens a black console window and one that does not.
3. ImageBase 0x140000000 is the default preferred base for a 64-bit executable. A 32-bit executable defaults to 0x400000 and a DLL to 0x10000000.
4. DYNAMIC_BASE means the image opts in to ASLR, so the loader will place it elsewhere and apply base relocations. NX_COMPAT means data pages can be marked non-executable. HIGH_ENTROPY_VA allows 64-bit address randomization.
5. SectionAlignment 0x1000 is the page size in memory. FileAlignment 0x200 is the alignment on disk. Sections are therefore padded differently in the file and in memory, which is why file offsets

and virtual addresses differ.

6. The section table from the same binary:

```
$ objdump -h hello.exe
Idx Name      Size      VMA           File off
 0 .text      00006b68  0000000140001000 00000600
 1 .data      000000c0  0000000140008000 00007200
 2 .rdata     00000da0  0000000140009000 00007400
 3 .pdata     00000468  000000014000a000 00008200
 4 .xdata     0000050c  000000014000b000 00008800
 5 .bss       00000b80  000000014000c000 00000000
 6 .idata     000006d0  000000014000d000 00008e00
 7 .CRT       00000060  000000014000e000 00009600
 8 .tls       00000010  000000014000f000 00009800
 9 .reloc     00000084  0000000140010000 00009a00
10 .debug_aranges ...
```

7. What the standard sections hold:

Section	Contents	Permissions
.text	machine code	read, execute
.rdata	constants, import tables	read only
.data	initialized variables	read, write
.bss	zero-filled variables	read, write
.rsrc	icons, strings, manifest	read only
.reloc	base relocation entries	read only
.idata	import descriptors	read
.edata	export descriptors	read
.pdata	unwind function table	read only

8. Note .bss has file offset 00000000. It occupies no space in the file at all. The loader simply allocates zeroed pages of the stated size. This is why a program with a 100 MB static buffer can be a 50 KB file.

9. The import table, dumped from the same binary:

```
$ objdump -p hello.exe
DLL Name: KERNEL32.dll
d350 281 DeleteCriticalSection
d368 317 EnterCriticalSection
d380 628 GetLastError
d3d8 1034 MultiByteToWideChar
d422 1489 VirtualProtect
DLL Name: msvcrt.dll
d49e 82 __getmainargs
d4c8 97 __set_app_type
d51c 219 _fmode
d526 285 _initterm
```

10. Each imported name has a slot in the **Import Address Table**. Before loading, each slot holds a pointer to the name. After loading, the loader overwrites each slot with the real address of the function inside the loaded DLL. Calls go through the slot, so they cost one extra memory read.
11. The IAT location is Data Directory entry 12. In this file it is at RVA 0xd1c8, length 0x188, which is 392 bytes, or 49 pointers.

12. Because the IAT is a table of writable function pointers, hooking it is a classic technique for both debugging tools and malware. Overwrite the slot and every call to that function goes to your code instead.
13. A DLL adds an export table. We built a real DLL with two functions:

```
$ x86_64-w64-mingw32-gcc -shared -O2 -o mylib.dll mylib.c
$ file mylib.dll
mylib.dll: PE32+ executable (DLL) (console) x86-64,
               for MS Windows, 20 sections
$ objdump -p mylib.dll
Export Address Table -- Ordinal Base 1
  [0] +base[1] 1390 Export RVA
  [1] +base[2] 13a0 Export RVA
[Ordinal/Name Pointer] Table
  [0] add
  [1] mul
```

14. Every export has both a name and an ordinal number. Importing by ordinal is smaller and faster but breaks if the library is rebuilt with a different order. Importing by name is the normal choice.
15. The addresses in these tables are **RVA**s, relative virtual addresses: offsets from ImageBase. Add 0x140000000 to 0x1390 to get the real address if the image loads at its preferred base.
16. Tools that read PE files: `objdump -x` and `objdump -p` from GNU binutils, `dumpbin /headers` from Microsoft, `llvm-readobj --coff-all`, and the Python `pefile` library.

■ 20.7.6 WORDS – remember these

1. MZ header — the two letters at the start of every Windows program — the DOS IMAGE_DOS_HEADER, whose `e_lfanew` field at offset 0x3C locates the PE header.
2. DOS stub — the small dead program after the MZ header — 16-bit code invoking `int 21h` to print a message and terminate.
3. Section — one labelled region of a program image — a record in the section table with a name, RVA, size, file offset and characteristic flags.
4. RVA — an address measured from the start of the image — Relative Virtual Address, added to ImageBase to get a linear address.
5. IAT — the table of addresses of borrowed functions — the Import Address Table, patched by the loader with resolved function pointers.
6. Subsystem — the flag deciding console or window — a 16-bit field; 2 is `WINDOWS_GUI`, 3 is `WINDOWS_CUI`, 10 to 13 are `UEFI`.
7. Base relocation — a fix-up applied when the image loads elsewhere — an entry in `.reloc` naming an address that must be adjusted by the load delta.

20.8 Anatomy of an ELF file

■ 20.8.1 PLAIN – in simple words

1. Every Linux program begins with byte 127 followed by the letters E, L, F. That is the whole signature.
2. After it comes a small header of 64 bytes that answers the basic questions: 32-bit or 64-bit, which byte order, which processor, where to start.
3. Then the file has two separate tables of contents, and this surprises people.
4. One table is for the loader: it lists a handful of big chunks to map into memory. These are called **segments**.

5. The other table is for the linker and for tools: it lists many small named pieces such as `.text` and `.data`. These are called **sections**.
6. Both tables describe the same bytes, grouped differently, for two different readers.
7. When a program runs, only the first table matters. The second can be deleted entirely and the program still runs.
8. Most Linux programs also name a helper program inside themselves: the dynamic linker, which finds the shared libraries at start-up.
9. That helper is usually `/lib64/ld-linux-x86-64.so.2`, and it is itself an ELF file.

■ 20.8.2 PLAIN – a picture in your head

1. Think of a large printed cookbook that has two indexes at the back.
2. The first index is by chapter: Starters, Mains, Puddings. Three entries, big blocks, useful when you want to open the book roughly in the right place.
3. The second index is by recipe name: four hundred entries, precise, useful when you want one exact thing.
4. Both indexes describe the same pages. Neither is wrong. They serve different readers.
5. A cook in a hurry uses the chapter index. A librarian cataloguing the book uses the recipe index.
6. The loader is the cook in a hurry: it wants three or four big blocks and the permissions for each.
7. The linker and the debugger are the librarian: they want the exact name and boundary of every small piece.

Where this comparison breaks: you cannot tear the recipe index out of a cookbook and still have the chapter index work, because they refer to page numbers that would not change. In ELF you genuinely can delete the whole section table. The `strip` command does this, the file gets smaller, and the program runs identically. There is no equivalent to that in a book, and it is the clearest proof that the two tables truly serve different masters.

■ 20.8.3 PLAIN – a worked example

1. Here is the start of a real Linux executable compiled for this chapter, sixty-four bytes, which is exactly the ELF header.

```
$ xxd -l 64 hello
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF.....
00000010: 0300 3e00 0100 0000 8010 0000 0000 0000  ..>.....
00000020: 4000 0000 0000 0000 9836 0000 0000 0000  @.....6....
00000030: 0000 0000 4000 3800 0d00 4000 1f00 1e00  ....@.8...@....
```

2. `7f 45 4c 46` is the magic: 127, E, L, F.
3. `02` means 64-bit. `01` would mean 32-bit.
4. The next `01` means little-endian byte order. `02` would mean big-endian.
5. `01` again is the ELF version, which has been 1 since 1989 and has never changed.
6. `0300` is `0x0003`, the type. Type 3 is a shared object, which is what a modern position-independent executable is.
7. `3e00` is `0x003E`, decimal 62, the code for x86-64.
8. `8010 0000 ...` is the entry point: address `0x1080`.
9. `4000 ...` is 64: the program header table starts at byte 64, immediately after this header.
10. `9836 0000` is `0x3698`, decimal 13976: where the section header table starts, near the end of the file.
11. And the tool agrees with our hand reading:

```
$ readelf -h hello
Class: ELF64
```

```

Data:                2's complement, little endian
Type:                DYN (Position-Independent Executable)
Machine:             Advanced Micro Devices X86-64
Entry point address: 0x1080
Start of program headers: 64 (bytes into file)
Start of section headers: 13976 (bytes into file)
Size of this header: 64 (bytes)
Size of program headers: 56 (bytes)
Number of program headers: 13
Size of section headers: 64 (bytes)
Number of section headers: 31

```

■ 20.8.4 PLAIN - what is really happening inside

1. Thirteen program headers, thirty-one section headers. Only the thirteen matter at run time, and of those only four are of type LOAD.
2. Each LOAD entry says: take this many bytes from this file offset, put them at this virtual address, and give them these permissions.
3. One LOAD entry has read and execute permission. That is the code. One has read and write. That is the data.
4. A separate entry of type INTERP holds a plain text string: the path of the dynamic linker. On this machine it is `/lib64/ld-linux-x86-64.so.2`.
5. If that entry is present, the kernel loads the named program too, and gives control to it, not to your program.
6. The dynamic linker then reads a special region called the **dynamic section**, which is a list of tagged values.
7. One tag, `NEEDED`, appears once per required library. Our program has exactly one: `libc.so.6`.
8. Other tags point at the symbol table, the string table, the relocation lists, and the initialization function.
9. The dynamic linker opens each needed library, maps it, then walks the relocation lists patching addresses until every borrowed function has a real address.
10. Only then does it jump to your entry point.

■ 20.8.5 TECHNICAL - the engineer's version

1. Program headers of the same binary, as a table. `FileSiz` and `MemSiz` differ only for the writable segment, because `.bss` occupies memory but not file.

Type	File off	Virt addr	Flags
PHDR	0x000040	0x0000040	R
INTERP	0x000318	0x0000318	R
LOAD	0x000000	0x0000000	R
LOAD	0x001000	0x0001000	R E
LOAD	0x002000	0x0002000	R
LOAD	0x002db8	0x0003db8	RW
DYNAMIC	0x002dc8	0x0003dc8	RW
GNU_EH_FRAME	0x002014	0x0002014	R
GNU_STACK	0x000000	0x0000000	RW
GNU_RELRO	0x002db8	0x0003db8	R

2. `GNU_STACK` with `RW` and no `E` is what makes the stack non-executable. If this header were missing or had `E` set, the kernel would map an executable stack.

3. GNU_RELRO marks a region that the dynamic linker makes read-only after relocation is finished. It covers `.init_array`, `.fini_array`, `.dynamic` and, with full RELRO, the global offset table. This blocks a whole class of pointer-overwrite attacks.
4. The interpreter string is stored literally in the file and reported by `readelf -l` as [Requesting program interpreter: `/lib64/ld-linux-x86-64.so.2`].
5. Section headers, abbreviated from the real output of `readelf -SW hello`:

Section	Type	Address	Size
<code>.interp</code>	PROGBITS	0x318	0x1c
<code>.dynsym</code>	DYNSYM	0x3d8	0xa8
<code>.rela.dyn</code>	RELA	0x550	0xc0
<code>.rela.plt</code>	RELA	0x610	0x18
<code>.init</code>	PROGBITS	0x1000	0x1b
<code>.plt</code>	PROGBITS	0x1020	0x20
<code>.text</code>	PROGBITS	0x1060	0x109
<code>.rodata</code>	PROGBITS	0x2000	0x11
<code>.dynamic</code>	DYNAMIC	0x3dc8	0x1f0
<code>.got</code>	PROGBITS	0x3fb8	0x48
<code>.data</code>	PROGBITS	0x4000	0x10
<code>.bss</code>	NOBITS	0x4010	0x8
<code>.symtab</code>	SYMTAB	0	0x360

6. `.bss` has type `NOBITS`: no bytes in the file. `.symtab` has address 0: it is never loaded into memory.
7. The dynamic section, real output, trimmed:

```
$ readelf -d hello
(NEEDED)      Shared library: [libc.so.6]
(INIT)        0x1000
(FINI)        0x116c
(INIT_ARRAY)  0x3db8
(GNU_HASH)    0x3b0
(STRTAB)      0x480
(SYMTAB)      0x3d8
(PLTGOT)      0x3fb8
(JMPREL)      0x610
(RELA)        0x550
(FLAGS)       BIND_NOW
(FLAGS_1)     Flags: NOW PIE
```

8. `BIND_NOW` with full RELRO means every symbol is resolved at start-up rather than lazily on first call. This is the default on Ubuntu and most modern distributions. Lazy binding is faster to start and weaker to attack.
9. Relocations from the same file:

```
$ readelf -r hello
Relocation section '.rela.dyn':
   R_X86_64_RELATIVE          1160
   R_X86_64_RELATIVE          1120
   R_X86_64_GLOB_DAT  __libc_start_main@GLIBC_2.34
   R_X86_64_GLOB_DAT  __cxa_finalize@GLIBC_2.2.5
Relocation section '.rela.plt':
   R_X86_64_JUMP_SLOT  puts@GLIBC_2.2.5
```


10. R_X86_64_RELATIVE entries need no symbol lookup. They just say “add the load base to the value stored here”, and there are three because the image is position independent.
11. Static versus dynamic linking, measured on this machine with the same source:

Build	File size	text bytes
dynamic, PIE	15,960	1,369
static	785,360	668,329
dynamic, stripped	14,472	1,369

12. The static binary is 49 times larger because the entire C library is copied in. It has no INTERP header, no NEEDED entries, and ldd reports it as “not a dynamic executable”.
13. Stripping removed .symtab and .strtab, cutting 1,488 bytes and reducing the section count from 31 to 29. nm then reports “no symbols”. The program runs identically; only debugging is harder.
14. PIE, position-independent executable, is Type: DYN with an entry point of 0x1080, a small number, because the real base is decided at load time. Building with -no-pie produced Type: EXEC with entry 0x401070, a fixed address. PIE is the default on Ubuntu since 16.10 and on Debian since Stretch in 2017.
15. Tools that read ELF: readelf, objdump, nm, ldd, eu-readelf from elfutils, llvm-readelf, and LD_DEBUG=libs to watch the dynamic linker work.

■ 20.8.6 WORDS – remember these

1. Segment — one chunk the loader maps — a program header entry of type LOAD with a file offset, virtual address, sizes and permission flags.
2. Section — one named region for tools — a section header entry used by the linker, debugger and stripper, ignored at run time.
3. Dynamic section — the run-time instruction list — an array of tagged values including DT_NEEDED, DT_STRTAB, DT_RELA and DT_INIT.
4. Interpreter — the helper that starts your program — the path in the PT_INTERP header, normally ld.so, loaded by the kernel before your code.
5. PIE — a program with no fixed address — Position-Independent Executable, type ET_DYN, relocated to a random base by the kernel.
6. Stripping — deleting the names from a binary — removing .symtab and .strtab, shrinking the file without changing behaviour.
7. RELRO — making some data read-only after start-up — RELocation Read-Only, applied by ld.so via mprotect after relocations complete.

20.9 What happens when you double-click a program

■ 20.9.1 PLAIN – in simple words

1. You double-click an icon. A fraction of a second later a window appears. A great deal happened in between.
2. First, something has to work out which file you meant and whether you are allowed to run it.
3. Then a request goes to the core of the operating system: please replace a process with this program.
4. The core opens the file, reads the first bytes, and checks that it recognizes the format and that the instructions are for this processor.
5. It then throws away the old program’s memory entirely and builds a fresh, empty memory space.
6. It maps the pieces of the file into that space at the addresses the header asked for.

7. It writes your command line arguments and the environment settings onto the new program's stack, so the program can find them.
8. If the program needs shared libraries, the core starts a helper program first and hands control to it.
9. The helper finds each library, maps it in, and fills in every borrowed address.
10. Only then does control reach your program's real starting point, and even that is not your code yet. A short start-up routine runs first, and it calls your main.

■ 20.9.2 PLAIN – a picture in your head

1. Think of an empty theatre and a script arriving at the stage door.
2. The stage manager checks the envelope: right theatre, right company, do we have permission to perform this.
3. The stage crew clears the stage completely. Everything from the last performance goes.
4. The set pieces are then brought in and placed at marked positions on the floor. The script says exactly where each goes.
5. A list of props is pinned up: the ones we own, and the ones we must borrow from the props house next door.
6. A runner is sent to the props house. He fetches each borrowed item and puts it in its marked place.
7. Only when every marked place is filled does the stage manager call "curtain up", and the first line is spoken.

Where this comparison breaks: a real stage is fully dressed before the curtain rises. A program is not. The loader mostly writes down promises rather than moving anything. A page of code is fetched from disk only when the processor first tries to execute it, causing a page fault that the kernel quietly services. A large program can start running having read only a few dozen kilobytes of its own file. The set is built while the play is already running.

■ 20.9.3 PLAIN – a worked example

1. We traced the system calls of the tiny program from earlier. This is the real output, lightly trimmed to fit the page.

```
$ strace ./hello
execve("./hello", ["/hello"], 0x7ffe.. /* 151 vars */) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, ...) = 0x7ff5dfab6000
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL, 54027, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff5dfaa8000
close(3) = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\2\1\1\3\0\0..."..., 832) = 832
mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff5df800000
mmap(0x7ff5df828000, 1605632, PROT_READ|PROT_EXEC, ...)
mmap(0x7ff5df9b0000, 323584, PROT_READ, ...)
mmap(0x7ff5df9ff000, 24576, PROT_READ|PROT_WRITE, ...)
mmap(0x7ff5dfa05000, 52624, PROT_READ|PROT_WRITE|ANONYMOUS)
close(3) = 0
arch_prctl(ARCH_SET_FS, 0x7ff5dfaa5740) = 0
write(1, "hello, world\n", 13) = 13
exit_group(0) = ?
+++ exited with 0 +++
```

- Count the lines. Seventeen system calls to print thirteen characters. Only one of them, the `write`, is our program doing our work.
- Look at the four `mmap` calls for `libc.so.6`. Each one maps a different segment with different permissions: read only, read plus execute, read only again, then read plus write.
- The last one has `ANONYMOUS` and no file. That is `.bss`: zeroed memory backed by nothing on disk.
- `read(3, "\177ELF...", 832)` is the dynamic linker reading the first 832 bytes of the library to parse its ELF header and program headers.
- Here is the same journey as a picture.

```

you double-click
      |
      v
+-----+      name, permissions, association
| shell / file mgr |-----+
+-----+      |
      | fork + execve      |
      v                  v
+-----+      +-----+
| KERNEL      |      | check +x bit, ACLs |
| read first bytes |      +-----+
| match binfmt |
| new address space|
| map LOAD segments|
| build stack:   |      argv, envp, auxv
| load PT_INTERP |
+-----+
      | jump to ld.so entry
      v
+-----+
| ld.so      |      find libs, mmap them,
| dynamic linker |      apply relocations, run
+-----+      library init functions
      | jump to e_entry
      v
+-----+
| _start      |      set up argc/argv
| __libc_start_main|      init stdio, TLS, locale
+-----+
      | call
      v
      main(argc, argv)
      | return
      v
      exit -> atexit handlers -> _exit -> exit_group
      |
      v
      kernel frees memory, notifies parent

```

■ 20.9.4 PLAIN – what is really happening inside

1. Here is the whole sequence in thirty numbered steps. Steps are given for Linux; the Windows path is noted in the technical block below.
2. Step 1. The file manager receives a double-click on an icon and resolves it to a full path, following any symbolic links or shortcuts.
3. Step 2. It decides how to run it. On Linux it checks whether the file is executable; on Windows it consults the registry association for the extension.
4. Step 3. It checks permission. On Linux, the execute bit must be set for you. On Windows, an access control list is evaluated.
5. Step 4. The file manager calls `fork`, creating a child process that is a copy of itself.
6. Step 5. The child calls `execve` with the path, the argument list, and the environment list.
7. Step 6. The kernel opens the file and reads the first 256 bytes into a buffer.
8. Step 7. The kernel walks its list of registered binary format handlers, offering the buffer to each in turn.
9. Step 8. The ELF handler checks for 7f 45 4c 46 and accepts.
10. Step 9. It checks the class field, 64-bit here, and the machine field, 62 for x86-64, against the running kernel. A mismatch fails with `ENOEXEC`.
11. Step 10. The kernel checks the set-user-id and set-group-id bits and decides the new credentials.
12. Step 11. The kernel destroys the old address space: all mappings from the previous program are released.
13. Step 12. A brand-new, empty address space is created for the process.
14. Step 13. The kernel reads the program header table and iterates over the `PT_LOAD` entries.
15. Step 14. For each, it creates a mapping from the file into memory at the stated address, with the stated permissions. Nothing is copied yet.
16. Step 15. For a `PT_LOAD` whose memory size exceeds its file size, the extra is mapped as anonymous zeroed pages. That is `.bss`.
17. Step 16. For a position-independent executable, the kernel picks a random base address and adds it to every mapping address.
18. Step 17. The kernel creates the stack, at a randomized address, sized by the process resource limit, usually 8 MB.
19. Step 18. It writes onto that stack, from the top down: the environment strings, the argument strings, then pointer arrays for each, then a table of auxiliary values called the **auxiliary vector**.
20. Step 19. The auxiliary vector carries facts the program cannot easily discover otherwise: page size, the address of the program headers, the entry point, the real and effective user ids, and a pointer to sixteen random bytes.
21. Step 20. If the file has a `PT_INTERP` header, the kernel opens that path, maps its segments too, and remembers its entry point.
22. Step 21. The kernel sets the instruction pointer to the interpreter's entry point, not the program's, and returns to user mode.
23. Step 22. The dynamic linker begins. It relocates itself first, since nothing has fixed its own addresses yet.
24. Step 23. It reads the program's dynamic section and collects every `DT_NEEDED` library name.
25. Step 24. For each name, it searches: `DT_RPATH` and `DT_RUNPATH` in the file, then `LD_LIBRARY_PATH`, then the cache in `/etc/ld.so.cache`, then the default directories.
26. Step 25. Each found library is opened and mapped exactly as the main program was, and its own dependencies are added to the queue.
27. Step 26. With every object loaded, the linker resolves symbols. For each relocation entry, it looks

up the symbol name in the load order and writes the resolved address into the named slot.

28. Step 27. With full RELRO enabled, it then calls `mprotect` to make the global offset table and the dynamic section read-only.
29. Step 28. It runs the initialization functions of every library, in dependency order, then those of the program itself, from `DT_INIT` and `DT_INIT_ARRAY`.
30. Step 29. It jumps to the program's real entry point, `_start`, which is not your code. `_start` arranges the arguments into registers and calls `__libc_start_main`.
31. Step 30. `__libc_start_main` sets up thread-local storage, initializes the standard input, output and error streams, registers the cleanup handler, and finally calls `main`. When `main` returns, `exit` runs the registered handlers, flushes buffers, calls destructors, and issues `exit_group`, at which point the kernel frees everything and stores the exit status for the parent.

■ 20.9.5 TECHNICAL - the engineer's version

1. On Linux the kernel entry point is `execve(2)`, implemented in `fs/exec.c`. It fills a struct `linux_binprm`, reads `BINPRM_BUF_SIZE` bytes, currently 256, and calls `search_binary_handler`.
2. Registered handlers in a typical kernel: `binfmt_elf`, `binfmt_script`, `binfmt_misc`, and `binfmt_elf_fdpic` on some embedded targets. The old `binfmt_aout` was removed from mainline Linux in 2022.
3. Mapping is done with `vm_mmap` using `MAP_PRIVATE`, so pages are copy-on-write and shared between every process running the same binary until written.
4. The auxiliary vector entries most often used: `AT_PHDR`, `AT_PHENT`, `AT_PHNUM`, `AT_ENTRY`, `AT_PAGESZ`, `AT_RANDOM`, `AT_SECURE`, `AT_HWCAP` and `AT_HWCAP2`. You can print them with `LD_SHOW_AUXV=1 ./program`.
5. `AT_RANDOM` points at sixteen kernel-supplied random bytes. `glibc` uses them to seed the stack canary. This is why the canary differs on every run.
6. The dynamic linker's work can be observed directly:

```
$ LD_DEBUG=libs ./hello
find library=libc.so.6 [0]; searching
  search cache=/etc/ld.so.cache
    trying file=/lib/x86_64-linux-gnu/libc.so.6
calling init: /lib64/ld-linux-x86-64.so.2
calling init: /lib/x86_64-linux-gnu/libc.so.6
initialize program: ./hello

$ LD_DEBUG=bindings ./hello
binding file ./hello [0] to libc.so.6 [0]:
  normal symbol `__libc_start_main' [GLIBC_2.34]
binding file ./hello [0] to libc.so.6 [0]:
  normal symbol `puts' [GLIBC_2.2.5]
```

7. Note that our source called `printf` but the binding is to `puts`. The compiler replaced a `printf` with a constant string and no format specifiers by the cheaper `puts`. Verified in the disassembly:

```
$ objdump -d hello
0000000000001060 <main>:
   1060: f3 0f 1e fa    endbr64
   1064: 48 83 ec 08    sub     $0x8,%rsp
   1068: 48 8d 3d 95..  lea     0xf95(%rip),%rdi
   106f: e8 dc ff ff ff call    1050 <puts@plt>
   1074: 31 c0          xor     %eax,%eax
   107a: c3            ret
```

8. The resulting address space, from `/proc/PID/maps` of a running process, shows the pattern clearly: five mappings for the executable and five for `libc`, each with different permissions.

```
55ffa11de000-55ffa11e0000 r--p /usr/bin/sleep
55ffa11e0000-55ffa11e4000 r-xp /usr/bin/sleep
55ffa11e4000-55ffa11e5000 r--p /usr/bin/sleep
55ffa11e6000-55ffa11e7000 rw-p /usr/bin/sleep
55ffd305000-55ffd326000 rw-p [heap]
7f9296600000-7f9296628000 r--p libc.so.6
7f9296628000-7f92967b0000 r-xp libc.so.6
7f92967b0000-7f92967ff000 r--p libc.so.6
7f9296803000-7f9296805000 rw-p libc.so.6
```

9. On Windows the equivalent path is: `CreateProcess` in `kernel32` calls `NtCreateUserProcess` in `ntdll`, the kernel creates a section object for the image, maps it, and starts a thread at `RtlUserThreadStart`. The user-mode loader in `ntdll`, often called `LdrpInitializeProcess`, then walks the import directory, loads each DLL, calls each DLL's `DllMain` with `DLL_PROCESS_ATTACH`, patches the IAT, and finally jumps to `AddressOfEntryPoint`.
10. On macOS the kernel's `execve` recognizes Mach-O, maps segments, and loads `/usr/lib/dyld`, which reads `LC_LOAD_DYLIB` commands, consults the shared cache, binds symbols and calls `LC_MAIN`.
11. Approximate cost on the machine used here: `execve` plus dynamic linking for a one-library program completes in under two milliseconds. A large application linking against forty shared libraries can spend tens of milliseconds purely in symbol resolution, which is why `prelink` existed and why static linking is popular again for command-line tools.
12. Tools that observe each stage: `strace -f`, `ltrace`, `LD_DEBUG`, `LD_SHOW_AUXV`, `perf trace`, `/proc/PID/maps`, and on Windows Process Monitor and the `!dlls` command in `WinDbg`.

■ 20.9.6 WORDS - remember these

1. `exec` — replacing a process with a new program — the `execve(2)` family, which discards the old address space and never returns on success.
2. `fork` — making a copy of a process — `fork(2)`, creating a child with a copy-on-write duplicate of the parent's address space.
3. Auxiliary vector — a list of facts the kernel hands the new program — the `auxv` array of type-value pairs on the initial stack.
4. Page fault — the moment a page is actually fetched — a processor exception the kernel services by reading from the mapped file or allocating a zero page.
5. Copy-on-write — sharing memory until someone writes — mapping pages read-only and duplicating them on the first write fault.
6. Lazy binding — resolving a function's address on first call — PLT stubs that call into the linker once, then patch the GOT slot.
7. C runtime start-up — the code that runs before `main` — `_start` and `__libc_start_main`, which initialize TLS, `stdio` and `atexit`.

20.10 Installers and packaging

■ 20.10.1 PLAIN - in simple words

1. An installer is just a program whose job is to put other files in the right places.
2. There is nothing sacred about installing. You could do all of it by hand with a file manager, given enough patience.

3. A typical installer does five things.
4. It copies the program's files into a folder, usually somewhere central rather than in your downloads.
5. It tells the system which file types this program can open, so double-clicking works.
6. It creates shortcuts, in a menu, on a desktop, or in a dock.
7. It writes settings somewhere the program can find them later.
8. And it records enough information for the program to be removed cleanly.
9. That last one is the part most often done badly, which is why uninstalling often leaves rubbish behind.

■ 20.10.2 PLAIN – a picture in your head

1. Think of a new appliance being delivered to a flat.
2. The delivery team carries it to the right room. That is copying the files.
3. They plug it in and register it with the building's electrics. That is registering file types.
4. They put a label on the fuse box saying which switch controls it. That is the shortcut.
5. They fill in a form for the building manager: what was installed, when, and what has to be undone to remove it. That is the uninstall record.
6. A careless team dumps the box in the hallway and leaves. Later, nobody knows what to remove.

Where this comparison breaks: an appliance stays where it was put. Software changes the building. It can alter shared settings, replace shared libraries that other programs also use, and add background services that start every time the building's power comes on. Removing it is therefore not the reverse of a delivery. It is closer to undoing a renovation, which is why package managers that track every file exist at all.

■ 20.10.3 PLAIN – a worked example

1. A Debian package is not a mysterious binary. It is an ar archive, the same ancient Unix format used for static libraries.

```
$ xxd -l 24 zstd_1.5.5+dfsg2-2build1.1_amd64.deb
00000000: 213c 6172 6368 3e0a 6465 6269 616e 2d62  !<arch>.debian-b
00000010: 696e 6172 7920 2020                               inary

$ ar t zstd_1.5.5+dfsg2-2build1.1_amd64.deb
debian-binary
control.tar.zst
data.tar.zst

$ dpkg -I zstd_1.5.5+dfsg2-2build1.1_amd64.deb
new Debian package, version 2.0.
size 644020 bytes: control archive=1069 bytes.
    755 bytes,  17 lines   control
    1064 bytes,  17 lines  md5sums
Package: zstd
Version: 1.5.5+dfsg2-2build1.1
Architecture: amd64
Installed-Size: 1802
Depends: libc6 (>= 2.34), libgcc-s1 (>= 3.3.1),
        liblz4-1 (>= 1.8.0), liblzma5, libstdc++6,
        zlib1g (>= 1:1.1.4)
Section: utils
Priority: optional
```


2. Three members. `debian-binary` is a text file containing 2.0, the format version.
3. `control.tar.zst` holds the metadata: package name, version, dependencies, checksums, and optional scripts to run before and after install.
4. `data.tar.zst` holds the actual files, with their full destination paths inside.
5. Installing is therefore: read the control data, check the dependencies are satisfied, unpack the data archive at the root of the file system, record the file list, run the post-install script.
6. Uninstalling is exact, because the file list was recorded. This is the whole advantage of a package manager over an installer program.

■ 20.10.4 PLAIN – what is really happening inside

1. Different systems answer the same problem with different shapes.
2. Windows historically used self-contained installer programs: a single .exe that unpacks itself and does the work in code.
3. Microsoft's own answer, the Windows Installer, replaced code with data. An .msi file is a small relational database of tables: which files, which registry keys, which shortcuts, in which order.
4. Because it is data, the system itself performs the installation, can roll back a failure, and can uninstall precisely.
5. macOS mostly avoids installers. An application is a folder whose name ends in .app, containing the executable, the icons and everything else.
6. The Finder shows that folder as a single item. Dragging it to Applications is the entire installation.
7. Linux distributions use package managers with a central database and hard dependency rules, which is why installing one thing can pull in twenty others.
8. The newer Linux formats invert this. Instead of sharing libraries, each app carries its own, so it works on any distribution at the cost of size.
9. And on every platform there is now a signing layer: the operating system checks a cryptographic signature before running the installer, and warns you loudly if there is not one.

■ 20.10.5 TECHNICAL – the engineer's version

1. Windows packaging formats:

Format	What it is	Notes
MSI	relational database	Windows Installer, 1999
NSIS	scripted self-extractor	Nullsoft, 2000, open source
InstallShield	commercial toolkit	from the 1990s
MSIX	signed app container	since Windows 10, 2018
Portable exe	no install at all	user copies one file

2. Windows Installer shipped in 1999 alongside Office 2000. An MSI file is actually an OLE compound document holding roughly eighty defined tables such as File, Component, Registry, Shortcut and InstallExecuteSequence.
3. NSIS was written at Nullsoft for distributing Winamp and released in 2000. It produces a single executable containing a compressed payload and a compiled script. It is the most common format for open-source Windows applications.
4. Uninstall information on Windows lives under `HKLM\Software\Microsoft\Windows\CurrentVersion\Uninstall` one key per product, with `DisplayName`, `UninstallString` and `InstallLocation`.
5. macOS packaging:

Format	What it is	Notes
.app	a directory, not a file	bundle from NeXTSTEP
.dmg	disk image	mounts as a volume
.pkg	installer package	XAR archive, since 10.5
.mpkg	several pkgs together	metapackage

- An .app bundle contains Contents/Info.plist declaring the identifier, version and document types, Contents/MacOS/ with the Mach-O executable, and Contents/Resources/ with icons and localizations. It is an ordinary directory; cd into it from a terminal and everything is visible.
- Linux packaging: .deb from Debian, described above; .rpm from Red Hat, created in the mid-1990s, consisting of a lead, a signature header, a metadata header, and a compressed cpio payload.
- The distribution-independent formats:

Format	Started	Isolation model
AppImage	klik 2004, renamed 2011	none, single file
Flatpak	2015, was xdg-app	bubblewrap sandbox
Snap	Canonical, 2014–2016	AppArmor confinement

- An AppImage is a single executable file: a small runtime concatenated with a SquashFS image. Running it mounts the image with FUSE and executes the application inside. It requires no installation and no root.
- Flatpak shares common runtimes between applications through OSTree deduplication and sandboxes each app with bubblewrap and portals. Snap uses compressed SquashFS images mounted as loop devices and confined by AppArmor.
- Code signing by platform:

Platform	Mechanism	Introduced
Windows	Authenticode	1996
macOS	codesign + Gatekeeper	2012
macOS	notarization required	2019, Catalina
Linux	repository GPG signing	Debian, early 2000s

- Authenticode embeds a PKCS#7 signature in the PE file, referenced by Data Directory entry 4, the Security Directory. The signed hash deliberately excludes the checksum field and the certificate table itself, so those can change without invalidating the signature.
- Extended Validation certificates and, since 2023, the requirement that code-signing private keys live on hardware tokens or in a hardware security module, exist because stolen signing keys were used to sign malware. The Stuxnet worm, discovered in 2010, was signed with certificates stolen from two Taiwanese hardware companies.
- Apple's Gatekeeper arrived with OS X 10.8 Mountain Lion in 2012. From macOS 10.15 Catalina in 2019, software distributed outside the App Store must also be notarized: uploaded to Apple, scanned, and issued a ticket that the system checks.
- Windows SmartScreen, part of Windows since version 8 in 2012, warns about executables with little reputation history regardless of signing. A newly signed application still triggers it until enough installs accumulate. This is a reputation system, not a signature check, and the two are often confused.

16. Linux distributions sign the package index rather than each package. `apt` verifies a detached GPG signature on the Release file, which contains hashes of the package lists, which contain hashes of the packages. Trust flows down that chain.

■ 20.10.6 WORDS - remember these

1. Installer — a program that puts software in place — a bootstrapper performing file placement, registration and uninstall bookkeeping.
2. MSI — the Windows database installer format — an OLE compound file of installation tables executed by the Windows Installer service.
3. Bundle — a macOS application folder shown as one item — a directory with a defined layout and an `Info.plist` describing it.
4. Package manager — a tool that tracks every installed file — a database-backed system resolving dependencies and enabling exact removal.
5. Sandbox — a fence around a running application — a confinement mechanism such as bubblewrap, AppArmor or the macOS App Sandbox.
6. Code signing — proving who built a program — attaching a signature over a hash of the image, verified against a trusted certificate chain.
7. Notarization — Apple checking your app before users run it — submission to Apple's service, producing a ticket stapled to the distribution.

20.11 Scripts versus binaries

■ 20.11.1 PLAIN - in simple words

1. A `.sh` or `.py` file contains no machine instructions at all. It is plain text a human wrote.
2. Yet you can mark it executable and run it exactly like a program. Something must be bridging that gap.
3. The bridge is two characters at the very start of the file: a hash and an exclamation mark, written `#!`.
4. Those two characters are followed by the path of a real program.
5. When you run the file, the system reads that first line, and instead of trying to execute your text, it runs the named program and hands it your file.
6. So `./script.sh` really becomes `/bin/sh ./script.sh` behind your back.
7. The named program is called an **interpreter**. It reads your text line by line and does what it says.
8. The interpreter itself is an ordinary compiled program, with an ELF or PE header, exactly as described earlier.
9. So nothing was ever magic. One real program is running, and your file is its input.

■ 20.11.2 PLAIN - a picture in your head

1. Think of a sealed envelope handed to a receptionist.
2. On the outside is written: "Please pass to the French translator."
3. The receptionist does not read the letter. She reads only the instruction on the outside and walks the envelope to the right desk.
4. The translator opens it and works through the contents.
5. The `#!` line is the writing on the outside of the envelope. The kernel is the receptionist. The interpreter is the translator.
6. If the writing names a desk that does not exist, the receptionist comes back with an error, and the error is confusingly about the envelope, not the desk.

Where this comparison breaks: a receptionist can see the letter is in French even without the label, and would work it out. The kernel cannot and does not try. It reads exactly two characters. If they are not #!, it gives up with an error called “exec format error”, and any success after that is the shell doing extra work of its own accord, not the kernel being clever.

■ 20.11.3 PLAIN – a worked example

1. We wrote two shell scripts, identical except that one has a #! line.

```
$ cat s1.sh
#!/bin/sh
echo "I am a script, arg1=$1"

$ xxd -l 16 s1.sh
00000000: 2321 2f62 696e 2f73 680a 6563 686f 2022  #!/bin/sh.echo "

$ ./s1.sh hello
I am a script, arg1=hello

$ file s1.sh
s1.sh: POSIX shell script, ASCII text executable
```

2. The first two bytes are 23 21, which are # and !. That is the whole mechanism.
3. Now the file without a #! line, run two ways.

```
$ cat s2.sh
echo "no shebang here"

$ ./s2.sh
no shebang here

$ python3 -c "import os; os.execve('/tmp/ch20/s2.sh',
    ['s2.sh'], {})"
execve failed: [Errno 8] Exec format error: 's2.sh'
```

4. Read that carefully. The shell ran it fine. A direct execve refused.
5. The kernel genuinely could not run the file. Error 8 is ENOEXEC, no recognized format.
6. When the shell got the same error, it did something extra: it decided to run the file itself, as shell commands. That is a rule in the shell, defined by POSIX, not a kernel feature.
7. So the file “worked” for a reason that has nothing to do with the file being executable at the system level.

■ 20.11.4 PLAIN – what is really happening inside

1. Inside the kernel, running any file goes through a list of format handlers, tried in order.
2. One of them, on Linux called `binfmt_script`, is very simple.
3. It checks whether the first two bytes are # and !.
4. If so, it reads up to a limit of characters, stopping at the first newline.
5. It splits that into the interpreter path and, on Linux, at most one further argument.
6. It then restarts the whole exec operation, but with the interpreter as the program and with the argument list rewritten.
7. The new argument list is: the interpreter path, the optional argument from the #! line, the original file path, and then your original arguments.
8. That restart is limited. A script whose interpreter is itself a script is allowed, but only to a fixed depth, to stop infinite loops.

- Windows has no `#!` mechanism in the kernel. It uses file associations instead: `.py` is registered to the Python launcher, which then reads a `#!` line itself as a convention borrowed from Unix.

■ 20.11.5 TECHNICAL – the engineer’s version

- The mechanism is implemented in `fs/binfmt_script.c` in the Linux kernel. The `#!` sequence is often called the **shebang**, or hash-bang.
- It was added to Unix by Dennis Ritchie around 1980 and first appeared in the Eighth Edition of Research Unix. It is specified in POSIX only as an unspecified behaviour, meaning portable scripts may rely on it in practice but it is not formally guaranteed.
- Line length limits differ and this matters for portability:

System	<code>#!</code> line limit	Extra args
Linux	255 bytes	at most one
FreeBSD	8192 bytes	several
macOS	512 bytes	at most one
Solaris	1023 bytes	at most one

- Because Linux allows only one argument, `#!/usr/bin/env python3 -u` passes `python3 -u` to `env` as a single string on Linux and fails. Writing `#!/usr/bin/env -S python3 -u` works on GNU coreutils 8.30 and later, released in 2018.
- `#!/usr/bin/env python3` is the common idiom because it searches `PATH` instead of hard-coding a location, which differs across distributions and inside virtual environments.
- A shebang must be an absolute path. The kernel does no `PATH` search. That is the entire reason `env` is used as an indirection.
- If a script has CRLF line endings, the interpreter path becomes `/bin/sh\r`, which does not exist. The resulting message, `bad interpreter: No such file or directory`, is one of the most confusing errors in Unix, because the path printed looks correct on screen.
- `binfmt_misc`, added to Linux in 1997, generalizes this. You register a magic number or extension with an interpreter path by writing to `/proc/sys/fs/binfmt_misc/register`.
- This is how a Linux system transparently runs ARM binaries under QEMU, and how Windows executables can be handed to Wine automatically. It was not mounted in the sandbox used for this chapter, which is common inside containers.
- The interpreter receives the script by path, not on standard input. It can therefore seek in the file, which is why Python can read a `.pyc` cache and why a shell script can safely have binary data appended to it after an `exit` line.
- Cost comparison, approximate and measured informally: starting a compiled C binary that prints one line takes about 1 ms on the machine used here. Starting the same task through `python3` takes roughly 20 to 40 ms, dominated by importing the interpreter’s own startup modules.
- The honest version: saying “scripts are interpreted and binaries are compiled” is a useful lie. CPython compiles your source to bytecode before running it. Java compiles to bytecode and then to machine code at run time. Modern JavaScript engines do the same. The real distinction is when the translation happens and who ships the result, not whether translation happens.

■ 20.11.6 WORDS – remember these

- Shebang — the `#!` at the top of a script — the two-byte magic recognized by `binfmt_script`, followed by an absolute interpreter path.
- Interpreter — the program that reads and runs your text — an ordinary executable receiving the script path as an argument.

3. ENOEXEC — the error meaning “I cannot run this” — `errno` 8, returned by `execve` when no binary format handler accepts the file.
4. `binfmt_misc` — the Linux feature for registering new formats — a kernel module mapping magic bytes or extensions to interpreter paths.
5. `env` — the tool used to find an interpreter on `PATH` — `/usr/bin/env`, run from the shebang so the real path need not be hard-coded.
6. Bytecode — a compact instruction set for a virtual machine — the intermediate form produced by CPython, `javac` and similar compilers.

20.12 Archives and compression in practice

■ 20.12.1 PLAIN – in simple words

1. Two different jobs get muddled together: putting many files into one, and making things smaller.
2. Putting many files into one is called **archiving**. Making things smaller is called **compression**.
3. ZIP does both jobs in one format. It squeezes each file separately and then packs the squeezed pieces together.
4. Unix split the jobs. `tar` packs files together and does no squeezing at all. `gzip` squeezes one stream and knows nothing about files.
5. That is why you see `.tar.gz`: first `tar` joined everything into one stream, then `gzip` squeezed that whole stream.
6. The order matters enormously. Squeezing one big stream finds repetition between files. Squeezing each file alone cannot.
7. Squeezing everything as one stream is called **solid** compression. It gives smaller results and makes extracting a single file slower.
8. Different squeezing methods trade speed against size, and the trade is real and measurable.
9. There is no best one. There is a fastest, a smallest, and several sensible middles.

■ 20.12.2 PLAIN – a picture in your head

1. Think of packing a suitcase for a family of four.
2. Method one: each person rolls their own clothes tightly, and the four bundles go in. Anyone can pull out their own bundle without disturbing the others.
3. Method two: all the clothes are laid out together and vacuum-packed as one slab. It takes far less space, because socks fill the gaps between shirts.
4. But to get one shirt you must open the whole slab and repack it.
5. Method one is ZIP. Method two is `tar` plus `gzip`, or a solid `7z` archive.
6. If everyone’s clothes are similar, method two wins by a lot, because the machine notices the repetition across people.

Where this comparison breaks: vacuum packing removes air, which is a fixed saving. Compression removes repetition, which is not fixed at all. A slab of already-random data compresses by nothing, and can come out slightly larger because of the bookkeeping added. There is no method that shrinks every possible input. That is not an engineering limit; it is a counting argument, and it is provably true.

■ 20.12.3 PLAIN – a worked example

1. We compressed the same 47,575,040 bytes of C header files, all real text, with six settings on one machine, and timed both directions.

Tool	Size (bytes)	Ratio	Compress
zstd -3	8,220,027	5.79x	0.19 s
gzip -6	8,466,301	5.62x	1.30 s
gzip -9	8,384,098	5.67x	2.93 s
bzip2 -9	6,632,497	7.17x	3.67 s
zstd -19	5,895,562	8.07x	19.20 s
xz -6	5,760,568	8.26x	16.09 s

- Read the first two rows together. `zstd -3` produced a smaller file than `gzip -6` and did it about seven times faster.
- Now read the top and bottom rows. `xz -6` is 30 percent smaller than `zstd -3`, and took 85 times longer.
- Decompression times tell a different story again, and this is the number that matters for software you ship.

Tool	Decompress	Notes
zstd -3	0.06 s	same for -19
gzip -6	0.26 s	same for -9
xz -6	0.38 s	slower to build
bzip2 -9	1.12 s	slowest both ways

- Note that `zstd` decompresses at the same speed whether it was compressed at level 3 or level 19. Higher levels cost the packer more, not the unpacker.
- That single property is why `zstd` replaced `gzip` in Debian and Ubuntu packages, in Arch Linux packages, and in the Linux kernel's own build options.

■ 20.12.4 PLAIN - what is really happening inside

- Nearly all general compression works by finding repeats and referring back to them instead of writing them again.
- The reference says "go back this many bytes and copy this many". That is the LZ77 idea, published by Abraham Lempel and Jacob Ziv in 1977.
- How far back the tool may look is the **window**. A bigger window finds more repeats and needs more memory.
- After that, the remaining symbols are re-coded so that common ones use fewer bits. That is Huffman coding, or the newer arithmetic and range coders.
- Solid compression matters because of the window. If every file is compressed separately, the window is reset at each file boundary and cross-file repetition is invisible.
- In a directory of four thousand C header files that all begin with a similar licence comment, that repetition is enormous.
- This is why the same data, packed two ways, can differ in size by more than a factor of two.

■ 20.12.5 TECHNICAL - the engineer's version

- We measured solid versus per-file directly, on the same 4,394 files.

Method	Size (bytes)	Time
zip -9, per file	13,580,396	3.38 s
tar + gzip -9, solid	8,384,087	2.98 s
7z -mx=5 -ms=off	11,690,306	9.43 s
7z -mx=5 -ms=on	6,224,950	12.58 s

2. Compare rows one and two. Identical Deflate algorithm, identical level. The solid version is 38 percent smaller purely because the window spans file boundaries.
3. Compare rows three and four. Identical LZMA2 algorithm, identical level. Solid mode is 47 percent smaller.
4. The cost of solid mode is random access. Extracting the last file from a solid archive requires decompressing everything before it. ZIP can seek directly.
5. History and specifications:

Tool	Author, year	Specification
tar	Version 7 Unix, 1979	POSIX.1 ustar
gzip	Gailly and Adler, 1992	RFC 1952, DEFLATE RFC 1951
bzip2	Julian Seward, 1996	no formal spec
7-Zip	Igor Pavlov, 1999	LZMA, documented
xz	Tukaani project, 2009	xz file format spec
zstd	Yann Collet, 2016	RFC 8478

6. tar means tape archive. Its record size of 512 bytes and its habit of padding to 10,240-byte blocks come directly from nine-track magnetic tape drives. A modern .tar file is still padded to a multiple of 10,240 bytes, which is why a tar of one small file is 10,240 bytes.
7. Window sizes: Deflate uses 32 KiB, fixed by RFC 1951. bzip2 uses blocks of up to 900 KiB. LZMA defaults to 8 to 64 MiB depending on level. zstd's window grows with level, up to 128 MiB at level 22 with long mode enabled.
8. bzip2 uses a completely different approach, the Burrows-Wheeler transform published in 1994, which reorders data to group similar bytes before coding. It is slow in both directions and has largely been displaced.
9. zstd's speed comes from finite state entropy coding, an implementation of asymmetric numeral systems published by Jaroslav Duda in 2009. It achieves arithmetic-coding compression ratios at table-lookup speeds.
10. A practical rule that holds up: choose zstd for anything you compress once and decompress often, gzip when compatibility with everything matters, and xz only when the transfer saving genuinely outweighs a long build.
11. None of these is encryption. ZIP's original password scheme, from 1990, is broken and can be attacked with known plaintext. ZIP AES-256 and 7z AES-256 are sound, but 7z still leaves file names visible unless header encryption is switched on with -mhe=on.

■ 20.12.6 WORDS – remember these

1. Archive — many files joined into one — a container preserving names, sizes, permissions and directory structure.
2. Compression — making data smaller — removing redundancy so the original can be reconstructed exactly, in lossless schemes.

3. Solid — compressing everything as one stream — allowing back-references to cross file boundaries, at the cost of random access.
4. Window — how far back the packer may look — the maximum LZ77 match distance, 32 KiB in Deflate, up to 128 MiB in zstd.
5. Ratio — how many times smaller it got — original size divided by compressed size, quoted with the exact corpus and level.
6. Lossless — nothing is thrown away — the decompressed output is byte-identical to the input, unlike JPEG or MP3.

20.13 File metadata beyond the basics

■ 20.13.1 PLAIN – in simple words

1. Beyond size and dates, a file can carry extra labels that most tools never show you.
2. On Linux and macOS these are called **extended attributes**: small named pieces of data attached to a file but not part of its contents.
3. macOS uses them heavily. When you download a file, the system attaches a label meaning “this came from the internet”.
4. That label is why macOS asks you whether you are sure the first time you open a downloaded application.
5. Windows has a stranger version. A file on NTFS can have several separate contents, each with its own name.
6. The main one is what you see. The extra ones are invisible in Explorer, do not change the reported file size, and were abused for years to hide code.
7. Photographs carry a third kind: a block of camera information inside the image file itself.
8. That block can include the exact time, the camera model, the settings, and often the precise place the photo was taken.
9. Sharing a photo can therefore share your home address without you saying a word.

■ 20.13.2 PLAIN – a picture in your head

1. Think of a book returned to a library.
2. The book has a title and pages. That is the name and contents.
3. A librarian has stuck a small note inside the back cover: “returned late, donated by the Smith family, check for water damage”.
4. The note is not part of the book. Photocopying the book does not copy the note. Lending it might.
5. Those notes are extended attributes. Useful, invisible, and easily lost.
6. Now imagine a book where extra chapters are bound in behind the endpapers, invisible unless you know to look. The spine still says 300 pages.
7. That is a Windows alternate data stream.

Where this comparison breaks: the librarian’s note is written by a human who knows what it means. Extended attributes are written by programs, in binary, with names like `com.apple.quarantine`, and are silently dropped by many everyday operations: uploading, emailing, extracting from a plain zip, or copying with the wrong flag. The library at least knows the note existed.

■ 20.13.3 PLAIN – a worked example

1. Extended attributes on Linux, all real commands run for this chapter.

```
$ setfattr -n user.kedbyte -v "chapter20" note.txt
$ getfattr -d note.txt
```

```
# file: note.txt
user.kedbyte="chapter20"

$ ls -l note.txt
-rw-r--r-- 1 root root 26 Aug 13 01:57 note.txt

$ cp note.txt note_copy.txt
$ getfattr -d note_copy.txt
(no output)

$ cp --preserve=xattr note.txt note_copy2.txt
$ getfattr -d note_copy2.txt
# file: note_copy2.txt
user.kedbyte="chapter20"
```

2. The reported size is still 26 bytes. The attribute is stored outside the data.
3. A plain copy lost it silently. No warning, no error. Only the explicit flag preserved it.
4. Now the photo case. We took a real JPEG and added camera and location tags.

```
$ exiftool -Make="ACME" -Model="Phone X" \
  -DateTimeOriginal="2026:03:14 09:41:00" \
  -GPSLatitude=18.5204 -GPSLatitudeRef=N \
  -GPSLongitude=73.8567 -GPSLongitudeRef=E photo.jpg

$ exiftool -G1 photo.jpg
[IFD0]      Make           : ACME
[IFD0]      Camera Model Name : Phone X
[ExifIFD]   Date/Time Original : 2026:03:14 09:41:00
[Composite] GPS Position      : 18 deg 31' 13.44" N,
                               73 deg 51' 24.12" E
```

5. That is a location accurate to a few metres, a timestamp to the second, and the device, all inside an ordinary picture.
6. Removing it is one command, and it shrank the file by 2,311 bytes:

```
$ ls -l photo.jpg
-rw----- 1 root root 8520 photo.jpg
$ exiftool -all= photo.jpg
$ ls -l photo.jpg
-rw----- 1 root root 6209 photo.jpg
$ exiftool -Make -Model -GPSLatitude photo.jpg
(nothing listed)
```

■ 20.13.4 PLAIN – what is really happening inside

1. Extended attributes are stored by the file system next to the inode, or in a separate block if they grow too large.
2. They are name-value pairs. The name has a prefix declaring a namespace: `user.` for anything, `security.` for SELinux labels, `system.` for access control lists, `trusted.` for privileged data.
3. macOS uses the same mechanism with Apple-defined names. Two matter most.
4. `com.apple.quarantine` is set by browsers and mail clients on every downloaded file, and holds flags, a timestamp and the downloading application.
5. `com.apple.ResourceFork` is the modern home of the old resource fork, a second data stream that classic Macintosh files had from 1984.
6. Windows does it differently. On NTFS, every file is a set of named attributes, and the contents are just the unnamed `$DATA` attribute.

7. Adding a second named \$DATA attribute gives the file a second, hidden set of contents. The syntax is a colon: `report.txt:hidden`.
8. EXIF is not an operating system feature at all. It is a block of data placed inside the image file, in a JPEG marker segment, using the TIFF tag structure.
9. That is why EXIF survives copying, emailing and zipping, whereas extended attributes usually do not. It is part of the file's own bytes.

■ 20.13.5 TECHNICAL – the engineer's version

1. Linux extended attribute namespaces and their rules:

Namespace	Who can write	Typical use
user.	file owner	application metadata
trusted.	CAP_SYS_ADMIN	privileged tooling
security.	policy modules	SELinux labels
system.	kernel	POSIX ACLs

2. On ext4 the total size of all extended attributes for one file is limited to one file system block, normally 4096 bytes. XFS allows more. This is why attributes are used for labels, not for storage.
3. macOS quarantine values look like `0083;5f2b1c40;Safari;UUID`, giving flags, a hexadecimal timestamp, the agent name and an event identifier. Gatekeeper reads this before first launch and removes it after approval. `xattr -d com.apple.quarantine app` clears it manually.
4. NTFS alternate data streams have existed since NTFS shipped with Windows NT 3.1 in 1993, originally to support Macintosh resource forks through Services for Macintosh.
5. The reported file size is the length of the unnamed \$DATA stream only. `dir` shows no sign of the others. `dir /r`, added in Windows Vista, does. `Get-Item -Stream *` in PowerShell lists them.
6. Windows itself uses one: `Zone.Identifier`, the mark of the web, written by browsers and by Outlook. Office reads it and opens the document in Protected View. Removing it is what “Unblock” does in the file properties dialog.
7. Malware families including Backdoor.Rustock in 2006 and later commodity loaders stored payloads in alternate data streams to evade scanners that read only the default stream. Every serious scanner now enumerates streams. Streams are lost the moment a file is copied to FAT32, exFAT, a network share without support, or into a plain ZIP.
8. EXIF was first published by JEIDA in 1995 and is maintained today by CIPA and JEITA. It stores TIFF-format image file directories inside a JPEG APP1 marker segment beginning with the bytes `Exif\0\0`.
9. Commonly present tags, taken from the real file inspected here: `Make`, `Model`, `DateTimeOriginal`, `ExposureTime`, `FNumber`, `ISO`, `Orientation`, `LensModel`, plus a full GPS sub-directory with latitude, longitude, altitude and a UTC timestamp.
10. The `Orientation` tag is the reason a photo appears rotated in one program and upright in another: the pixels are unrotated and the tag says how to display them.
11. Most large social platforms strip EXIF on upload. Most chat applications strip it when sending a photo as an image and keep it when sending as a file. Cloud storage links and direct email attachments keep it. Treat any file you send as carrying its metadata unless you removed it yourself.
12. Tools: `getfattr`, `setfattr`, `attr` on Linux; `xattr -l` and `mdls` on macOS; `Get-Item -Stream`, `dir /r` and `streams.exe` on Windows; `exiftool` and `exiv2` for image metadata.

■ 20.13.6 WORDS – remember these

1. Extended attribute — a small named label attached to a file — a namespaced name-value pair stored outside the file's data.
2. Quarantine flag — the macOS mark on downloaded files — the `com.apple.quarantine` extended attribute read by Gatekeeper.
3. Resource fork — the classic Macintosh second stream — historically an HFS fork, now the `com.apple.ResourceFork` extended attribute.
4. Alternate data stream — a hidden second set of contents on NTFS — an additional named `$DATA` attribute addressed as `file:stream`.
5. Mark of the web — the Windows record of where a file came from — the `Zone.Identifier` stream, driving Protected View and SmartScreen.
6. EXIF — camera information stored inside a photo — TIFF image file directories in the JPEG APP1 segment, including an optional GPS directory.

20.14 Reading a file you do not understand

■ 20.14.1 PLAIN – in simple words

1. Sooner or later you are handed a file with no extension, or a wrong one, and asked what it is.
2. There is a reliable order of steps, and it takes about a minute.
3. Step one: ask the type-guessing tool. It is right most of the time and costs nothing.
4. Step two: look at the first sixteen bytes yourself. Compare them with the magic number table.
5. Step three: pull out the readable text. Format names, version strings, file paths and error messages usually appear in plain sight.
6. Step four: look at the very end. Many formats keep their index there.
7. Step five: if the file looks like nothing, check whether it is compressed or encrypted, because both look like random noise.
8. Step six: scan the whole file for magic numbers, not just the start, because files are often nested inside other files.
9. Then unwrap one layer and start again from step one.

■ 20.14.2 PLAIN – a picture in your head

1. Think of a parcel with no address label.
2. You shake it. Something rattles. That is the type-guessing tool: fast, rough, usually right.
3. You cut off one corner and peek. That is the hex dump of the first bytes.
4. You read the customs sticker for words you recognize. That is `strings`.
5. You find another wrapped parcel inside, and you start again. Nested formats are the norm, not the exception.

Where this comparison breaks: shaking a parcel gives you a hint about the whole thing at once. A file gives you nothing unless you look in the right place. A 1 GB file whose first bytes are unrecognized may be a perfectly ordinary format that simply begins with a length field, and no amount of staring at the start will tell you. Real identification means checking several places: the start, the end, and a scan across the middle.

■ 20.14.3 PLAIN – a worked example

1. We were handed a file called `mystery.dat` and asked what it was. Here is the whole session.

```
$ ls -l mystery.dat
-rw-r--r-- 1 root root 259 mystery.dat
```

```
$ file mystery.dat
mystery.dat: gzip compressed data, was "inner.tar",
             last modified Thu Aug 13 02:02:54 2026,
             max compression, from Unix,
             original size modulo 2^32 10240

$ xxd -l 16 mystery.dat
00000000: 1f8b 0808 ce25 7d6a 0203 696e 6e65 722e  ....%}j..inner.
```

2. 1f 8b is the gzip magic. 08 is the compression method, Deflate. The fourth byte 08 is a flag meaning “an original filename follows”.
3. That is why the readable text inner. appears at offset 10. The gzip header literally stores the original name.
4. So it is one layer of gzip around something called inner.tar. Unwrap it.

```
$ gzip -dc mystery.dat > mystery.out
$ file mystery.out
mystery.out: POSIX tar archive (GNU)

$ tar tvf mystery.out
-rw-r--r-- root/root 69 2026-08-13 01:55 tiny.png
-rw-r--r-- root/root 26 2026-08-13 01:57 note.txt

$ tar xf mystery.out -C out && file out/*
out/note.txt: ASCII text
out/tiny.png: PNG image data, 1 x 1, 8-bit/color RGB
```

5. Three layers, three commands, one minute. Nothing was guessed.
6. Note that file told us the original name and the uncompressed size without decompressing anything, because gzip stores both in its header and trailer.

■ 20.14.4 PLAIN – what is really happening inside

1. Scanning for embedded formats is worth doing by hand, because tools like binwalk are not always installed.
2. The method is simple: search the whole file for every known magic byte sequence and report the offsets.
3. We wrote a twenty-line script that does exactly this and ran it on a file made by gluing a Linux executable and a ZIP archive together.

```
scanning combo.bin (16516 bytes)
offset      0  ELF object
offset  15960  ZIP local file header
offset  16052  ZIP local file header
offset  16130  ZIP local file header
offset  16260  ZIP central directory
offset  16338  ZIP central directory
offset  16416  ZIP central directory
offset  16494  ZIP end of central dir
```

4. The structure is now completely clear without opening the file in an editor: an executable of 15,960 bytes, then a three-entry ZIP archive.
5. That layout is exactly how self-extracting archives and single-file applications are built, and it is what to look for when a program seems suspiciously large.

- False positives happen. Two-byte signatures like MZ appear by chance roughly once every 65,536 bytes of random data. Always confirm a hit by checking a second field, not just the magic.

■ 20.14.5 TECHNICAL – the engineer’s version

- The identification toolkit, and what each answers:

Tool	Question it answers	Cost
file	what format is this	milliseconds
xxd, od	what are the exact bytes	instant
strings	what text is inside	one pass
binwalk	what is embedded where	one pass
7z l, unzip -l	what is in the archive	fast

- strings by default prints runs of four or more printable characters. Use `-n 8` to cut noise and `-e l` or `-e b` for UTF-16 text, which is where Windows binaries hide most of their readable content.
- Real strings output from the Linux binary we built tells you almost everything about it:

```
$ strings -n 6 hello
/lib64/ld-linux-x86-64.so.2
__libc_start_main
libc.so.6
GLIBC_2.2.5
GLIBC_2.34
hello, world
GCC: (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0
hello.c
```

- From eight lines we learned the interpreter, the library, the minimum glibc version, the program’s output, the exact compiler build, and the source file name. This is why release builds are stripped.
- Practical checks for the “it looks like random noise” case. Compute the byte entropy. Compressed data typically measures 7.9 to 8.0 bits per byte; encrypted data measures the same. Distinguish them by structure: compressed formats have headers, encrypted blobs usually do not.
- Check the length. AES in a block mode produces output that is a multiple of 16 bytes, often plus a fixed header. That is a strong hint.
- If the file is a container you half-recognize, list rather than extract. `unzip -l`, `tar tvf`, `7z l`, `ar t`, `dpkg -I` and `rpm -qip` all read metadata without writing anything to disk.
- For executables, `objdump -x`, `readelf -a`, `nm -D`, `ldd` and `otool -L` on macOS answer what it links against. For unknown binary blobs inside firmware, `binwalk -e` automates the scan-and-extract loop described above.
- Never run an unknown executable to find out what it is. Read it statically first, and if you must run it, do so in a disposable virtual machine with no network.
- When nothing matches, the remaining evidence is structure: look for repeating record lengths, plausible 32-bit little-endian sizes near the start, and offsets that point at other offsets. Most private formats are a header, a count, and an array.

■ 20.14.6 WORDS – remember these

- Hex dump — the raw bytes shown as pairs of hex digits — an offset-addressed listing produced by `xxd`, `od -t x1` or a hex editor.
- strings — the readable text pulled out of a binary — runs of printable characters above a minimum length, in a chosen encoding.

3. Carving — pulling embedded files out of a bigger one — locating known signatures at arbitrary offsets and extracting the ranges between them.
4. Entropy — how random the bytes look — Shannon entropy in bits per byte, near 8.0 for compressed or encrypted data.
5. Nested format — a file inside a file inside a file — layered containers, each requiring its own identification pass.
6. False positive — a magic match that is a coincidence — an accidental byte sequence, more likely with short signatures.

20.98 Common wrong ideas

1. Wrong: the extension tells the computer what a file is. Right: the extension is part of the name. Windows chooses to trust it, Linux tools read the bytes instead, and the file itself is unchanged either way.
2. Wrong: renaming a file converts it. Right: renaming changes a directory entry. Every byte of contents stays exactly the same, as two identical SHA-256 sums in 20.2.3 demonstrated.
3. Wrong: .exe files are a special kind of thing the computer understands natively. Right: an .exe is an ordinary file whose bytes happen to be machine instructions plus a header. The loader does all the work.
4. Wrong: a text file and a binary file are different at the storage level. Right: both are byte sequences. “Text” describes how the bytes happen to render under an encoding, nothing more.
5. Wrong: an empty line ending is just one invisible character everywhere. Right: it is one byte on Linux and macOS, two on Windows, and a different single byte on classic Mac OS, and `wc -l` reported zero lines for a file full of text because of it.
6. Wrong: a DOCX is a Microsoft binary format. Right: it is a ZIP archive of XML files, openable with any unzip tool, as shown in 20.5.3.
7. Wrong: bigger compression level always means slower decompression. Right: for `zstd`, decompression took 0.06 seconds at both level 3 and level 19. Level affects the packer’s search effort, not the unpacker’s work.
8. Wrong: a ZIP is read from the front. Right: correct readers start at the end of central directory record and jump backwards. That is why you can prepend a program to an archive and both still work.
9. Wrong: stripping a binary makes it faster. Right: stripping removes the symbol table, which is never loaded at run time. The program is smaller on disk and identically fast, only harder to debug.
10. Wrong: deleting metadata from a photo before sharing is paranoid. Right: a single command revealed a location accurate to metres, a timestamp and the device model from an ordinary JPEG.

20.99 Chapter summary in 20 lines

1. A file is a named sequence of bytes plus a small metadata record. Nothing in it declares what the bytes mean.
2. Metadata lives in an inode or MFT record: size, permissions, owner, link count and four separate timestamps. The name lives in the directory, not the file.
3. File extensions are a convention inherited from CP/M in 1974, not a rule.
4. Windows decides types from the extension through the registry, macOS from Uniform Type Identifiers introduced in 2005, Linux tools from content sniffing with libmagic.
5. Hiding known extensions by default made `invoice.pdf.exe` display as `invoice.pdf`, and the ILOVEYOU worm of May 2000 used exactly that.
6. Most formats begin with a magic number: 89 PNG for PNG, PK for ZIP, 7f ELF for Linux binaries, MZ for Windows binaries.
7. Java class files and macOS universal binaries share the magic CAFEBABE, a real collision resolved by checking the next field.
8. "Text" means the bytes render as readable characters. Line endings are LF on Unix, CRLF on Windows and network protocols, CR on classic Mac OS, all descended from teleprinter mechanics.
9. A UTF-8 byte order mark is three bytes, EF BB BF, permitted but not recommended, and it breaks shebangs, shell scripts and strict JSON parsers.
10. ZIP stores local headers, then the data, then a central directory, then an end record. Readers start at the end, which is why appending and prepending both work.
11. DOCX, XLSX, PPTX, ODT, JAR, APK, EPUB and XPI are all ZIP archives with an agreed internal layout.
12. PDF is a body of numbered objects plus a cross-reference table of byte offsets and a trailer, read from the end, supporting incremental update.
13. An executable is machine code plus a header describing segments, permissions and one entry point. PE on Windows, ELF on Linux, Mach-O on macOS.
14. Every PE still begins with an MZ header and a DOS stub printing "This program cannot be run in DOS mode", because offset 0x3C inside it holds the pointer to the real header.
15. ELF carries two tables over the same bytes: program headers for the loader and section headers for the linker. `strip` deletes the second and the program runs identically.
16. Running a program is thirty steps: resolve, permission-check, `execve`, match the format, build a fresh address space, map segments, build the stack with arguments and the auxiliary vector, load `ld.so`, resolve libraries and relocations, then `_start`, then `__libc_start_main`, then `main`.
17. A `.sh` or `.py` file runs because of two bytes, `#!`, handled by `binfmt_script`, which re-execs the named interpreter with your file as an argument. Without them, `execve` returns `ENOEXEC`.
18. Installers copy files, register types, create shortcuts, write settings and record uninstall data. Package managers do the same with an exact file list, which is why removal is clean.
19. `tar` archives without compressing and `gzip` compresses without archiving; combining them gives solid compression, which measured 38 percent smaller than the same algorithm applied per file.
20. Files carry more than you see: extended attributes, macOS quarantine flags, NTFS alternate data streams, and EXIF blocks that can hold your exact location. `file`, `xxd`, `strings` and a signature scan will identify almost anything in about a minute.



PART E

Seeing and Showing

One key press traced from your finger to the light in your eye, and how the picture is computed.

From Key Press to Pixel — The Complete Trace

21.0 What this chapter gives you

1. You will be able to follow one key press, the letter A, from your fingertip to the light entering your eye, without a single gap in the story.
2. You will be able to name every piece of hardware and every piece of software that touches that press, in order, and say what each one does to it.
3. You will be able to explain why a switch that closes once produces several electrical changes, and what the machine does about that.
4. You will be able to read a raw USB keyboard report by hand and say which keys are down.
5. You will be able to explain why the keyboard does not know what letter you pressed, and where the letter is actually decided.
6. You will be able to describe what a window system does with an event and why the same key goes to a different program depending on where you clicked.
7. You will be able to give a real time budget for the whole chain, stage by stage, and say which stage is costing you the most.
8. You will be able to do the same trace for a mouse click and a touchscreen tap and say exactly what is different.
9. You will be able to argue honestly about input lag, with numbers, instead of with feelings.

21.1 The finger and the switch

■ 21.1.1 PLAIN - in simple words

1. Under every key on your keyboard there is a switch.
2. A switch is a thing that joins two pieces of metal when you push it, and separates them when you let go.
3. When the metal pieces touch, electricity can flow through. When they are apart, it cannot.
4. That is the whole job of the key. Nothing more. It is a gate for electricity.
5. The key does not know it is the letter A. It has no idea. It only knows “touching” or “not touching”.
6. Different keyboards use different ways of making that touch happen, and they feel very different under the finger.
7. The cheapest way uses a small rubber dome that squashes flat. The most expensive ways use a spring and a sliding plastic part, or a magnet, or a beam of light.
8. All of them are trying to solve the same problem: turn a finger moving a couple of millimetres into a clean electrical signal.

■ 21.1.2 PLAIN – a picture in your head

1. Think of a garden gate with a spring on it.
2. You push the gate, it swings open, and when you let go the spring pulls it shut again.
3. Now imagine the gate has a metal latch that touches a metal post when the gate reaches a certain point.
4. You do not have to push the gate all the way to the wall for the latch to touch. It touches partway through the swing.
5. That partway point is what engineers call the actuation point: the exact moment the key counts as pressed.
6. Now imagine the gate is light and springy. When the latch first hits the post it does not stop dead. It taps, bounces off a fraction of a millimetre, taps again, and only then settles.
7. Anyone watching only the latch would see it touch, break, touch, break, touch, and finally stay touching. All from one push of the gate.
8. That tapping is exactly what a key switch does, and it is the first real problem in this whole chapter.

Where this comparison breaks: a real gate bounces over tens of milliseconds and you can see it. A key switch bounces over a handful of milliseconds and the movements are microscopic. Also, a gate has one latch, while some switch types have no moving contact at all: an optical switch just blocks a light beam, and a Hall-effect switch just moves a magnet past a sensor. Those types cannot bounce mechanically, because there is nothing to bounce.

■ 21.1.3 PLAIN – a worked example

1. Take a Cherry MX Red switch, one of the most common mechanical switches sold.
2. Its published numbers, from the Cherry datasheet, are these.

Property	Value
Actuation force	45 cN (about 45 g)
Pre-travel to actuation	2.0 mm
Total travel	4.0 mm
Initial force	30 cN minimum
Bounce time	under 5 ms

3. So the key sits at rest. You start pushing.
4. After 2.0 mm of downward movement, the sliding plastic part has let the metal contact leaf spring back into contact. The circuit closes.
5. You are still moving. There are 2.0 mm left before the key hits the bottom. Those 2.0 mm change nothing electrically.
6. This is important. The key fires at 2.0 mm, not at the bottom. If you type by slamming keys to the bottom, you are wasting 2.0 mm of your own time.
7. At the moment of contact, the metal leaf vibrates. For under 5 milliseconds the connection flickers on and off.
8. The keyboard's own chip therefore sees something like this over those few milliseconds.

```
time (ms):  0.0  0.4  0.7  1.1  1.5  2.0  2.4  3.0  ...  5.0
contact:    off  ON   off  ON   off  ON   ON   ON    ON
```

9. If the keyboard reported every one of those changes, you would get “AAAA” from one press.
10. So the firmware waits. It only accepts the change once the signal has held steady for a set period. That waiting is called debouncing.

■ 21.1.4 PLAIN – what is really happening inside

1. Let us go through the four main switch families, because they behave differently and cost you different amounts of time.
2. **Rubber dome / membrane.** Three flexible plastic sheets are stacked. The top and bottom sheets have printed conductive traces. The middle sheet has holes.
3. A moulded rubber dome sits over each key. Pressing collapses the dome, which pushes the top sheet through the hole in the middle sheet, so the top trace touches the bottom trace.
4. The collapse is sudden, which is why a cheap keyboard has a soft top and then a sharp give. Actuation and bottoming out are almost the same moment, typically at 3.5 to 4 mm.
5. **Mechanical.** Each key has its own sealed plastic housing with a spring, a sliding stem, and a springy metal contact leaf.
6. The stem has a ramp cut into it. As the stem descends, the ramp lets the leaf spring back into contact with a fixed metal terminal.
7. Because the contact is a springy metal blade hitting a metal terminal, it bounces. Cherry specifies under 5 ms of bounce.
8. **Scissor.** Used on laptops and thin keyboards. A rubber dome still does the electrical work, but two plastic arms crossed like scissors hold the keycap level.
9. That means the key works even if you press a corner, and it allows travel of roughly 1 to 2 mm instead of 4 mm. Apple's butterfly design of 2015 pushed this to about 0.5 mm and was abandoned by 2021 because of reliability complaints.
10. **Optical.** The stem carries a small flag. At rest the flag blocks an infrared beam between an emitter and a photo-transistor. Pressing lifts or lowers the flag and the beam changes.
11. There is no metal-to-metal contact, so there is no contact bounce. The signal still has to cross a threshold, and comparators near their threshold can chatter, so firmware still applies a small filter.
12. **Hall effect / magnetic.** A magnet in the stem moves past a sensor that measures magnetic field strength. The output is not on or off. It is a number that says how far down the key is.
13. That means the actuation point is a software setting. Wooting's Lekker switch is specified at 40 gf actuation, 60 gf bottom out, 4.0 mm total travel, and an actuation point adjustable anywhere from 0.1 mm to 4.0 mm.
14. It also allows "rapid trigger": the key releases the moment you start moving up, rather than waiting to cross a fixed point. That is a firmware behaviour, not a physical one.

■ 21.1.5 TECHNICAL – the engineer's version

1. Contact bounce is a mechanical resonance of the moving contact against the fixed contact, with the contact modelled as a damped mass-spring system. Bounce duration scales with contact mass and inversely with damping.
2. Cherry's MX1A-Lxx datasheet, which covers MX Red, specifies bounce time under 5 ms measured during actuation at a closing speed of 0.4 m/s. The MX2A revision keeps the same 45 cN, 2.0 mm, 4.0 mm figures.
3. Typical debounce strategies in keyboard firmware:

Strategy	How it works	Added latency
Eager / asym	Fire on first edge	~0 ms on press
Defer / symmetric	Wait N stable scans	3 to 8 ms
Per-key timer	5 ms lockout per key	up to 5 ms
Integrator	Count up/down to limit	2 to 6 ms

Strategy	How it works	Added latency

4. QMK, the widely used open-source keyboard firmware, ships DEBOUNCE defaulting to 5 milliseconds, with selectable algorithms named `sym_defer_g`, `sym_eager_pk`, `asym_eager_defer_pk` and others. The eager variants report the press immediately and then lock the key out, giving zero added press latency at the cost of possible false presses on a dirty contact.
5. Actuation force is quoted in centinewtons (cN) by Cherry and in grams-force (gf) by most others. 1 cN is 0.01 N; 1 gf is 0.00980665 N. So 45 cN is about 45.9 gf. Treating cN and gf as interchangeable introduces about 2 percent error, which is inside the manufacturing tolerance anyway.
6. Rated lifetime: Cherry MX2A claims 100 million actuations per key. Rubber domes are typically rated 5 to 20 million. Buckling spring keyboards, the IBM Model M design of 1985, are commonly rated at 25 million and many are still working after forty years.
7. IBM patented the buckling spring mechanism in 1978. In it, a coil spring under compression buckles sideways, and the buckling motion flips a hammer that strikes a capacitive sense pad on the PCB. Actuation and the audible click are the same event, which is why the feedback is honest, unlike a clicky mechanical switch where the click and the electrical actuation are produced by different parts and can drift apart.
8. Optical switches used in production include Razer's opto-mechanical design (2018) and Bloody's LK Libra series. Vendors claim 0.2 ms actuation latency. That is a **marketing claim** about the sensor alone and does not include scan, debounce filtering, or USB, which dominate.
9. Hall-effect switches use a linear Hall sensor such as the Allegro A1304 or equivalent, sampled by the microcontroller's ADC. Resolution of the reported depth is limited by ADC bits and calibration drift, not by the magnet.
10. Tools: an oscilloscope across the switch terminals shows the bounce directly. On a keyboard you control, a logic analyser on the matrix pins shows the scan and the debounce window together.

■ 21.1.6 WORDS – remember these

1. Switch — the thing under a key that joins two wires — an electromechanical or non-contact single-pole momentary contact.
2. Actuation point — how far down the key must go before it counts — pre-travel distance at which the contact closes, typically 2.0 mm on Cherry MX.
3. Travel — total distance the key can move — total actuator travel, typically 4.0 mm full size, 1 to 2 mm scissor.
4. Actuation force — how hard you must press — operating force in cN or gf, typically 45 to 60.
5. Bounce — the flickering of the contact when it first touches — contact chatter caused by mechanical resonance, under 5 ms on Cherry MX.
6. Debounce — waiting until the flickering stops — firmware filtering of contact chatter, typically a 5 ms stability window.
7. Hall effect — sensing a magnet instead of touching metal — a linear magnetic field sensor giving continuous key depth rather than a binary state.

21.2 The key matrix

■ 21.2.1 PLAIN – in simple words

1. A full-size keyboard has 104 keys. It does not have 104 wires going to the chip inside.
2. That would need 104 pins on the chip plus a common return, and cheap chips do not have that many pins.
3. Instead the keys are arranged in a grid, like squares on graph paper.
4. There are wires running across, called rows, and wires running down, called columns.
5. Each key sits where one row crosses one column, and when pressed it joins that row to that column.
6. With 8 rows and 18 columns you get 144 crossing points, enough for 104 keys, using only 26 wires.
7. The chip cannot look at all crossings at once. So it checks them in turn, very fast, over and over.
8. It energizes one row, reads all the columns, then moves to the next row. Doing all rows once is called a scan.
9. A good keyboard does a complete scan a thousand times a second. A cheap one might do it a hundred times a second.

■ 21.2.2 PLAIN – a picture in your head

1. Think of a block of flats with a caretaker who has to know which flats have someone at home.
2. There are 8 floors and 18 flats on each floor. The caretaker cannot see all 144 flats at once.
3. So the caretaker switches on the light in the corridor of floor 1 only, then walks along and looks at which doorways show light spilling out.
4. Then floor 1's light goes off, floor 2's comes on, and the same check happens. And so on to floor 8.
5. One full trip through all 8 floors is one scan. If the caretaker can do it a thousand times a second, nothing gets missed for more than a millisecond.
6. Now the problem. Suppose three flats are occupied and they happen to form three corners of a rectangle. Light can leak from floor to floor through the open doors and make the fourth corner look occupied too.
7. That false fourth key is called ghosting, and it is a real thing that happens on cheap keyboards.
8. The fix is a one-way door on every flat: light can go out but not back in. In electronics, a one-way door is a diode.

Where this comparison breaks: the caretaker is slow and the keyboard is not. A scan of the whole matrix takes tens of microseconds of actual work; the rest of the millisecond is the chip waiting on purpose. And the “light leaking” is not light, it is a current finding a path backwards through two other closed switches.

■ 21.2.3 PLAIN – a worked example

1. Here is a tiny 3x3 matrix. Rows R0 to R2 run across. Columns C0 to C2 run down. A key is at every crossing.

	C0	C1	C2	
R0	-----+-----+-----+-----			
R1	-----+---Q---+---W---+-----			Q at R1/C0, W at R1/C1
R2	-----+---A---+---S---+-----			A at R2/C0, S at R2/C1

2. The chip drives R0 low and reads C0, C1, C2. All read high, so no key on row 0 is down.

3. The chip drives R1 low. C0 reads low, so Q is down. C1 and C2 read high.
4. The chip drives R2 low. C0 reads low, so A is down.
5. Now hold three keys at once: Q (R1/C0), W (R1/C1) and A (R2/C0).
6. Drive R2 low. Current goes out of R2, through A, up C0, back through Q into R1, then out through W into C1.
7. So C1 reads low as well, and the chip believes S (R2/C1) is pressed. It was not. That is a ghost key.
8. Put a diode in series with every switch, all facing the same way. Now current can go from row to column but never from column back to row.
9. The path through Q backwards is blocked. No ghost. Every combination of keys is now readable. That property is called N-key rollover.
10. Some cheap keyboards do not add diodes. Instead the firmware detects the rectangle pattern and refuses to report anything, which is called masking or blocking. You press three keys and get two. That is safer than a ghost and cheaper than 104 diodes.

■ 21.2.4 PLAIN – what is really happening inside

1. The chip's row pins are outputs. The column pins are inputs with pull-up resistors, meaning each column reads high unless something pulls it down.
2. To scan a row, the chip sets that one row pin low and leaves all the others floating or high.
3. It then waits a short settle time, typically a few microseconds, because the wires have capacitance and the voltage does not change instantly.
4. It reads all the column pins in one operation, usually a single register read, giving 18 bits at once.
5. It stores those bits in a row of a bitmap in memory. After 8 rows it has the full state of all 144 crossings.
6. It compares that bitmap with the previous one. Any bit that changed is a candidate key event.
7. Candidates go into the debounce filter. Only after a candidate has held its new value for the debounce window does it become a real event.
8. Real events go into a queue for the next stage, which is turning a matrix position into a code.
9. The scan rate is a choice. Scanning faster costs power and leaves less CPU time for lighting effects and macros. Scanning slower adds latency.
10. On average, waiting for the scan costs you half a scan period, because your press lands at a random point in the cycle. At 1000 Hz that average is 0.5 ms. At 100 Hz it is 5 ms.

■ 21.2.5 TECHNICAL – the engineer's version

1. A typical full-size layout uses an 8 x 18 or 6 x 21 matrix. Pin count is rows plus columns, so 26 or 27 general-purpose I/O pins.
2. Diodes are small-signal types, commonly 1N4148 in through-hole builds or BAV70 / 1N4148W in surface mount, chosen for low forward voltage (about 0.7 V) and fast recovery.
3. Orientation is a convention, and both exist. COL2ROW means the diode cathode faces the row; ROW2COL is the reverse. QMK has a compile-time DIODE_DIRECTION setting for exactly this reason.
4. Scan rates in real firmware:

Board / firmware	Scan rate	Note
Apple IIe (AY-3600)	~556 Hz	1978 dedicated chip
Ergodox (original fw)	~167 Hz	claimed by project
QMK on ARM, typical	1000+ Hz	often 2 to 5 kHz
Cheap membrane board	100 to 200 Hz	common, not specified

5. Dan Luu’s 2017 analysis computed the Apple IIe’s General Instrument AY-3600 keyboard encoder at a 1.8 ms scan delay and a 6.8 ms debounce delay, under 8.6 ms of internal keyboard logic, using the RC values on the board.
6. N-key rollover over USB requires more than diodes. The 8-byte boot report carries only six key codes. Full NKRO needs a report-protocol descriptor with a bitmap of usages, and most NKRO keyboards expose two HID interfaces: a boot-compatible one for BIOS use and an NKRO one for the operating system.
7. Ghosting terminology: strictly, **ghosting** is a phantom key appearing; **masking** or **jamming** is a real key being suppressed. Consumer marketing uses “anti-ghosting” for both, which is imprecise.
8. Matrix settle time matters at high scan rates. With long traces and no series resistors, a 1 microsecond settle can produce false reads; QMK exposes DEBOUNCE, MATRIX_I0_DELAY (default 30 microseconds) and DIRECT_PINS for keyboards small enough to skip the matrix entirely.
9. Hall-effect boards do not have a conventional matrix scan of digital states. They multiplex analogue sensors into an ADC. The limiting rate is ADC conversion time times the number of keys, which is why 8 kHz analogue boards are recent and expensive.
10. Tools: `sudo evtest`, `wex` on Wayland or `xev` on X11 will show whether keys are being lost. A logic analyser on the row pins shows the scan period directly.

■ 21.2.6 WORDS – remember these

1. Matrix — a grid of wires so a few pins can read many keys — an $m \times n$ row/column arrangement needing $m+n$ GPIO pins for $m*n$ keys.
2. Scan — one full pass over all rows — the periodic sequential strobe and sample loop, typically 100 Hz to 5 kHz.
3. Ghosting — a key that appears pressed but is not — a phantom detection from a sneak current path through three closed switches.
4. Masking — keys deliberately dropped to avoid ghosts — jamming, the firmware suppressing an ambiguous rectangle pattern.
5. Diode — a one-way valve for current — a small-signal rectifier per key blocking the reverse sneak path, giving NKRO.
6. N-key rollover — every key readable at once — unlimited simultaneous key detection, requiring both per-key diodes and a suitable HID report format.

21.3 The keyboard’s own computer

■ 21.3.1 PLAIN – in simple words

1. Inside your keyboard there is a small computer. A real one, with a processor, memory and a program.
2. It is not powerful. It might run at 16 to 100 million cycles a second, with a few kilobytes of memory. Your phone is a hundred thousand times bigger.
3. Its whole job is: scan the grid, clean up the signal, decide what changed, and tell the host machine.
4. When it decides that the key at row 2, column 0 has gone down, it does not send “row 2, column 0”. It looks up a table.
5. The table says: this position on this keyboard corresponds to code number such-and-such.
6. That number is called a scan code, or in USB language a usage code.
7. Here is the part people get wrong. That number is not a letter. It is not the letter A.
8. It is a name for a physical position on the keyboard. “The key that is one to the right of Caps Lock”, roughly.

9. If you switch your computer to a French layout, that same key produces Q on screen. The keyboard sent the identical number both times.
10. So the keyboard does not know your language, your layout, or what you typed. It only knows which physical key moved.

■ 21.3.2 PLAIN – a picture in your head

1. Imagine a hotel with numbered rooms and a bell board behind the reception desk.
2. When someone in room 214 presses their bell, a flag drops on the board labelled “214”.
3. The board does not say “Mrs Patel wants tea”. It says “214”.
4. Reception has a separate list mapping rooms to guests. Guests change every week. The bell board never changes.
5. The bell board is the keyboard. The room number is the scan code. The guest list is your keyboard layout, and it lives at reception, not in the wiring.
6. This split is why a hotel can rewire nothing when new guests arrive, and why your computer can switch between English, French and Devanagari instantly without you touching the keyboard.

Where this comparison breaks: the hotel board reports only “someone pressed”. The keyboard reports both press and release, and reports the full set of keys currently held, so the computer can know that Shift and A are down at the same moment. Also, some keyboards do relabel themselves in firmware, which would be like the bell board printing guest names. That is a firmware remap, and it is the exception, not how layouts normally work.

■ 21.3.3 PLAIN – a worked example

1. You press the A key on a standard 104-key board wired to a US layout.
2. The firmware’s key map is an array. Position [row 2][column 0] holds the value KC_A, which in QMK is the number 0x04.
3. That 0x04 is the USB HID usage code for “Keyboard a and A” on Usage Page 0x07.
4. The name in the specification is literally “Keyboard a and A”. The specification refuses to pick one, because that choice is not the keyboard’s to make.
5. Now compare the three older PS/2 scan code sets for the same physical key.

System	Make code for A	Break code
Set 1 (XT, 1981)	0x1E	0x9E
Set 2 (AT, 1984)	0x1C	0xF0 0x1C
Set 3 (PS/2, 1987)	0x1C	0xF0 0x1C
USB HID (1996 on)	0x04	absent from report

6. Four different numbers for one physical key, across four standards. None of them is the ASCII code for A, which is 0x41, or for a, which is 0x61.
7. Notice the last row. USB does not send a release code at all. It sends the full list of keys currently held down. A key disappearing from the list is the release.
8. Now hold Shift and press A. The firmware does not combine them. Shift is reported as a separate bit in a separate byte, and A is still 0x04.
9. Combining them into a capital letter is somebody else’s job, several stages later.

■ 21.3.4 PLAIN – what is really happening inside

1. The microcontroller runs a loop, forever, in this order.
2. Scan the matrix into a bitmap. Compare with the last bitmap. Feed changes into the debounce filter.
3. For each debounced change, look up the key map for the current layer.
4. Layers are a firmware idea. Holding a Fn key switches to a second key map array, so the same physical key gives a different code. This is how a 60-key board offers arrow keys.
5. Sort the resulting codes into modifiers and normal keys. The eight modifiers, left and right Ctrl, Shift, Alt and GUI, go into a single byte as bits.
6. Normal keys go into a list, in the order they were pressed, oldest first.
7. Build the report. Byte 0 is the modifier bits. Byte 1 is reserved and always zero. Bytes 2 to 7 are up to six normal key codes.
8. Hand the report to the USB stack, which puts it in a buffer attached to an endpoint and waits for the host to ask.
9. If more than six normal keys are held, the boot protocol has no room. The specification says to fill all six slots with 0x01, the code named ErrorRollOver.
10. There is no auto-repeat in the keyboard. Hold a key and the keyboard just keeps reporting the same list. The repeating you see when you hold a key is generated by the operating system with a timer.

■ 21.3.5 TECHNICAL – the engineer’s version

1. Common keyboard microcontrollers: Atmel ATmega32U4 (8-bit AVR, 16 MHz, 2.5 KB SRAM, native USB) in older custom boards; STM32F072/F103 and RP2040 in newer ones; proprietary parts from Holtek, SONiX and Nuvoton in mass-market boards.
2. The USB HID Usage Tables document defines Usage Page 0x07 as the Keyboard/ Keypad page. Selected values:

Usage ID	Name in the specification
0x00	Reserved (no event)
0x01	ErrorRollOver
0x04	Keyboard a and A
0x1D	Keyboard z and Z
0x28	Keyboard Return (ENTER)
0xE1	Keyboard LeftShift

3. Note 0xE1. Left Shift has a usage ID, but in the boot report it is carried as bit 1 of byte 0 rather than as a key code. Both representations exist and report-protocol descriptors normally declare the modifier byte as a bitmap of usages 0xE0 to 0xE7.
4. PS/2 scan code set 2 is what AT-class keyboards actually send on the wire. The 8042 keyboard controller on the motherboard translates set 2 into set 1 before the operating system sees it, when translation is enabled. That translation is why so much documentation disagrees about which set is “the” scan code set. Set 3 was defined for the IBM 3270 terminal style and is rarely used, though it has the useful property that every key can be individually configured as make-only, make-break or typematic.
5. USB was published as version 1.0 in January 1996 by a group including Intel, Microsoft, Compaq, IBM, DEC, NEC and Nortel. The HID class specification version 1.11 dates from 2001 and is still the current document.

6. Legacy support: the BIOS or UEFI cannot run a full USB stack early in boot, so the boot protocol exists. `SET_PROTOCOL` with value 0 selects boot protocol, value 1 selects report protocol. Firmware also historically used SMM-based USB legacy emulation to make a USB keyboard appear at I/O ports 0x60 and 0x64 like an 8042.
7. Firmware options in wide use: QMK (2015 on, from Jun Wako's TMK), ZMK for wireless, VIA and Vial for runtime remapping, and closed vendor firmware. QMK's `keymap.c` is a C array of arrays, one per layer.
8. Latency inside the microcontroller for the lookup and report build is small: on a 16 MHz AVR, a full 8x18 scan plus debounce plus report build measures in the tens of microseconds, so under 0.1 ms. The scan period, not the computation, is the cost.
9. Where the layout is NOT decided, said plainly: not in the switch, not in the matrix, not in the scan code table, and not in the USB report. All of those carry physical position only.

■ 21.3.6 WORDS – remember these

1. Microcontroller — the tiny computer in the keyboard — an MCU with CPU, flash, SRAM and a USB device peripheral, typically 8 to 100 MHz.
2. Scan code — a number naming a physical key — a make/break code from PS/2 sets 1, 2 or 3, or a HID usage ID on Usage Page 0x07.
3. Usage code — the USB name for a key — a 16-bit usage page plus usage ID pair identifying a control, defined in the HID Usage Tables.
4. Modifier — Shift, Ctrl, Alt, Cmd — usages 0xE0 to 0xE7, carried as a bitmap in byte 0 of the boot keyboard report.
5. Layer — a second key map reached by holding a key — a firmware-level alternate keycode array, selected by momentary or toggle actions.
6. ErrorRollOver — the code meaning “too many keys” — usage 0x01 placed in all six slots when more keys are held than the report can carry.

21.4 Getting the data out: USB HID

■ 21.4.1 PLAIN – in simple words

1. The keyboard now has an 8-byte message ready. It has to get it to the computer.
2. USB does not let a device speak whenever it wants. The computer is in charge of the wire, always.
3. The keyboard cannot interrupt. It can only put its message in a small mailbox and wait to be asked.
4. The computer asks on a fixed schedule. “Anything for me?” Over and over, forever, whether you are typing or not.
5. If the mailbox is empty, the keyboard answers with a tiny “nothing”, which costs almost no time on the wire.
6. If the mailbox has the message, the keyboard hands it over and the computer acknowledges it.
7. The schedule is normally once every millisecond. Some fast devices are asked every 125 microseconds, which is eight times a millisecond.
8. So on average your key press waits half of one asking-interval before it can leave the keyboard.
9. That mailbox has a name: an endpoint. The kind of asking is called an interrupt transfer, which is a confusing name because nothing is interrupted. The device is polled.

■ 21.4.2 PLAIN – a picture in your head

1. Think of a postal round in a village with one postman and many houses.
2. The postman walks the same route every minute, stopping at every house that has asked to be visited.

3. At each stop the postman says “anything to send?” The householder either hands over a letter or says “nothing today”.
4. A householder cannot run down the street and shove a letter into the postman’s bag out of turn. They must wait for the visit.
5. If you write a letter just after the postman leaves, it waits nearly a full minute. If you write it just before he arrives, it goes almost at once.
6. On average it waits half a minute. That is exactly the average USB polling delay.
7. A house can ask to be visited more often, and the postman will agree if the route has room. That is what setting a shorter polling interval means.

Where this comparison breaks: the postman visits in a fixed order, whereas USB visits in a schedule computed by the host controller that can be reordered and that reserves bandwidth in advance. And the postman carries the letter away; USB copies the data and leaves the device’s buffer to be refilled.

■ 21.4.3 PLAIN – a worked example

1. Here is the exact 8-byte boot keyboard report format.

```
byte 0 : modifier bits
byte 1 : reserved, always 0x00
byte 2 : key code slot 1
byte 3 : key code slot 2
byte 4 : key code slot 3
byte 5 : key code slot 4
byte 6 : key code slot 5
byte 7 : key code slot 6
```

2. The modifier byte, bit by bit, lowest bit first:

```
bit 0 : Left  Ctrl      bit 4 : Right Ctrl
bit 1 : Left  Shift     bit 5 : Right Shift
bit 2 : Left  Alt       bit 6 : Right Alt
bit 3 : Left  GUI       bit 7 : Right GUI
```

3. You press A alone. The report on the wire is:

```
| 00 00 04 00 00 00 00 00
```

4. Read it: no modifiers, reserved zero, key code 0x04 in slot 1, nothing else.
5. You now hold Left Shift and press A. Two reports go out, because Shift lands first.

```
| 02 00 00 00 00 00 00 00    Left Shift down, no keys
| 02 00 04 00 00 00 00 00    Left Shift down, A down
```

6. Byte 0 is 0x02, which is binary 00000010, so bit 1 is set, which is Left Shift. Byte 2 is 0x04, which is the A key.
7. Notice what is not there. There is no capital A anywhere. There is no character at all. Two independent physical facts: a modifier is held, and a key position is down.
8. You release A but keep Shift. The report becomes 02 00 00 00 00 00 00 00 again. The disappearance of 0x04 is the release event.
9. You release Shift. The report becomes all zeros.
10. Now type A, then B without releasing A, then C without releasing either:

```
| 00 00 04 00 00 00 00 00    A
| 00 00 04 05 00 00 00 00    A then B
```



```

| 00 00 04 05 06 00 00 00    A then B then C
| 00 00 05 06 00 00 00 00    A released, B and C shuffle down

```

11. Press more than six normal keys at once and the boot report cannot carry them, so all six slots become 0x01:

```

| 20 00 01 01 01 01 01 01    Right Shift held, rollover error

```

■ 21.4.4 PLAIN – what is really happening inside

1. When you plug the keyboard in, the host asks it to describe itself. This is called enumeration.
2. The device returns a device descriptor, a configuration descriptor, one or more interface descriptors, endpoint descriptors and a HID report descriptor.
3. The interface descriptor says class 3, which means HID. Subclass 1 means it supports the boot protocol. Protocol 1 means keyboard, protocol 2 means mouse.
4. One of the endpoint descriptors says “interrupt IN”, meaning data flows device to host on a polled schedule. It carries a field called `bInterval`.
5. `bInterval` sets how often the host asks. On a full-speed device it is a number of 1 ms frames, from 1 to 255. On a high-speed device it is an exponent: the period is 2 to the power of (`bInterval` minus 1), counted in 125 microsecond microframes.
6. So the fastest possible polling is 125 microseconds, which is 8000 times a second, and that requires a high-speed connection.
7. Once running, the host controller issues an IN token to the device’s address and endpoint number at each scheduled slot.
8. If the keyboard has a new report, it replies with a DATA0 or DATA1 packet containing the 8 bytes, and the host replies ACK.
9. If nothing changed, the keyboard replies NAK, which is 3 bytes on the wire and takes under a microsecond.
10. Bluetooth keyboards do none of this. They use HID over GATT, a profile on Bluetooth Low Energy, where reports become notifications on a GATT characteristic.
11. Bluetooth Low Energy has its own polling clock called the connection interval, and it is much coarser. That is the single biggest reason wireless keyboards feel slower.

■ 21.4.5 TECHNICAL – the engineer’s version

1. USB speeds and their interrupt endpoint limits:

Speed	Bit rate	Min interrupt period
Low speed	1.5 Mbit/s	10 ms (<code>bInterval</code> 10)
Full speed	12 Mbit/s	1 ms (<code>bInterval</code> 1)
High speed	480 Mbit/s	125 us (<code>bInterval</code> 1)
SuperSpeed	5 Gbit/s	125 us (<code>bInterval</code> 1)

2. Per the USB 2.0 specification, section 9.6.6, a low-speed interrupt endpoint must declare `bInterval` between 10 and 255 in 1 ms frames; a full-speed one between 1 and 255 in 1 ms frames; and a high-speed one between 1 and 16 as an exponent, giving a period of $2^{(bInterval-1)}$ microframes of 125 us.
3. Most plain USB keyboards enumerate as low speed or full speed. Dan Luu’s 2017 measurements list many as “USB FS”. A low-speed keyboard with `bInterval` 10 is polled 100 times a second and adds an average of 5 ms.

4. Gaming peripherals advertising 1000 Hz are full speed with bInterval 1. Those advertising 4000 Hz or 8000 Hz must be high speed with bInterval 1 or 2, because full speed physically cannot go below a 1 ms frame.
5. The report descriptor for a boot-compatible keyboard is a well-known 63-byte blob. Its meaning, decoded:

```

Usage Page (Generic Desktop), Usage (Keyboard)
Collection (Application)
  Usage Page (Keyboard/Keypad)
  Usage Min (0xE0) Usage Max (0xE7)      modifiers
  Logical Min (0) Logical Max (1)
  Report Size (1) Report Count (8)
  Input (Data, Variable, Absolute)      -> byte 0
  Report Size (8) Report Count (1)
  Input (Constant)                      -> byte 1 reserved
  Report Size (8) Report Count (6)
  Logical Min (0) Logical Max (101)
  Usage Min (0) Usage Max (101)
  Input (Data, Array)                   -> bytes 2..7
End Collection

```

6. Note the difference in the two Input items. The modifier byte is *Variable*, meaning each bit is its own control. The six key slots are *Array*, meaning each byte holds the index of whichever control is active. That single distinction is why modifiers are a bitmap and keys are a list.
7. NKRO keyboards add a second report, or a second interface, declaring a *Variable* bitmap over usages 0x00 to 0xF7, which is 31 bytes. Every key then has its own bit and there is no six-key limit.
8. Bluetooth: HID over GATT Profile (HOGP) 1.0 was adopted by the Bluetooth SIG in 2011. Connection interval is negotiated in units of 1.25 ms, with a permitted range of 7.5 ms to 4 s. Apple's accessory guidelines historically asked for intervals no shorter than 15 ms for most accessories, with exceptions for HID. Vendors using proprietary 2.4 GHz dongles, such as Logitech Lightspeed or Razer HyperSpeed, bypass Bluetooth entirely and can hold a 1 ms interval.
9. Tools: `lsusb -v` on Linux prints bInterval and the full descriptor tree. `usbhid-dump` prints report descriptors. On Windows, USB Device Tree Viewer shows the same. Wireshark with `usbmon` captures the actual reports.
10. Honest note on 8000 Hz polling: it removes at most about 0.44 ms of average wait compared with 1000 Hz. It also multiplies host interrupt load by eight. Whether that trade is worth it is contested, and blind tests have not shown a consistent human benefit above roughly 1000 Hz.

■ 21.4.6 WORDS - remember these

1. Endpoint — a mailbox on the device — a uniquely addressed source or sink of data, identified by number and direction.
2. Interrupt transfer — the host asking on a timer — a guaranteed-bandwidth, bounded-latency polled transfer type, despite the name.
3. bInterval — how often the host asks — the polling interval field of the endpoint descriptor, in frames or as a power-of-two microframe exponent.
4. Boot protocol — the simple fixed 8-byte format — the reduced report format defined by HID subclass 1 so firmware can drive a keyboard without a parser.
5. Report descriptor — the device explaining its own data format — a self-describing byte stream of HID items parsed into a control tree.
6. NAK — “nothing to send” — a handshake packet meaning the endpoint has no data ready, costing about 3 bytes of bus time.

7. HOGP — Bluetooth’s version of HID — HID over GATT Profile, carrying reports as BLE notifications on a negotiated connection interval.

21.5 Into the machine: the host controller

■ 21.5.1 PLAIN – in simple words

1. The 8 bytes have left the keyboard and travelled down the cable. They arrive at a chip on the motherboard called the host controller.
2. The host controller is the thing that actually runs the USB schedule. It is the postman from the earlier picture.
3. It is not the CPU. It is a separate piece of hardware that works while the CPU is doing other things.
4. The operating system has already told the host controller, in advance, where in memory to put anything that arrives from this keyboard.
5. So when the bytes arrive, the host controller writes them straight into main memory itself, without asking the CPU.
6. Hardware writing to memory on its own is called DMA, direct memory access.
7. Then the host controller writes a small note into another area of memory saying “a transfer finished, here is what happened”.
8. Only then does it poke the CPU, by sending a special message that makes the CPU stop what it is doing and run some kernel code.
9. That poke is an interrupt. This time the word is accurate: something really is interrupted.

■ 21.5.2 PLAIN – a picture in your head

1. Think of a busy kitchen with a chef and a delivery hatch.
2. The chef does not stand at the hatch waiting for deliveries. That would waste the whole day.
3. Instead there are labelled shelves. The chef has already written a list: “onions go on shelf 3, flour goes on shelf 7”.
4. The delivery person puts the goods on the right shelf without speaking to the chef at all. That is DMA.
5. Then the delivery person writes a line in a notebook by the door: “onions, shelf 3, 10:42, all fine”. That notebook is the event ring.
6. Finally, they ring a bell. The chef stops chopping, wipes their hands, reads the notebook, and deals with whatever arrived.
7. The bell is the interrupt. Ringing it costs the chef a few seconds of disruption, so you do not ring it for every single carrot.

Where this comparison breaks: the chef can ignore the bell for a while; a CPU takes the interrupt at the very next instruction boundary unless interrupts are explicitly masked. And the notebook is a fixed-size circular list, so if the chef reads too slowly, the delivery person eventually has to stop and wait.

■ 21.5.3 PLAIN – a worked example

1. Modern machines use xHCI, the eXtensible Host Controller Interface, released by Intel in 2010. It replaced the older UHCI, OHCI and EHCI designs.
2. Before any keys are pressed, the driver has set up these structures in ordinary system memory.

Structure	What it holds
Device Context Base Array	pointer per USB device
Transfer Ring	queued work per endpoint
Event Ring	completed-work notices
Command Ring	driver-to-controller orders

- Each entry in these rings is a 16-byte block called a TRB, a Transfer Request Block.
- The driver puts a Normal TRB on the keyboard's transfer ring saying: "when data arrives on this endpoint, write up to 8 bytes to physical address 0x1A2B3000, then post an event".
- The driver then writes to a doorbell register on the controller to say "there is new work on this ring". Doorbells are the only place the driver pokes the hardware directly.
- Time passes. You press A. At the next 1 ms slot, the controller polls the keyboard and gets the 8 bytes.
- The controller performs a DMA write of those 8 bytes to 0x1A2B3000. This goes over PCI Express, through the IOMMU if one is enabled, into RAM.
- The controller then writes a Transfer Event TRB onto the event ring, saying which endpoint, how many bytes, and a completion code of Success.
- The controller then raises an MSI interrupt, which is itself just a PCI Express memory write to a special address that the interrupt controller watches.
- The CPU takes the interrupt. Total elapsed time from the last bit on the USB wire to the first instruction of the kernel handler: roughly 2 to 10 microseconds on a modern desktop.

■ 21.5.4 PLAIN - what is really happening inside

- Let us be precise about "the CPU is interrupted mid-instruction".
- On x86 and on Arm, the processor checks for pending interrupts at instruction boundaries, not in the middle of an instruction.
- So the current instruction finishes and retires. A few very long instructions, such as x86 repeated string moves, are designed to be interruptible and restartable partway.
- The core then does the following, in hardware, with no software involved.
- It saves the current instruction pointer, the flags register and the stack information onto a stack.
- It reads the interrupt vector number, a value between 0 and 255 on x86, that came with the MSI message.
- It uses that number as an index into the Interrupt Descriptor Table, a table the kernel built at boot, to find the address of the handler.
- It switches privilege level from user mode to kernel mode if it was in user code, and switches to the kernel stack.
- It jumps to the handler. From the program's point of view, absolutely nothing happened; when the handler returns, execution resumes at the exact next instruction with all registers intact.
- Whatever the CPU was running has now had its pipeline flushed and quite possibly its caches disturbed. That is the real cost of an interrupt, and it is why interrupt moderation exists.

■ 21.5.5 TECHNICAL – the engineer’s version

1. xHCI 1.0 was published by Intel in 2010; the current revision is 1.2 with errata. It unified low, full, high and SuperSpeed handling in one register interface, which the earlier split of UHCI/OHCI (USB 1.x) and EHCI (USB 2.0) did not.
2. Structures, precisely: the Device Context Base Address Array (DCBAA) is indexed by slot ID. Each device context contains a slot context and up to 31 endpoint contexts. Each endpoint context points at a transfer ring.
3. Rings are circular arrays of 16-byte TRBs. The last TRB in a segment is a Link TRB pointing back to the start. A Cycle Bit is toggled each time the producer wraps, which is how consumer and producer distinguish new entries from stale ones without a separate index.
4. The Event Ring Segment Table (ERST) describes the event ring’s memory. The controller is the producer of events; the driver is the consumer and updates the Event Ring Dequeue Pointer register when it has processed them.
5. Doorbell registers: one per device slot, plus doorbell 0 for the command ring. Writing a doorbell tells the controller to re-examine a ring. This is the only per-transfer MMIO write in the fast path, which matters because MMIO writes to a PCIe device cost hundreds of nanoseconds.
6. Interrupts: MSI was introduced in PCI 2.2 (1998) and MSI-X in PCI 3.0 (2004). Both replace a physical interrupt line with a posted memory write to an address decoded by the interrupt controller, which removes the shared-line ambiguity and the need to read a device register to find out who interrupted.
7. xHCI provides an Interrupt Moderation register per interrupter, expressed in 250 nanosecond units, which sets a minimum gap between interrupts. Raising it reduces interrupt rate and increases worst-case latency. Exact defaults are an **implementation detail** that differs between the specification’s reset value and what Linux, Windows and macOS drivers actually program.
8. IOMMU involvement: with an IOMMU enabled (Intel VT-d, AMD-Vi, Arm SMMU) the controller’s DMA addresses are I/O virtual addresses translated per device. This costs an IOTLB lookup, normally tens of nanoseconds when hot, and it is what stops a malicious USB controller from reading arbitrary memory.
9. Rough interrupt entry costs on x86-64, order of magnitude:

Step	Typical cost
DMA write of 8 bytes	under 1 us
MSI write to APIC	~0.1 to 0.5 us
Hardware IDT dispatch	~0.1 to 1 us
Handler entry with mitigations	+0.2 to 1 us

10. Tools: `cat /proc/interrupts` shows per-CPU counts per IRQ, including `xhci_hcd`. `sudo perf stat -e irq:irq_handler_entry` counts handler entries. `sudo trace-cmd record -e xhci-hcd` traces xHCI ring activity.

■ 21.5.6 WORDS – remember these

1. Host controller — the chip that runs the USB bus — the xHCI device that schedules transfers and DMA’s data, independent of the CPU.
2. DMA — hardware writing to memory by itself — direct memory access, a bus master writing to system RAM without CPU involvement.
3. TRB — one entry in a work list — a 16-byte Transfer Request Block on an xHCI transfer, command or event ring.

4. Ring — a circular list in memory — a producer/consumer queue with a Cycle Bit distinguishing fresh from stale entries.
5. Doorbell — the poke that says “new work” — a memory-mapped register write telling the controller to re-scan a ring.
6. MSI — an interrupt sent as a memory write — Message Signaled Interrupt, a posted PCIe write to an address the interrupt controller decodes.
7. IDT — the table of interrupt handlers — the Interrupt Descriptor Table, 256 gate descriptors indexed by vector number on x86.

21.6 The kernel takes over

■ 21.6.1 PLAIN - in simple words

1. The CPU is now running kernel code because of the interrupt. This code is called the interrupt service routine, or handler.
2. The handler must be fast. While it runs, other things are held up, so it does the minimum and defers the rest.
3. It reads the event note the controller left, finds out which device and which transfer finished, and marks that transfer complete.
4. It then schedules the real work to happen a moment later, outside the handler. On Linux that deferred half is a softirq or a work queue.
5. The real work starts in the USB core, the part of the kernel that knows about USB in general but nothing about keyboards.
6. The USB core looks at which driver claimed this interface and hands the 8 bytes to it. For a keyboard, that is the HID class driver.
7. The HID driver already read the device’s report descriptor at plug-in time and built a map of where every field lives inside the report.
8. It uses that map to pull out “modifier bits” and “the six key slots”, then compares with the previous report to work out what changed.
9. A key present now but not before is a press. A key present before but not now is a release.
10. Finally it translates each USB usage code into the operating system’s own key code, and hands that to the input subsystem, which is the thing applications can actually read from.

■ 21.6.2 PLAIN - a picture in your head

1. Imagine a large office building’s post room.
2. A courier drops a sack at the loading bay and rings the bell. Somebody signs for it in ten seconds flat and goes back to work. That is the interrupt handler: sign, do not open.
3. Later, a sorter opens the sack. They do not read the letters. They look at the department code on each envelope and put it in the right pigeonhole. That is the USB core.
4. The department clerk opens the envelope. They know this department’s forms, so they know that box 3 on the form is “employee number” and box 7 is “date”. That is the HID driver using the report descriptor.
5. The clerk then rewrites the information onto the building’s own standard form, because every department in the building uses different paper but the building’s computer only accepts one format. That is the translation from HID usages to kernel key codes.
6. The standard form goes into a tray that anybody with permission can read from. That tray is the input subsystem.

Where this comparison breaks: the post room is sequential and slow. The kernel does all of this in

a few microseconds, and the “tray” is a shared memory structure with a wake-up mechanism, so readers are woken the instant something lands rather than checking periodically.

■ 21.6.3 PLAIN – a worked example

1. On Linux you can watch this directly. `evtest` opens a device in `/dev/input/` and prints every event.
2. Press and release A on a US keyboard and you see this.

```
Event: time 1755105143.201933, type 4 (EV_MSC),
       code 4 (MSC_SCAN), value 70004
Event: time 1755105143.201933, type 1 (EV_KEY),
       code 30 (KEY_A), value 1
Event: time 1755105143.201933, ----- SYN_REPORT -----
Event: time 1755105143.298114, type 4 (EV_MSC),
       code 4 (MSC_SCAN), value 70004
Event: time 1755105143.298114, type 1 (EV_KEY),
       code 30 (KEY_A), value 0
Event: time 1755105143.298114, ----- SYN_REPORT -----
```

3. Line by line. `EV_MSC` with `MSC_SCAN` and value `70004` is the raw hardware code: usage page `0x0007`, usage `0x0004`. The kernel passes it through so tools can remap unusual keys.
4. `EV_KEY` with code `30` is `KEY_A`, the Linux input key code for that physical key. It is `30`, not `4` and not `65`. A third numbering.
5. value `1` means pressed. value `0` means released. value `2` means auto-repeat, generated by a kernel timer, not by the keyboard.
6. `SYN_REPORT` marks the end of one batch of simultaneous events. Everything between two `SYN_REPORT` lines happened at the same instant.
7. The gap between the two timestamps is `96.181` ms. That is how long the finger was on the key, which is a completely normal typing dwell time.
8. Notice what is still not present. Nothing anywhere says the letter A. Still no character.

■ 21.6.4 PLAIN – what is really happening inside

1. Order of events inside Linux, in detail.
2. `xhci_irq` runs as the hard interrupt handler. It reads the event ring, finds the Transfer Event TRB, and calls the completion function attached to the URB.
3. An URB, USB Request Block, is the kernel’s object for one transfer. Its completion runs the HID driver’s `usb_hid_irq_in` callback.
4. The HID driver calls `hid_input_report` with the 8 bytes. This is where the parsed report descriptor is used.
5. `hid_input_report` walks the report’s fields. For each field it extracts the bits and compares with the stored previous value.
6. For each changed usage it calls `hid_input_field`, which maps the HID usage to a Linux input event through `hidinput_hid_event`.
7. The mapping is a table. Usage page `0x07`, usage `0x04` becomes `EV_KEY`, `KEY_A`, which is `30`.
8. `input_event` is called, which puts the event into the input core.
9. The input core hands the event to every registered handler. `evdev` is one such handler and it appends to a per-open-file circular buffer, then wakes any process sleeping on that file descriptor.
10. Interrupt handlers must not sleep, so all of this after the initial read runs in `softirq` context or a tasklet. Under a normal load this whole path takes 5 to 50 microseconds.

- Under a heavily loaded system with the CPU busy, the wake-up of the reading process is at the mercy of the scheduler, and that is where a millisecond can quietly disappear.

■ 21.6.5 TECHNICAL – the engineer’s version

- Three operating systems, three input stacks, doing the same job:

OS	Layer names
Linux	usbcore, hid-core, input, evdev
macOS	IOUSBHostFamily, IOHIDFamily
Windows	USBXHCI, HIDCLASS, kbdclass

- Linux: character devices appear as `/dev/input/eventN` with a stable alias under `/dev/input/by-id/`. The protocol is a stream of `struct input_event { struct timeval time; __u16 type; __u16 code; __s32 value; }`. Key codes live in `include/uapi/linux/input-event-codes.h`. `KEY_A` is 30, `KEY_LEFTSHIFT` is 42, `KEY_ENTER` is 28.
- macOS: `IOHIDFamily` parses the report descriptor into `IOHIDElement` objects. Events become `IOHIDEvents`, then `NSEvents` once they reach the window server. Since macOS 10.15 (Catalina, 2019) reading raw HID input from another application requires Input Monitoring permission, which is why key loggers and remapping tools prompt for access.
- Windows: `HIDCLASS.SYS` parses reports. `kbdhid.sys` translates HID usages into PS/2-style scan codes for backward compatibility, then `kbdclass.sys` feeds `Win32k`, which posts a `WM_KEYDOWN` message to a thread’s message queue. Alternatively an application can call `RegisterRawInputDevice` and receive `WM_INPUT` with the raw HID report, bypassing the translation. Games do this, because the message path applies key repeat, focus filtering and accessibility features that a game does not want.
- The Windows raw input path is genuinely lower latency than the message path, mostly because it avoids waiting for the message pump and avoids the translation layers, not because the reports arrive earlier.
- Linux key code space: `KEY_MAX` is `0x2FF` (767), so there is room for many more keys than any keyboard has. Codes above 255 cannot be expressed in the older X11 protocol without an offset of 8, which is where the famous “X keycode equals evdev code plus 8” rule comes from.
- Timestamps: evdev timestamps are taken in the input core when the event is generated, not when userspace reads it. Since Linux 4.4 you can request `CLOCK_MONOTONIC` timestamps with `EVIOTC-SLOCKID`, which is what compositors do so that input times and frame times are comparable.
- Tools: `evtest`, `libinput debug-events`, `sudo showkey -s` for raw scan codes on a console, `xinput test-xi2` on X11, `wev` on Wayland, `dmesg` for enumeration messages, `hid-recorder` from the `hid-tools` project for capturing raw reports.

■ 21.6.6 WORDS – remember these

- ISR — the code that runs when hardware pokes the CPU — the interrupt service routine, running with interrupts partly disabled and forbidden to sleep.
- URB — the kernel’s object for one USB transfer — USB Request Block, submitted asynchronously with a completion callback.
- HID class driver — the code that understands report layouts — the driver that parses report descriptors and emits generic input events.
- evdev — the Linux input file interface — the event device layer exposing `/dev/input/eventN` as a stream of `input_event` structures.

5. Key code — the operating system’s own number for a key — a `KEY_*` constant on Linux, a virtual key on Windows, distinct from both scan codes and characters.
6. `SYN_REPORT` — the marker for “that batch is finished” — an `EV_SYN` event grouping simultaneous events into one atomic update.

21.7 From key code to character

■ 21.7.1 PLAIN – in simple words

1. So far, at every single stage, nothing has known what letter you typed.
2. The switch knew “closed”. The matrix knew “row 2, column 0”. The keyboard knew “usage 0x04”. The kernel knew “`KEY_A`, value 1”.
3. All of those are names for a position on a slab of plastic.
4. Now, finally, something turns that position into a character.
5. The thing that does it is a table called a keymap, and it belongs to your user account, not to your hardware.
6. The keymap says: for this key, with no modifiers, produce a. With Shift, produce A. With Alt Gr, produce something else again.
7. Change your keymap to French and the same key produces q and Q instead. You have not touched the keyboard.
8. Some keys produce nothing on their own. Press the acute accent key on a French keyboard and nothing appears. Press e afterwards and é appears. That is a dead key.
9. Some systems let you type a sequence: a special Compose key, then o, then c, giving the copyright sign. That is a compose sequence.
10. And for languages with thousands of characters, a whole extra program sits in the middle, taking your keystrokes and offering you candidate words to pick from. That is an input method editor.

■ 21.7.2 PLAIN – a picture in your head

1. Think of a piano with 88 keys and an orchestra behind a curtain.
2. Each piano key sends a number to the conductor. Key 40, key 41, and so on.
3. The conductor holds a sheet that says what each number means today. Today’s sheet says number 40 means a violin note.
4. Tomorrow the conductor swaps in a different sheet, and number 40 means a drum. The piano is unchanged. The keys are unchanged. The sound is different.
5. Now imagine a language with 50,000 sounds and only 88 keys. No sheet can map one key to one sound.
6. So instead the pianist plays a short phrase, and an assistant listens, guesses which of several possible sounds was meant, and holds up a small list to choose from.
7. That assistant is the input method editor. It sits between the keys and the result, and it is a real program with its own memory and its own predictions.

Where this comparison breaks: a piano key really does have a fixed physical string, and a keyboard key genuinely has nothing fixed at all. Also, the conductor’s sheet in real systems is not one flat list. It is a two-dimensional table with a group axis, for which language you are in, and a level axis, for which modifiers you hold.

■ 21.7.3 PLAIN – a worked example

1. One physical key. Four different keymaps. Four different results.

Keymap	Plain	With Shift
US QWERTY	a	A
French AZERTY	q	Q
German QWERTZ	a	A
Dvorak	a	A

2. The same physical key on a French AZERTY board is where Q lives, so the letter differs even though the code sent was identical.
3. Now dead keys. On a French keyboard, the circumflex key sends its own key code. The keymap marks it as a dead key.
4. Press it: nothing appears. The system remembers “a circumflex is pending”.
5. Press e: the system combines pending circumflex with e and produces ê.
6. Press the space bar instead: the system produces the bare ^ character.
7. Now compose sequences. On Linux, if you set a Compose key, this file controls it:

```
/usr/share/X11/locale/en_US.UTF-8/Compose
```

```
<Multi_key> <o> <c>          : "(c)"   copyright
<Multi_key> <a> <e>          : "ae"    ae ligature
<Multi_key> <minus> <minus> <period> : "--"  en dash
```

8. Now an input method editor. To type the Japanese word for Tokyo you might type toukyou on a plain US keyboard.
9. The IME converts those letters into kana as you go, showing the five kana to-u-kyo-u underlined as pre-edit text. Nothing is committed yet.
10. You press space. The IME offers a candidate list of kanji spellings that all read “toukyou”: the city name, a rarer compound, and others. You pick one with the arrow keys and press Enter.
11. Only then does the application receive the finished text. Everything before that was a conversation between you and the IME.

■ 21.7.4 PLAIN – what is really happening inside

1. A keymap is not a simple list. It has two axes.
2. The first axis is the **group**, which means which layout is active. You can have several loaded and switch between them with a hotkey.
3. The second axis is the **level**, which is chosen by which modifiers you hold. Level 1 is plain, level 2 is Shift, level 3 is Alt Gr, level 4 is Alt Gr with Shift.
4. So a lookup is: this key code, in this group, at this level, gives this symbol.
5. Modifiers themselves are stateful. The system tracks which are held, which are latched (pressed once, applying to the next key), and which are locked (Caps Lock).
6. Caps Lock is not Shift. Caps Lock normally affects only letters, whereas Shift affects every key. That is why Shift-2 gives @ but Caps Lock plus 2 still gives 2.
7. Dead keys are implemented as a small state machine. The keymap marks certain symbols as combining. When one is produced, the system stores it and waits for the next symbol.
8. If the next symbol has no defined combination, most systems emit both separately rather than losing your keystroke.

9. An input method editor is a separate process. It gets keystrokes first, before the application, and decides whether to consume them or pass them on.
10. While it is composing, it sends the application “pre-edit” text, which the application is expected to display inline but not to treat as committed.
11. When you confirm, the IME sends a commit event with the final string, and the pre-edit is cleared.
12. Why this cannot live in the keyboard: an IME needs a dictionary of hundreds of thousands of words, learns from your habits, needs megabytes of storage, and must know what the application is doing. A 2 KB microcontroller cannot and should not do that.

■ 21.7.5 TECHNICAL – the engineer’s version

1. Linux and X11 use XKB, the X Keyboard Extension, standardized in X11R6.1 in 1996. Its data lives in the xkeyboard-config package under /usr/share/X11/xkb/ split into key-codes, types, compat, symbols and geometry.
2. A symbols entry looks like this, and this is the actual syntax:


```
key <AC01> { [ a, A, ae, AE ] };
key <AD01> { [ q, Q, at,  Omega ] };
```
3. <AC01> is a position name meaning alphanumeric section, row C, key 01, counting from the bottom-left. The four entries are levels 1 to 4. Position names, not letters, are the identifiers. That is the whole point.
4. libxkbcommon is the modern library that implements this without needing an X server; Wayland compositors and many terminals use it directly.
5. macOS uses .keylayout XML bundles in /Library/Keyboard Layouts/, evaluated by UKey-Translate from the Carbon Unicode Utilities. Ukelele is the common editor for them.
6. Windows keyboard layouts are DLLs, historically built with the Microsoft Keyboard Layout Creator. kbdus.dll is the US layout. Applications call ToUnicodeEx to run a virtual key plus keyboard state through the active layout and get characters back.
7. Input method frameworks:

Platform	Framework	Common engines
Linux	IBus, Fcitx5	Mozc, Anthy, Pinyin
Windows	TSF (since 2001)	MS-IME, Pinyin
macOS	InputMethodKit	Kotoeri, Pinyin
Android	InputMethodService	Gboard

8. Scale of the problem: Unicode 16.0, published September 2024, defines 154,998 characters. A keyboard has around 104 keys. The ratio is roughly 1,500 characters per key. No hardware solution exists, and none can exist.
9. Korean is a special case worth knowing: Hangul syllables are algorithmically composed from jamo, so a 2-set keyboard plus a composition automaton produces all 11,172 modern syllable blocks with no dictionary at all. It is an input method, but a deterministic one.
10. Indic scripts need reordering as well as composition. In Devanagari, the vowel sign i is typed after its consonant but displayed before it, so the input method and the text shaper both have work to do on the same keystroke.
11. Tools: setxkbmap -print -verbose 10 shows the active XKB configuration; xkbcomp \$DISPLAY out.xkb dumps the compiled keymap; xkbcli interactive-wayland shows live keysym resolution; ibus engine and fcitx5-remote query the active input method.

12. Honest note on where experts disagree: some argue that firmware remapping, as offered by QMK and VIA, is better because the layout follows the keyboard between machines. Others argue it is worse because the operating system then cannot know what the layout is, breaking shortcut display, accessibility tools and input methods. Both are right about their own failure mode.

■ 21.7.6 WORDS – remember these

1. Keymap — the table turning key positions into characters — a per-user mapping from key code plus modifier state to keysyms and text.
2. Group — which language layout is active — the XKB axis selecting among several loaded layouts, switched by a hotkey.
3. Level — which modifier row of the table applies — the XKB axis selected by Shift, Alt Gr and their combinations, typically four levels.
4. Dead key — a key that waits for the next one — a combining accent that produces no character alone but modifies the following one.
5. Compose sequence — a short recipe of keys giving one character — a multi-key sequence resolved by a Compose table.
6. IME — the helper program for large scripts — an input method editor converting keystrokes to candidate text through a dictionary and a pre-edit/commit protocol.
7. Pre-edit — text you are still composing — uncommitted composition text sent to the application for display but not inserted into the document.

21.8 The window system

■ 21.8.1 PLAIN – in simple words

1. Your screen has many windows on it. Only one of them should receive what you type.
2. The program that decides this is the display server, or on modern Linux, the compositor.
3. It keeps a note of which window has focus, meaning which window is currently listening.
4. Focus changes when you click on a window, or use a keyboard shortcut to switch, or when a program asks for it and is allowed.
5. So the display server takes the key event from the operating system, looks at its focus note, and sends the event to that one program.
6. For a mouse click there is an extra step, because a click has a position. The server must work out which window is under that point, and then which part of that window.
7. Working out what is under a point is called hit testing.
8. Programs do not sit staring at their input. They sleep. The display server wakes them by delivering a message.
9. A program that is waiting for messages, handling them, and waiting again, is running an event loop. Almost every interactive program on earth has one.

■ 21.8.2 PLAIN – a picture in your head

1. Picture a large open-plan office with a single receptionist and many desks.
2. Post arrives addressed to “the person currently working on the Henderson file”. The receptionist keeps a note of who that is.
3. The note changes during the day. When it changes, the receptionist tells the old person “you are no longer on it” and the new person “you are now on it”. Those are the focus-out and focus-in events, and programs really do get them.
4. Every worker has an in-tray on their desk and is asleep at their desk.

5. The receptionist drops a letter in a tray and taps the sleeper's shoulder. They wake, read the whole tray, deal with everything, and go back to sleep.
6. That is the event loop: sleep, wake, drain the tray, act, sleep again.
7. A worker who takes twenty minutes on one letter is not asleep and not reading their tray. Their tray fills up and their window appears frozen. That is exactly what an unresponsive application is.

Where this comparison breaks: a real receptionist can be bypassed. On a modern compositor the routing is enforced, and one application cannot read another application's keystrokes. On the old X11 design it could, which is a real security difference and one of the main reasons Wayland exists.

■ 21.8.3 PLAIN – a worked example

1. Here is what an event loop actually looks like. This is pseudo-code, but it is very close to real code in every toolkit.

```
for (;;) {
    /* sleep until something happens */
    event = wait_for_next_event(connection);

    switch (event.type) {

    case KEY_PRESS:
        widget = focused_widget(window);
        text = keymap_translate(event.keycode,
                               event.modifiers);
        widget->on_key(widget, event.keycode, text);
        break;

    case BUTTON_PRESS:
        widget = hit_test(window, event.x, event.y);
        set_focus(widget);
        widget->on_click(widget, event.x, event.y);
        break;

    case RESIZE:
        relayout(window, event.width, event.height);
        break;

    case FRAME_CALLBACK:
        if (window->damaged)
            redraw(window);
        break;
    }

    if (window->damaged)
        request_frame_callback(window);
}
```

2. `wait_for_next_event` is where the program spends over 99 percent of its life. On Linux it is a `poll` or `epoll_wait` on a file descriptor.
3. Note there is no loop spinning and checking. The process is genuinely asleep and consumes no CPU until the kernel wakes it.
4. Note also that `hit_test` is called for the mouse and not for the keyboard. The keyboard goes straight to the focused widget.

5. And note the last two blocks. Drawing does not happen inside the key handler. The key handler only marks the window as needing redraw. The actual painting happens later, once, even if fifty keys arrived.

■ 21.8.4 PLAIN – what is really happening inside

1. Four systems, four architectures. It helps to see them side by side.
2. **X11.** The X server owns the screen and all input. Applications are clients connected over a socket. The server sends a `KeyPress` event containing a keycode and a modifier mask.
3. In classic X11 the client then does the keymap lookup itself using XKB data fetched from the server. Any client could also ask to receive all key events through `XGrabKeyboard` or passive grabs, which is why keyloggers were trivial on X.
4. **Wayland.** There is no separate server. The compositor is the display server, the window manager and the compositor in one process.
5. At connect time the compositor sends the client the entire keymap as a file descriptor holding XKB text. The client maps it and uses `libxkbcommon` to translate.
6. Then `wl_keyboard.key` carries the raw evdev key code and a state, and `wl_keyboard.modifiers` carries the modifier state separately. Only the focused client gets them at all.
7. **Quartz on macOS.** The `WindowServer` process owns the screen. Events become `CGEvents`, then `NSEvents`. Each application has a `CFRunLoop`, and `NSApplication` pulls events from it.
8. **Windows.** Every GUI thread has a message queue. `GetMessage` blocks on it. `TranslateMessage` looks at a `WM_KEYDOWN` and, if the key maps to a character, posts an additional `WM_CHAR` with the character in it. `DispatchMessage` calls the window's procedure.
9. That Windows split is worth remembering: `WM_KEYDOWN` is the physical key, `WM_CHAR` is the character, and they are two separate messages produced by one press.
10. In all four systems, the moment the event is handed over, a context switch happens: the compositor's process stops running and the application's process starts. That switch costs 1 to 10 microseconds when the machine is idle, and can cost milliseconds when it is busy.

■ 21.8.5 TECHNICAL – the engineer's version

1. History: X Version 11 Release 1 shipped on 15 September 1987 from MIT's Project Athena, with Robert Scheifler and Jim Gettys as the principal authors of the protocol. The protocol is still wire-compatible today, which is nearly forty years of stability and also the source of most of its problems.
2. Wayland was started by Kristian Hoegsberg in 2008 while at Red Hat. Version 1.0 of the protocol was released in October 2012. GNOME and KDE both default to Wayland sessions on current distributions, and Fedora dropped the GNOME X11 session in Fedora 41 (2024).
3. Input handling on modern Linux normally goes through `libinput`, which owns pointer acceleration, tap-to-click, palm rejection and gesture recognition, so that every compositor behaves the same way. It reads `/dev/input/eventN` directly.
4. Security model comparison:

System	Can app A see app B keys
X11	Yes, by design
Wayland	No, focus-only delivery
Windows	Only with raw input hooks
macOS	Only with granted permission

5. Wayland's frame callback (`wl_surface.frame`) is the mechanism that ties drawing to the display refresh. A client that draws only on frame callbacks never renders frames that will not be shown, which saves power but does add the wait to the latency.
6. X11 uses an offset in key codes. An X keycode is the evdev code plus 8, for historical reasons dating to the original X server key code range. So evdev `KEY_A` (30) is X keycode 38.
7. Event delivery cost: on Linux with a Wayland compositor, the path from evdev read to the client's socket becoming readable is typically 100 to 500 microseconds when idle. Under load with the compositor competing for CPU it can exceed 5 ms, which is why compositors run their input handling at elevated scheduling priority where they can.
8. Tools: `wev` prints Wayland events including the resolved keysym. `xev` does the same for X11. `libinput debug-events --verbose` shows the layer below. On Windows, `Spy++` shows the message queue. On macOS, `Quartz Debug` and the `CGEventTap` API let you observe the stream.

■ 21.8.6 WORDS – remember these

1. Display server — the program that owns the screen and input — the process holding the DRM master or equivalent and arbitrating input delivery.
2. Compositor — the thing that merges all windows into one image — on Wayland, the same process as the display server.
3. Focus — which window receives keyboard input — the keyboard focus target, tracked as state and changed by explicit focus events.
4. Hit testing — finding what is under a point — geometric resolution of a coordinate to a surface and then to a widget, respecting stacking and input regions.
5. Event loop — sleep, handle, sleep — the blocking dispatch loop at the heart of every interactive program, typically built on `poll` or `epoll`.
6. Frame callback — permission to draw for the next frame — a compositor notification synchronizing client rendering with display refresh.

21.9 The application reacts

■ 21.9.1 PLAIN – in simple words

1. The text editor is now awake with a key event in its hand.
2. It asks: which part of me has the keyboard right now? A window can have many text boxes, buttons and menus.
3. The answer is the focused widget, which in a text editor is the main text area.
4. The text area receives the event and asks the toolkit: what character is this, given the current layout and modifiers?
5. The answer comes back: the character A.
6. The editor then modifies its text. It has a big structure in memory holding every character of the document.
7. It inserts A at the cursor position, and then moves the cursor one place to the right.
8. That change may push words onto the next line, which means the text has to be laid out again from that point.
9. Laying out text means deciding, for every character, which picture to draw and exactly where on the screen to put it.
10. Finally the editor says “this rectangle of my window is now wrong, please ask me to redraw it”. It does not draw yet.

■ 21.9.2 PLAIN – a picture in your head

1. Think of a typesetter in an old print shop, arranging metal letters in a frame.
2. You hand them a new letter to insert into the middle of a line.
3. They do not just drop it in. They push everything after it along.
4. If the line now runs past the edge of the frame, the last word has to move down to the next line, and that might push a word off that line too.
5. This cascade stops when a line still has room. Usually that is one or two lines. Occasionally, adding one letter to the first line of a page reflows the entire page.
6. The typesetter also knows that certain letter pairs sit badly next to each other. A capital A followed by a capital V looks better if they are pushed slightly closer. They adjust the spacing by hand.
7. That adjustment has a name and it is still used today: kerning.

Where this comparison breaks: the typesetter has one physical piece of metal per letter. Modern text has no fixed one-to-one relation between characters and shapes. One character can produce several shapes, several characters can produce one shape, and in some scripts the shapes get reordered relative to the characters.

■ 21.9.3 PLAIN – a worked example

1. The document currently says `Hello world` and the cursor is after `Hello`, at index 5.
2. The editor stores the text. A simple editor uses a plain array, which is easy but means every insert copies everything after it.
3. Real editors use smarter structures.

Structure	Used by	Insert cost
Gap buffer	Emacs	$O(1)$ near the gap
Piece table	MS Word, VS Code	$O(\log n)$
Rope	some editors, Xi	$O(\log n)$
Flat array	toy editors	$O(n)$

4. The insert happens. The buffer now holds `HelloA world` and the cursor moves to index 6.
5. Now layout. The editor walks the paragraph and splits it into runs of text that share a font, a size, a direction and a script.
6. Each run goes to a shaping engine. The shaper takes characters and produces glyphs with positions.
7. For `HelloA` in a normal Latin font, the mapping is nearly one to one:

```
char 'H' -> glyph 43, advance 722 units
char 'e' -> glyph 72, advance 444 units
char 'l' -> glyph 79, advance 222 units
char 'l' -> glyph 79, advance 222 units
char 'o' -> glyph 82, advance 500 units
char 'A' -> glyph 36, advance 667 units
```

8. Those advances are in font design units. A typical font has 1000 or 2048 units per em. At 16 pixels per em with 1000 units, one unit is 0.016 pixels.
9. So the A occupies $667 * 0.016 = 10.67$ pixels of width. Fractional. That fraction matters and we come back to it when we rasterize.
10. The editor now knows the pixel rectangle that changed, expands it a little for safety, and calls something like `invalidate_rect(x, y, w, h)`.
11. Nothing has been drawn. One flag has been set and one rectangle recorded.

■ 21.9.4 PLAIN – what is really happening inside

1. Let us be precise about characters, glyphs and shaping, because this is where most people’s mental model is wrong.
2. A **character** is a unit of text meaning. The letter A is a character. It has a Unicode code point, U+0041.
3. A **glyph** is a picture in a font. It has a number that is meaningful only inside that one font file.
4. The mapping between them is not one to one, and here are the four ways it breaks.
5. **Ligatures.** The characters f and i can be drawn as one joined shape. Two characters, one glyph.
6. **Contextual forms.** In Arabic, most letters have four shapes depending on whether they are alone, at the start, in the middle, or at the end of a word. One character, four possible glyphs.
7. **Decomposition.** In some Indic scripts one character is drawn as several marks placed around a base. One character, several glyphs.
8. **Reordering.** In Devanagari the vowel sign for i is stored after its consonant but drawn before it. The glyph order does not match the character order.
9. Shaping is the process that resolves all of this. It reads tables inside the font that describe substitutions and positioning rules.
10. Kerning is part of the positioning step. The font contains a table saying that when glyph A is followed by glyph V, shift V left by some units.
11. After shaping the editor has a list of glyph IDs, each with an x and y offset in fractional pixels. That is the layout result.
12. The layout is cached. Editing one line does not re-shape the whole document, which is why a fast editor stays fast on a large file.

■ 21.9.5 TECHNICAL – the engineer’s version

1. Font format history: TrueType was created by Apple around 1991 and licensed to Microsoft. PostScript Type 1 came from Adobe in 1984. OpenType, announced by Microsoft and Adobe in 1996, unified them and added the GSUB and GPOS tables that make real shaping possible. OpenType became ISO/IEC 14496-22 as part of MPEG-4 in 2003.
2. Key OpenType tables in this path:

Table	Purpose
cmap	code point to glyph ID
glyf / CFF2	the outlines themselves
hmtx	advance widths
GSUB	ligatures, contextual forms
GPOS	kerning, mark attachment

3. HarfBuzz is the dominant open-source shaping engine, used by Chrome, Firefox, Android, GNOME, LibreOffice and Qt. Its name is Persian for “open type”. Apple uses CoreText, and Microsoft uses DirectWrite with its own shaping engine, usp10’s successor.
4. The hb-shape command-line tool shows shaping output directly:

```
hb-shape --font-file=/usr/share/fonts/DejaVuSans.ttf \
--unicodes=0048,0065,006C,006C,006F,0041
```

5. Text buffer structures, precisely. A gap buffer keeps a movable hole at the cursor, giving amortized O(1) inserts at the cursor and O(n) cursor jumps. A piece table keeps the original text immutable and stores an ordered list of spans into an original buffer and an append-only add buffer, which

makes undo nearly free. Visual Studio Code moved to a piece tree, a balanced tree of pieces, in 2018 and documented the change publicly.

6. Damage tracking terminology differs by toolkit. GTK calls it `gtk_widget_queue_draw`, Qt calls it `QWidget::update`, Cocoa calls it `setNeedsDisplay:`, Win32 calls it `InvalidateRect`. All four mean the same thing: record a dirty region and return immediately.
7. Coalescing is the reason this design exists. If you hold a key and the system generates 30 repeats per second, but the display refreshes 60 times per second, drawing per event would be wasteful. Marking damage and drawing once per frame is strictly better.
8. Latency cost of this section, measured on a modern desktop with a warm cache: widget dispatch under 10 microseconds, buffer insert under 1 microsecond, shaping a single line 20 to 200 microseconds depending on script and cache state, damage marking under 1 microsecond.
9. The honest version: those are the good numbers. In a browser-based editor, this same section can take 5 to 50 milliseconds, because the key event goes through JavaScript, a virtual DOM diff, a style recalculation and a layout pass, and can be delayed by garbage collection. That is a real and measurable difference, and it is why native editors still feel sharper.

■ 21.9.6 WORDS – remember these

1. Widget — one interactive part of a window — a node in the toolkit’s hierarchy with its own geometry, focus state and event handlers.
2. Character — a unit of meaning in text — a Unicode code point such as U+0041.
3. Glyph — a drawing in a font — an indexed outline within one font file, meaningless outside it.
4. Shaping — turning characters into positioned glyphs — the application of GSUB and GPOS rules by an engine such as HarfBuzz or CoreText.
5. Kerning — nudging two letters closer or apart — pairwise or class-based positioning adjustment from the font’s GPOS or legacy kern table.
6. Advance — how far to move before the next glyph — the horizontal advance width in font design units, scaled by point size.
7. Damage — the part of the window that is now wrong — the invalidated region recorded for repainting on the next frame.

21.10 Drawing

■ 21.10.1 PLAIN – in simple words

1. The editor now knows it must draw the letter A at a certain spot.
2. Inside the font, A is not a picture of dots. It is a description of its outline: a set of curves and straight lines forming the edge of the shape.
3. Turning that outline into coloured dots is called rasterizing.
4. The rasterizer works out, for every pixel, how much of that pixel falls inside the shape.
5. If a pixel is completely inside, it is fully dark. Completely outside, untouched. Half covered, half dark.
6. That partial darkness is anti-aliasing, and it is why text on a screen looks smooth instead of like a staircase.
7. There is a trick that goes further. Each pixel on most screens is actually three tiny lights side by side: red, green and blue.
8. If you treat those three as three separate samples instead of one, you get three times the horizontal detail. That is sub-pixel rendering.
9. Once the letter is drawn, the editor’s window is a finished rectangle of pixels sitting in memory.

- Then the compositor takes that rectangle, and every other window's rectangle, and stacks them into one single image the size of your screen.

■ 21.10.2 PLAIN – a picture in your head

- Imagine drawing a shape on graph paper and then colouring in the squares.
- A square fully inside the shape gets fully coloured. A square fully outside stays white.
- A square the edge passes through is the hard case. If you only allow black or white, the edge looks jagged.
- So instead you shade it grey in proportion to how much of the square the shape covers. Seventy percent covered, seventy percent grey.
- Step back and the eye blends the greys into a smooth edge. That is anti-aliasing in one sentence.
- Now imagine each square on your graph paper is actually three thin vertical strips, coloured red, green and blue, that only look white together.
- If the shape's edge covers only the left strip, you darken only red. The eye still reads it as a slightly-left-shifted edge, at a third of a square's precision.

Where this comparison breaks: darkening only the red strip really does tint the edge, and on a large low-density screen you can see coloured fringes on text. This is a genuine trade of colour accuracy for apparent sharpness, and it only works if the software knows the exact stripe order of your panel.

■ 21.10.3 PLAIN – a worked example

- Rasterize the letter A at 16 pixels per em from a font with 1000 units per em.
- Scale factor is $16/1000 = 0.016$ pixels per font unit.
- The outline is a list of points. In TrueType the curves are quadratic Beziers, each with one control point. In CFF and PostScript fonts they are cubic, with two.
- The rasterizer flattens each curve into short straight segments, fine enough that the error is under a fraction of a pixel.
- It then sweeps down the shape one scanline at a time, finding where the outline crosses that line, and filling between crossings.
- For the anti-aliased version it computes area coverage per pixel instead of a yes/no fill. The result is a small grey bitmap:

. . 3 9 9 3 . .	digits are coverage
. . 8 4 4 8 . .	0 = empty, 9 = full
. 3 9 . . 9 3 .	
. 8 9 9 9 9 8 .	the crossbar of the A
3 9 9 3	
9 4 4 9	

- Sub-pixel rendering triples the horizontal sampling first, producing three coverage values per pixel, then applies a filter across neighbouring sub-pixels to reduce colour fringing.
- Those coverages are blended with the background. Doing that blend on the raw stored values is wrong, because screen values are gamma encoded. Correct blending happens in linear light, and getting it wrong makes text look too thin on dark backgrounds and too fat on light ones.
- The glyph bitmap is cached in a glyph cache keyed by font, size, hinting mode and sub-pixel position. Typing a second A costs a cache lookup, not a rasterization.

■ 21.10.4 PLAIN – what is really happening inside

1. There are two places the drawing can happen and modern applications use both.
2. **On the CPU.** The application writes pixel values into a buffer in ordinary memory using a library such as FreeType plus Cairo, Skia's software backend, or CoreGraphics.
3. This is simple, predictable and works everywhere. It costs CPU time proportional to the number of pixels touched.
4. **On the GPU.** The application does not write pixels. It writes commands into a command buffer: bind this texture, draw these triangles, use this shader.
5. Glyphs are pre-rasterized into a big texture called a glyph atlas. Drawing a letter becomes drawing two triangles with the right texture coordinates.
6. The command buffer is submitted to a queue, and the GPU executes it independently. The CPU does not wait.
7. Either way the result is a buffer of pixels representing this one window.
8. Now the compositor. It has one buffer per visible window, plus a cursor, plus wallpaper, plus panels.
9. It composites: for each output pixel, work out which window is on top there, apply any transparency, scaling or rotation, and write the final colour.
10. This is almost always done on the GPU, because it is exactly the kind of massively parallel per-pixel work a GPU is built for.
11. There is a shortcut worth knowing. If a window is full-screen, opaque, and already in the right format, the compositor can skip compositing entirely and tell the display hardware to scan out that window's buffer directly.
12. That is called direct scanout, and it removes one full copy and often one full frame of latency. It is why full-screen games are faster than windowed ones.

■ 21.10.5 TECHNICAL – the engineer's version

1. Hinting: TrueType fonts carry a bytecode program per glyph that moves points onto the pixel grid so that stems have consistent widths at small sizes. Apple's patents on the TrueType bytecode interpreter expired in 2010, which is why FreeType enabled the interpreter by default from version 2.4 onward.
2. FreeType's own autohinter is used when a font has no useful hints. Full hinting produces crisper but distorted shapes; the current preference on high-density displays is slight or no hinting, keeping shapes faithful.
3. Sub-pixel rendering: Microsoft announced ClearType at COMDEX in November 1998. The work was led by Bill Hill with Greg Hitchcock, and Microsoft published "Displaced Filtering for Patterned Displays" in May 2000. The idea itself is older; Apple's 1976 Apple II used a related trick on colour artefacts, and IBM described sub-pixel addressing earlier still.
4. Sub-pixel rendering has three real limitations. It requires knowing the stripe order, which is RGB on most desktop LCDs and BGR on some. It breaks on a rotated display. And it is meaningless on OLED panels with non-striped sub-pixel layouts such as PenTile.
5. Apple removed sub-pixel anti-aliasing from macOS in Mojave (10.14, 2018), on the reasoning that Retina-class pixel densities make it unnecessary. This remains contested by people using non-Retina external monitors.
6. Signed distance field text, described by Chris Green of Valve at SIGGRAPH 2007 in "Improved Alpha-Tested Magnification for Vector Textures and Special Effects", stores distance-to-edge instead of coverage, allowing one atlas entry to scale smoothly. It is widely used in games and in map rendering, and it is slightly less accurate for small body text.

7. Buffer sharing on Linux is by `dma-buf` file descriptors passed over the Wayland socket, so no pixel data is copied between processes. macOS uses `IOSurface`. Windows uses `DXGI` shared surfaces.
8. Compositing cost, order of magnitude at 2560x1440 on a mid-range 2023 GPU:

Operation	Typical cost
Rasterize one glyph, CPU	5 to 50 us
Glyph cache hit	under 1 us
Composite 6 windows, GPU	0.3 to 1.5 ms
Direct scanout instead	~0 ms

9. Tools: `fc-list` and `fc-match` for font selection; `ftview` and `ftbench` from FreeType; `GALLIUM_HUD` or `mangohud` for frame timing; `weston-debug` and `drm_info` for plane and scanout state; `about:gpu` in Chrome for its compositing decisions.

■ 21.10.6 WORDS – remember these

1. Outline — the shape of a letter as curves — quadratic or cubic Bezier contours stored in `glyph` or `CFF2`.
2. Rasterize — turn a shape into pixels — scan conversion with area coverage computation.
3. Anti-aliasing — grey edges instead of jagged ones — using fractional coverage as an alpha value.
4. Sub-pixel rendering — using R, G and B as separate samples — horizontal tripling with a filter, as in `ClearType`.
5. Hinting — nudging outlines onto the pixel grid — bytecode or automatic grid fitting for stem consistency at small sizes.
6. Glyph atlas — a texture full of pre-drawn letters — a packed cache of rasterized glyphs for GPU text drawing.
7. Direct scanout — showing a window's buffer with no compositing — assigning a client buffer straight to a display plane.

21.11 Out to the screen

■ 21.11.1 PLAIN – in simple words

1. The compositor has produced one image the size of your whole screen. That image is called the `framebuffer`.
2. A piece of hardware called the display controller reads that image out of memory, continuously, forever, whether or not anything changed.
3. It reads it in a fixed order: the top-left pixel first, then along the top row, then the next row, and so on to the bottom-right.
4. Then it starts again at the top. On a 60 Hz screen it does this 60 times a second. On a 240 Hz screen, 240 times.
5. Between finishing the bottom and starting the top again there is a short gap. That gap is called the `vertical blanking interval`.
6. If you swap in a new image during the gap, the screen shows one whole image at a time. If you swap in the middle, the top of the screen shows the old image and the bottom shows the new one, with a visible line between them. That is `tearing`.
7. So the system usually waits for the gap. That waiting is called `vsync`, and it is one of the largest single delays in this whole chapter.
8. The pixel values travel out of the cable, one after another, as fast electrical signals.

9. At the other end, the monitor has its own computer. It may need to resize the image, adjust the colours, and then drive the actual panel.
10. Finally, tiny amounts of liquid crystal twist, or tiny organic diodes emit, and light leaves the screen and reaches your eye.

■ 21.11.2 PLAIN – a picture in your head

1. Picture a very fast painter repainting a fence, left to right, top plank to bottom plank, over and over.
2. The painter never stops. Even if nothing has changed, the whole fence is repainted 60 times a second.
3. You want to change the design. If you hand over the new design while the painter is halfway down the fence, the top half keeps the old design and the bottom half gets the new one. Half and half.
4. So you wait until the painter reaches the bottom and is walking back to the top. That walk back is the blanking interval. Hand over the design then and the whole fence changes together.
5. The cost of politeness is the wait. If you finish your design just after the painter starts a pass, you wait almost a whole pass.

Where this comparison breaks: the painter's walk back is short, but the pass itself takes a full frame, and a change to the top of the screen appears before a change to the bottom of the same frame. That means the physical position of your text on the screen genuinely affects when you see it, by up to a whole frame time.

■ 21.11.3 PLAIN – a worked example

1. Take 1920x1080 at 60 Hz over HDMI with standard CEA-861 timings.
2. The controller does not scan 1920x1080 pixels. It scans 2200 x 1125, including the blanking areas where no pixel is shown.
3. Total pixel clock: $2200 \times 1125 \times 60 = 148,500,000$ pixels per second, which is the well-known 148.5 MHz pixel clock for 1080p60.
4. One full frame takes $1/60$ second = 16.67 ms.
5. One scanline takes $16.67 \text{ ms} / 1125 = 14.8$ microseconds.
6. Your text is on line 400 of the screen. Once the frame starts scanning out, that line is transmitted at $400 \times 14.8 \text{ us} = 5.9 \text{ ms}$ into the frame.
7. Now compare refresh rates:

Refresh	Frame time	Average vsync wait
60 Hz	16.67 ms	8.33 ms
120 Hz	8.33 ms	4.17 ms
144 Hz	6.94 ms	3.47 ms
240 Hz	4.17 ms	2.08 ms
360 Hz	2.78 ms	1.39 ms

8. The average wait is half the frame time because your frame becomes ready at a random point in the cycle.
9. That is only the wait. On top of it comes the scanout position, the panel's own processing, and the pixel's response time.
10. And if the system is triple buffered, your finished frame may sit behind one or two other finished frames, adding one or two whole frame times on top. At 60 Hz that is another 16.67 or 33.3 ms.

■ 21.11.4 PLAIN – what is really happening inside

1. The display controller, called a CRTC on Linux, holds the memory address of the framebuffer, the timing numbers, and the format.
2. Changing which buffer is displayed is called a page flip. The driver programs the new address and the hardware latches it at the next blanking interval.
3. A page flip is atomic and cheap. Nothing is copied. Only a pointer changes.
4. The pixel data is serialized onto the cable. HDMI and DVI use TMDS, transition-minimized differential signalling, on three data lanes plus a clock. DisplayPort uses packetized micro-packets on one to four lanes with an embedded clock.
5. At the monitor, the received stream first meets a scaler if the incoming resolution does not match the panel's native resolution. Scaling costs real time, often several milliseconds, and is a common hidden source of lag.
6. Then the timing controller, the TCON, converts the stream into the signals the panel needs, driving gate lines row by row and source lines column by column.
7. On an LCD, the backlight is always on. Each sub-pixel is a tiny cell of liquid crystal between polarizers, with a colour filter. Applying voltage twists the crystal, changing how much light passes.
8. The crystal takes time to twist. That is response time, and it is why a fast-moving object smears on some panels.
9. On an OLED, each sub-pixel emits its own light. There is no backlight and no crystal to twist, so response is under 0.1 ms.
10. Light then leaves the screen. At 60 cm, the photons take about 2 nanoseconds to reach your eye. That is two millionths of a millisecond, and it is the only stage in this chapter you can safely ignore.

■ 21.11.5 TECHNICAL – the engineer's version

1. Standards and dates: HDMI 1.0 in December 2002; DisplayPort 1.0 from VESA in 2006; HDMI 2.1 in November 2017 at 48 Gbit/s; DisplayPort 2.0 in 2019 and 2.1 in 2022, with UHBR20 giving 80 Gbit/s raw.
2. Variable refresh rate: NVIDIA G-SYNC shipped in 2013 with a hardware module in the monitor. VESA added Adaptive-Sync to DisplayPort 1.2a in 2014, and AMD's FreeSync built on it from 2015. HDMI added VRR in HDMI 2.1 in 2017.
3. VRR changes the latency picture. Instead of the frame waiting for the display, the display waits for the frame, within a supported range such as 48 to 240 Hz. The average vsync wait largely disappears while the frame rate is inside that window.
4. Linux exposes all of this through DRM/KMS. The atomic modesetting API, merged in Linux 4.2 (2015), lets a compositor commit a whole new display state, including plane assignments, in one transaction.
5. Panel response times, honestly stated:

Panel type	Gray-to-gray	Notes
OLED	under 0.1 ms	near instant
TN LCD	1 to 3 ms	poor colour
Fast IPS	3 to 6 ms	common now
VA LCD	4 to 15 ms	slow dark shifts

6. Marketing "1 ms" figures are usually the fastest single transition with overdrive at its most aggressive, not an average, and overdrive causes overshoot artefacts. Dan Luu measured that a monitor

advertising 1 ms switching took roughly 10 ms to fully settle a character.

7. Display processing latency is a separate number from response time and is dominated by the scaler and any image processing. Televisions in normal picture modes commonly add 20 to 100 ms; game mode disables most of it and typically brings it under 10 ms.
8. Sample and hold: on an LCD or OLED the image is held static for the whole frame, so your eye, tracking a moving object, smears it across the retina. This persistence blur is independent of response time and is why backlight strobing exists.
9. Tools: `modetest` and `drm_info` show CRTC timings and plane usage; `xrandr -verbose` prints modelines including blanking; `sudo powermetrics` on macOS and `PresentMon` on Windows show present and display timings; the Blur Busters test patterns measure real panel behaviour.

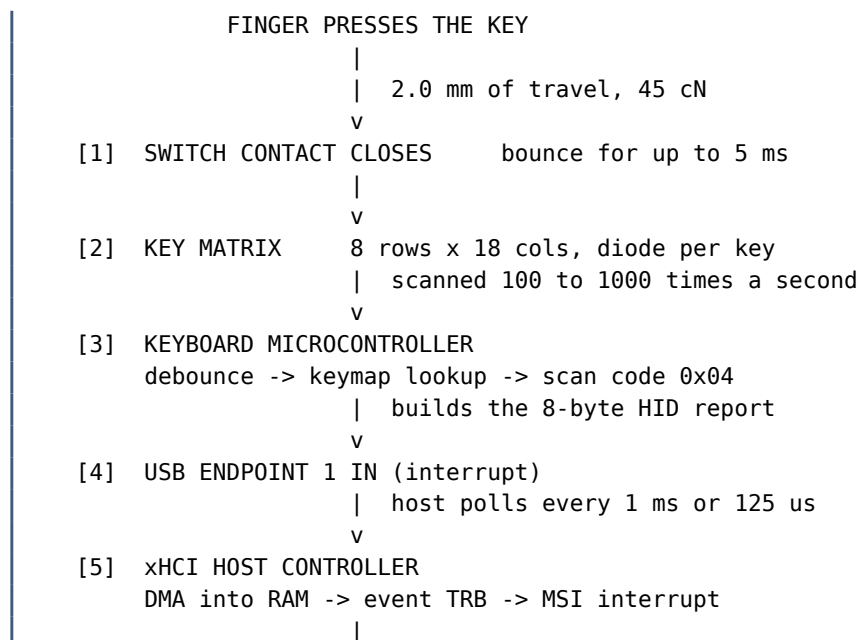
■ 21.11.6 WORDS – remember these

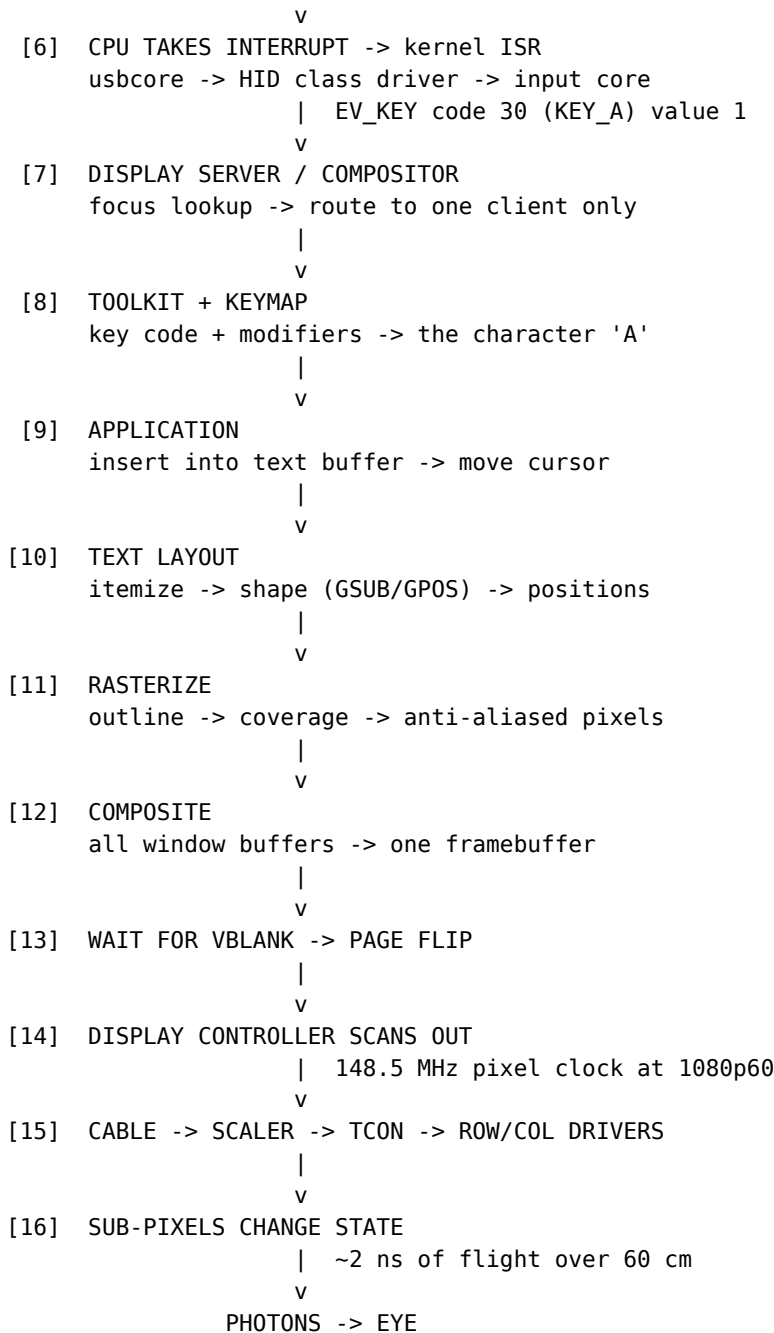
1. Framebuffer — the finished image in memory — a region of memory holding one full screen of pixels in a defined format and stride.
2. Scanout — the controller reading the image to the cable — continuous sequential transmission of the framebuffer at the pixel clock.
3. Vblank — the gap between frames — the vertical blanking interval, when no visible pixel is being transmitted.
4. Page flip — swapping which buffer is shown — an atomic change of the scanout address latched at vblank.
5. Tearing — two frames visible at once — the artefact of changing the scanout source mid-frame.
6. TCON — the panel's own controller — the timing controller driving gate and source lines from the received video stream.
7. Response time — how long a pixel takes to change — the gray-to-gray transition time of the pixel element, distinct from display processing lag.

21.12 The complete time budget

■ 21.12.1 PLAIN – in simple words

1. Here is the whole chain in one picture. Read it from top to bottom. This is every stage in this chapter, in order.





2. Now the numbers. Two columns: a carefully built fast setup, and an ordinary badly configured one.
3. The clock starts at the moment the switch closes, not when your finger starts moving.

■ 21.12.2 PLAIN – a picture in your head

1. Think of a relay race with sixteen runners, each handing a baton to the next.
2. The total time is not decided by how fast the runners are. Most of them are very fast indeed.
3. It is decided by the four or five runners who stand still and wait for a whistle before setting off.
4. In our chain the whistles are: the matrix scan, the USB poll, the frame callback, and vsync.
5. Making the fast runners faster changes almost nothing. Making the whistles more frequent changes everything.

Where this comparison breaks: some of our runners genuinely are slow, not just waiting. A browser-based editor doing a layout pass, or a television doing image processing, is real work taking real time, and no amount of scheduling fixes it.

■ 21.12.3 PLAIN – a worked example

Stage	Good ms	Bad ms
1 Contact settle	1.0	5.0
2 Debounce hold-off	0.0	5.0
3 Matrix scan wait	0.5	5.0
4 Firmware build report	0.05	0.5
5 USB poll wait	0.5	5.0
6 Transfer, DMA, IRQ	0.05	0.2
7 Kernel HID and evdev	0.05	1.0
8 Compositor routing	0.2	2.0
9 App wake and handle	0.3	15.0
10 Layout and shaping	0.1	5.0
11 Rasterize and draw	0.5	5.0
12 Compositing	0.5	8.0
13 Extra queued frames	0.0	33.3
14 Vsync wait	2.1	8.3
15 Scanout to that row	1.0	6.0
16 Panel scaler and TCON	1.0	20.0
17 Pixel response	1.0	10.0
18 Photons to eye	0.000002	0.000002
TOTAL	8.85	134.3

1. The good column assumes: eager debounce, 1000 Hz matrix scan, 1 ms USB polling, a lightly loaded machine, a native application, direct scanout, a 240 Hz monitor with no scaling, and a fast panel.
2. The bad column assumes: 5 ms symmetric debounce, 100 Hz scan, a low-speed USB keyboard polled every 10 ms, a busy machine, a browser-based editor, triple buffering, a 120 Hz refresh, and a television doing image processing.
3. The ratio between them is about fifteen to one. Nothing in that table is exotic. Both configurations are extremely common.
4. Notice where the milliseconds actually are in the bad column. Queued frames at 33.3 ms, panel processing at 20.0 ms, and the application itself at 15.0 ms. Those three are more than half the total.
5. Notice what is almost free in both columns. The USB transfer, the DMA, the interrupt and the kernel. All the parts people worry about most are the parts that cost least.

■ 21.12.4 PLAIN – what is really happening inside

1. It is important to compare that table with real measured numbers, because a sum of estimates is not evidence.
2. Dan Luu, in 2017, measured keyboards alone with a logic analyser on a cut USB cable, from the key starting to move to the USB packet leaving.

Keyboard	Latency
Apple Magic (USB)	15 ms
HHKB Lite 2	20 ms
Unicomp Model M	30 ms
Razer Ornata Chroma	35 ms
Kinesis Advantage	50 ms
Logitech K360	60 ms

3. Those are much larger than the first six rows of our table, which sum to about 2 ms in the good case. Why?
4. Because his clock starts when the key starts moving, not when the contact closes. The finger's own travel to the actuation point is several milliseconds and it is genuinely part of the experience.
5. And because real keyboards use conservative debounce and modest scan rates. A 35 ms gaming keyboard is not a measurement error. It is firmware choosing safety over speed.
6. Dan Luu also measured whole systems, key press to the character finishing on screen, with a high-speed camera.

System	Year	End to end
Apple IIe	1983	30 ms
Custom PC, 165 Hz	2014	50 ms
SGL Indy	1993	60 ms
MacBook Pro	2014	100 ms
PowerSpec G405, Windows	2017	200 ms
Symbolics 3620	1986	300 ms

7. A 1983 Apple IIe beats a 2017 desktop by a factor of nearly seven. That is the single most useful fact in this chapter.
8. For gaming setups, NVIDIA's Reflex tooling reports click-to-photon system latency. TFTCentral, testing an Asus ROG Swift PG259QNR in 2021, measured about 13 ms of PC-plus-display latency at 360 Hz with Reflex on, plus 0.8 ms of mouse latency, for 13.8 ms total, versus about 26 ms at 60 Hz with Reflex off.
9. So our "good" column of 8.85 ms is optimistic but the right order of magnitude, and our "bad" column of 134 ms sits comfortably inside the range of real measured systems.

■ 21.12.5 TECHNICAL – the engineer's version

1. Sources of the buffering penalty, which is the largest single controllable item:
2. Triple buffering with a queue depth of 3 lets the GPU render ahead by two frames. Throughput improves, latency worsens by up to two frame times.
3. Dan Luu computed this from his own data: going from 24 Hz to 165 Hz gave a 90 ms improvement, whereas pure vsync mathematics predicts only about 18 ms. The residual implies roughly 2.5 frames of buffering, independent of refresh rate, on that Windows machine running PowerShell.

4. Mitigations that actually work, with their real mechanism:

Mitigation	Mechanism
Reduce queue depth to 1	fewer frames in flight
VRR (FreeSync, G-SYNC)	display waits for frame
Direct scanout / fullscreen	skips a composite pass
Monitor game mode	disables scaler processing
Raise app scheduling prio	shortens wake-up delay

5. NVIDIA Reflex, announced 1 September 2020, works by having the driver and the game agree to delay the start of CPU work so that the CPU finishes just in time for the GPU, instead of queueing frames ahead. It is a back-pressure mechanism, not a rendering speed-up.

6. Measurement methods and what each one actually includes:

Method	Starts at	Ends at
High-speed camera	key movement	pixel change
Logic analyser on USB	key movement	USB packet
LDAT / photodiode	click signal	light change
In-engine counters	input read	present call

7. Those four methods do not measure the same thing and their results are not comparable. Most arguments about input lag on the internet are two people quoting different methods at each other.

8. Tools: PresentMon on Windows for present-to-display timing; LatencyFlex and Reflex counters in supported games; libinput measure for device-side timing; a 1000 fps phone camera, which most phones now have, for a genuine end-to-end check that requires no trust in software.

■ 21.12.6 WORDS – remember these

1. End-to-end latency — the total wait from press to light — motion-to-photon or click-to-photon latency, measured externally.
2. Queue depth — how many finished frames wait in line — the maximum frames in flight, typically 1 to 3, trading latency against throughput.
3. Back pressure — making the producer wait for the consumer — the technique behind Reflex and anti-lag features.
4. Click to photon — the gamer’s version of the same measurement — latency from a switch event to a measured luminance change at the panel.
5. Jitter — the variation in the wait, not the average — the spread of latency across repeated events, often more noticeable than the mean.

21.13 The same trace for a mouse and a touchscreen

■ 21.13.1 PLAIN – in simple words

1. A mouse click follows almost the same road. Switch, debounce, matrix or direct pin, microcontroller, USB HID, kernel, compositor, application.
2. Two things differ. First, the mouse also carries movement, and movement comes from a camera, not a switch.
3. Under an optical mouse there is a tiny camera and a light. It photographs the surface thousands of times a second.

4. It compares each photo with the one before, works out how far the pattern shifted, and reports that shift.
5. So a mouse never reports where it is. It only reports how far it moved since last time. The position on screen is the computer adding those up.
6. Second, a click has a place. The system must work out what is under the pointer before it can deliver the click.
7. A touchscreen is different again. It reports an actual position on the glass, not a movement.
8. Under the glass is a grid of transparent wires. Your finger changes the electrical capacity where it touches, and the controller finds the centre of that change.
9. The controller then works out whether you tapped, held, dragged, or made a gesture with two fingers, and reports that.

■ 21.13.2 PLAIN – a picture in your head

1. Imagine walking across a field in fog with a torch and a camera pointed at the ground.
2. Every fraction of a second you photograph the grass. You cannot see where you are in the field, but you can see that this photo is shifted two centimetres left of the last one.
3. Add up all the shifts and you have a path. That is an optical mouse, and it is also why a mouse loses tracking on glass: no grass, nothing to compare.
4. Now imagine instead a giant chessboard where every square reports when somebody stands on it. You do not need to add anything up. You are told where you are.
5. That is a touchscreen. Absolute, not relative.

Where this comparison breaks: the chessboard squares are coarse and a touchscreen is not. The controller looks at how strongly several neighbouring squares respond and computes a weighted centre, giving a position much finer than the spacing of the wires.

■ 21.13.3 PLAIN – a worked example

1. Real optical mouse sensor numbers, from the Agilent ADNS-3080 datasheet of 2005, a design still widely copied:

Property	Value
Image sensor	30 x 30 pixels
Maximum frame rate	6469 frames/s
Resolution	up to 1600 counts/inch
Maximum speed	40 inches/second
Maximum acceleration	15 g

2. At 6469 frames per second, one frame takes 155 microseconds. That is the sensor's own contribution to latency and it is tiny.
3. DPI, properly called CPI or counts per inch, is how many counts the sensor reports per inch of movement. At 1600 CPI, moving one inch produces 1600 counts.
4. Higher CPI is not "more accurate". It is finer granularity. If the sensor's noise is larger than one count, extra CPI just reports noise more precisely.
5. The USB HID boot mouse report is three bytes:

```
byte 0 : buttons, bit 0 left, bit 1 right, bit 2 middle
byte 1 : X movement, signed 8-bit, negative is left
byte 2 : Y movement, signed 8-bit, negative is up
```

6. Left click with no movement gives 01 00 00. Release gives 00 00 00. Moving right 5 and down 3 with no buttons gives 00 05 03.
7. Signed 8-bit caps movement at 127 counts per report. At 1000 Hz polling and 1600 CPI, that caps out at about 79 inches per second, which is enough. At 125 Hz polling it would cap at about 10 inches per second, which is not. That is why real mice use the report protocol with 12 or 16-bit deltas.
8. Click versus drag is decided in software, not hardware. A press, then movement beyond a threshold before release, is a drag. A press and release inside the threshold, within a time limit, is a click.
9. Mouse switches bounce too, and a worn Omron D2FC-F-7N switch is the usual cause of a mouse that double-clicks by itself. It is the debounce window failing to hide a degraded contact.

■ 21.13.4 PLAIN – what is really happening inside

1. Optical mouse tracking, step by step. An LED or a laser illuminates the surface at a shallow angle so that microscopic texture casts shadows.
2. The sensor captures a small grey image. The processor cross-correlates it with the stored previous image, testing candidate shifts, and picks the shift with the best match.
3. Sub-pixel interpolation of the correlation peak gives finer resolution than the pixel spacing. The result is a pair of integers, dx and dy.
4. Frame rate is usually variable. The sensor speeds up when moving fast and slows down when still, to save power.
5. Touchscreen, step by step. Projected capacitive screens have two layers of transparent conductor, usually indium tin oxide, in rows and columns.
6. The controller drives one row with a signal and measures the coupling into each column. Your finger steals some of that coupling.
7. This produces a small grid of numbers, a capacitance image, typically at 60 to 240 scans per second.
8. The controller finds connected regions above a threshold, computes each region's weighted centre, and assigns each a tracking identifier so that a finger keeps its identity across frames.
9. Then it decides what the touches mean: a tap, a long press, a swipe, a pinch, a rotation. Some controllers do this themselves; on phones, most of it is done by the operating system.
10. Palm rejection, edge rejection and moisture handling all live here too, and they add processing time.
11. Coordinates are reported as absolute values in a device coordinate space, then scaled into screen coordinates. This is the deepest difference from a mouse: no accumulation, no acceleration curve, no pointer to lose.

■ 21.13.5 TECHNICAL – the engineer's version

1. History: Douglas Engelbart and Bill English built the first mouse at SRI in the mid-1960s; the patent, US 3,541,541, "X-Y Position Indicator for a Display System", was granted in 1970, and the public demonstration on 9 December 1968 is remembered as the Mother of All Demos.
2. The first optical mice needed a special printed mat. Surface-independent optical tracking arrived with Agilent's sensors and the Microsoft IntelliMouse with IntelliEye in 1999.
3. Capacitive touch sensing was described by E. A. Johnson at the Royal Radar Establishment in 1965. Multi-touch projected capacitive reached the mass market with the iPhone in 2007.
4. HID reporting: mice use Generic Desktop usages X (0x30) and Y (0x31) with Relative set. Touchscreens use the Digitizer usage page (0x0D) with Absolute coordinates, contact identifiers and a contact count, defined by the Windows Precision Touchscreen and Precision Touchpad specifications.

5. On Linux, touch arrives as multitouch protocol B events: `ABS_MT_SLOT`, `ABS_MT_TRACKING_ID`, `ABS_MT_POSITION_X`, `ABS_MT_POSITION_Y`, grouped by `SYN_REPORT`. Protocol A, the older form without slots, still exists on some hardware.
6. Polling rates and scan rates in current products:

Device	Rate
Standard USB mouse	125 Hz
Gaming mouse	1000 Hz
HyperPolling class mouse	8000 Hz
Phone touch digitizer	120 to 240 Hz
Apple Pencil scan	240 Hz

7. Dan Luu's 2017 scroll-latency measurements put the 2017 iPad Pro 10.5 inch at 70 ms with a finger and 30 ms with the Pencil, against 110 to 150 ms for most Android phones of the same period. The Pencil's advantage comes from the higher scan rate and a shorter dedicated path.
8. Pointer acceleration is a software transfer function mapping counts per report to pixels, and it is a genuine source of disagreement. Games disable it and read raw counts; desktops keep it. `libinput` implements it in userspace, which is why acceleration behaviour became consistent across Linux desktops after 2015.
9. Tools: `sudo libinput debug-events` shows both relative and absolute events; `evtest` shows raw multitouch slots; `xinput list-props` shows acceleration settings; the `mousetester` utility plots real polling intervals and reveals a mouse whose 1000 Hz claim is not honoured.

■ 21.13.6 WORDS – remember these

1. Relative input — reporting movement, not position — delta reporting, as from a mouse or trackball, accumulated by the host.
2. Absolute input — reporting a position directly — coordinate reporting, as from a touchscreen or tablet digitizer.
3. CPI / DPI — counts reported per inch moved — sensor resolution, commonly 400 to 26,000, independent of accuracy.
4. Cross-correlation — matching one photo against the last — the block-matching step that converts two images into a displacement.
5. Projected capacitance — sensing a finger through glass — mutual capacitance measurement across a transparent row/column electrode grid.
6. Tracking identifier — the number that says “this is the same finger” — the slot identity that keeps contacts distinguishable between scans.

21.14 Why this matters

■ 21.14.1 PLAIN – in simple words

1. It is tempting to say that a few milliseconds cannot matter, because a millisecond is a thousandth of a second.
2. That turns out to be wrong, and we can show it with measurements rather than opinions.
3. In a shooting game, the enemy you are aiming at is where they were, plus however long your whole chain took.
4. If your chain is 100 ms and theirs is 20 ms, they see the truth 80 ms earlier than you do. At normal movement speeds that is a body width.

5. In music, the same problem shows up as feel. You hit a drum pad and hear the sound late. Your hands compensate by playing early, and the timing drifts.
6. In plain typing, a slow chain does not stop you working, but it does make the machine feel unresponsive, and there is evidence that it makes people less accurate.
7. The honest part is this: for most people, most of the delay is not in the keyboard. It is in the screen and in the software.
8. Buying a faster keyboard when your monitor adds 40 ms of image processing is spending money in the wrong place.

■ 21.14.2 PLAIN – a picture in your head

1. Think of a conversation over a satellite phone with half a second of delay.
2. Nothing is broken. Every word arrives perfectly. Yet the conversation is exhausting, because you cannot interrupt, agree or laugh at the right moment.
3. Now think of the same delay applied to your own hand. You move it and see it move half a second later. You would immediately feel that something was badly wrong.
4. Your body has an extremely tight expectation for the delay between an action and its result, because that loop is how you learned to move at all.
5. Input lag is that same violation, just smaller. Small enough to be unnamed, large enough to be felt.

Where this comparison breaks: satellite delay is constant and you adapt to it. Input lag on a computer varies from press to press, and variation is harder to adapt to than a constant offset. A steady 40 ms often feels better than a wandering 20 to 60 ms.

■ 21.14.3 PLAIN – a worked example

1. NVIDIA Research ran an aiming study and reported it in 2020 alongside Reflex.
2. They compared PCs at 12 ms and at 20 ms of system latency, an 8 ms difference.
3. The average difference in the time taken to acquire and shoot a target was 182 ms. That is roughly 22 times the latency difference.
4. On a 128-tick server, that is about 23 ticks earlier that your shots land.
5. NVIDIA also stated that most gamers play on systems with 50 to 100 ms of system latency, which puts the 12 ms machine in a different world.
6. Now music. MIDI over the original 5-pin cable runs at 31,250 bits per second. A three-byte note-on message with start and stop bits is 30 bits, so about 960 microseconds.
7. A digital audio interface with a 128-sample buffer at 48 kHz adds $128/48000 = 2.67$ ms per buffer, and there are usually at least two, in and out.
8. Real round-trip latency on a well-configured system is 5 to 12 ms. Above roughly 10 ms most players start to feel the instrument is not under their fingers.
9. A useful check: sound travels about 343 metres per second, so 10 ms is 3.4 metres. A 10 ms delay is the same as standing three and a half metres back from your amplifier, which musicians do all the time without complaint. That is why 10 ms is roughly the boundary rather than 1 ms.

■ 21.14.4 PLAIN – what is really happening inside

1. Where do the milliseconds actually go, honestly, for a normal person on a normal machine?
2. The keyboard contributes 15 to 60 ms depending on the model, mostly in debounce and scan rate, per Dan Luu's 2017 measurements.
3. The USB layer contributes 0.5 to 5 ms, and almost never more.
4. The kernel contributes well under 1 ms unless the machine is loaded.

5. The application contributes anything from 0.3 ms for a native editor to tens of milliseconds for a browser-based one.
6. The compositor and buffering contribute one to three frame times, which at 60 Hz is 16.7 to 50 ms.
7. The display contributes the vsync wait plus its own processing plus pixel response, commonly 10 to 40 ms, and far more on a television not in game mode.
8. So the honest ranking, for a typical desktop, is: display and buffering first, application second, keyboard third, and everything else a rounding error.
9. That is why the 1983 Apple IIe beat a 2017 desktop. It had no compositor, no buffering, no process boundaries, and a display that was written to directly.
10. It is also why the fix is usually free. Turning on game mode, disabling a compositor's extra buffering, using a native application, and raising the refresh rate cost nothing and beat any hardware purchase.

■ 21.14.5 TECHNICAL – the engineer's version

1. Human perception limits, from published work rather than folklore:
2. Albert Ng, Paul Dietz and colleagues, in “Designing for Low-Latency Direct-Touch Input” at UIST 2012, built a touch system with about 1 ms of latency and found that people could discriminate latencies down to approximately 2 ms in direct-touch dragging tasks.
3. That result directly contradicts the widely repeated Nielsen Norman claim that anything under 100 ms feels instantaneous. The 100 ms figure describes perceived system responsiveness for discrete actions, not the sensorimotor loop of continuous input.
4. Where experts disagree: whether polling rates above 1000 Hz produce a human-detectable improvement. Manufacturers say yes and sell 8000 Hz products. Independent blind testing has not established a consistent benefit, and the arithmetic caps the possible gain at about 0.44 ms of average wait. Treat “8K polling” claims as **marketing** until a blind study says otherwise.
5. Established fact: total system latency measurably affects task performance in aiming, per NVIDIA's published study and per earlier work in human factors going back to Sheridan and Ferrell's teleoperation studies in 1963.
6. Active research: perceptual thresholds for latency in VR and AR, where motion-to-photon budgets under 20 ms are considered necessary to avoid sickness, and where John Carmack's 2013 essay “Latency Mitigation Strategies” set much of the agenda.
7. Practical budget for a competitive setup, achievable in 2026 with ordinary parts:

Component	Realistic contribution
8 kHz analogue keyboard	1 to 3 ms
Kernel and compositor	under 1 ms
Game with Reflex-style cap	5 to 10 ms
240 Hz OLED, game mode	3 to 6 ms
Total	roughly 10 to 20 ms

8. Tools that let you argue from data: a 1000 fps phone camera for end-to-end; PresentMon for the present pipeline; Reflex or LDAT counters for click-to-photon; cyclictst to check whether your machine's scheduling latency is the real problem.

■ 21.14.6 WORDS – remember these

1. Input lag — the delay between acting and seeing the result — end-to-end system latency across the full input and display pipeline.

2. Motion to photon — the VR name for the same thing — the interval from head or hand movement to the corresponding light emission.
3. Sensorimotor loop — the body's action-and-result cycle — the closed control loop the nervous system uses, sensitive to delays of a few milliseconds.
4. Round-trip audio latency — in through the microphone and out the speaker — the sum of input buffer, processing and output buffer, typically 5 to 12 ms.
5. Tick rate — how often a game server updates the world — the server simulation frequency, commonly 64 or 128 Hz, adding its own delay on top.

21.98 Common wrong ideas

1. Wrong: the keyboard sends the letter A. Right: it sends usage code 0x04 on usage page 0x07, which names a physical position. The letter is decided by the operating system's keymap, several stages later.
2. Wrong: a key press is a single clean electrical event. Right: the contact bounces for up to 5 ms on a Cherry MX switch, and firmware must filter that into one event.
3. Wrong: a 104-key keyboard needs 104 wires to its chip. Right: an 8 by 18 matrix needs 26 pins, and the chip scans the grid repeatedly rather than watching all keys at once.
4. Wrong: "anti-ghosting" and "N-key rollover" mean the same thing. Right: anti-ghosting often means the firmware suppressing ambiguous combinations, whereas true N-key rollover needs a diode per key and a HID report format that can carry more than six keys.
5. Wrong: a USB interrupt transfer means the device interrupts the computer. Right: the device can never speak first. The host polls it on a fixed schedule, and the name is historical.
6. Wrong: the CPU is stopped in the middle of an instruction by the keyboard. Right: the keyboard's data is written into RAM by the host controller using DMA, and the CPU takes the resulting interrupt at an instruction boundary.
7. Wrong: an 8000 Hz polling rate is eight times better than 1000 Hz. Right: it removes at most about 0.44 ms of average wait, which is small next to the frame and buffering costs that dominate the chain.
8. Wrong: one character equals one glyph. Right: ligatures merge two characters into one glyph, Arabic gives one character four contextual shapes, and Devanagari reorders glyphs relative to characters.
9. Wrong: a monitor advertising 1 ms response has 1 ms of latency. Right: response time is only the pixel transition. Display processing, scaling and the vsync wait are separate and usually much larger.
10. Wrong: new computers respond faster than old ones. Right: Dan Luu measured a 1983 Apple IIe at 30 ms end to end and a 2017 Windows desktop at 200 ms. Raw speed went up and latency went up with it, because of buffering and process boundaries.

21.99 Chapter summary in 20 lines

1. A key switch is only a gate for electricity; a Cherry MX Red closes after 2.0 mm of a 4.0 mm travel at 45 cN and bounces for under 5 ms.
2. Switch families differ in how they close: membrane domes, mechanical leaves, scissor domes, optical beams, and Hall-effect magnets that report depth as a number rather than as on or off.
3. Firmware must debounce, and that choice costs between 0 and 8 ms depending on whether the algorithm fires eagerly or waits for stability.
4. Keys are wired in a row and column matrix so a few pins can read many keys, scanned 100 to 1000 times a second, costing half a scan period on average.
5. Without a diode per key, three pressed keys can create a false fourth, which is ghosting; diodes give true N-key rollover.
6. The keyboard contains a small computer that turns a matrix position into a scan code through a lookup table, and that code is a position, not a letter.
7. USB HID's boot keyboard report is exactly 8 bytes: a modifier bitmap, a reserved byte, and six key code slots; Shift plus A is 02 00 04 00 00 00 00 00.
8. The host polls the keyboard's interrupt endpoint every 1 ms at full speed or every 125 microseconds at high speed, so the report waits half a period on average.
9. Bluetooth keyboards use HID over GATT with a negotiated connection interval in 1.25 ms units, which is the main reason they feel slower.
10. The xHCI host controller DMA's the report into RAM, posts an event TRB, and raises an MSI interrupt; the CPU takes it at an instruction boundary.
11. The kernel's USB core hands the bytes to the HID class driver, which uses the parsed report descriptor to find fields and emits EV_KEY with KEY_A, code 30, through evdev.
12. The keymap, not the keyboard, decides the character, using a group axis for layout and a level axis for modifiers, with dead keys and compose sequences as small state machines on top.
13. Scripts with more characters than keys need an input method editor, a separate program with a dictionary that sends pre-edit text and then a commit.
14. The display server routes the event to the focused window only, and every interactive program is a loop that sleeps, wakes, drains events and sleeps again.
15. The application inserts the character, moves the cursor, re-shapes the affected line into positioned glyphs, and marks a rectangle as damaged without drawing anything yet.
16. Rasterizing turns an outline into per-pixel coverage; anti-aliasing uses that coverage as transparency, and sub-pixel rendering treats the red, green and blue stripes as three horizontal samples.
17. The compositor merges all window buffers into one framebuffer, unless it can hand a single window's buffer straight to the display hardware.
18. The display controller scans that framebuffer out continuously, so a new frame waits for the vertical blanking interval, costing half a frame time on average plus any queued frames.
19. A carefully built chain totals under 10 ms from contact to light; an ordinary badly configured one totals over 130 ms, and real measurements put a 1983 Apple IIe at 30 ms against a 2017 desktop at 200 ms.
20. The milliseconds live in the display, the buffering and the application, not in the wire, so measure before you spend, and remember that NVIDIA measured an 8 ms latency difference producing a 182 ms difference in aiming time.

Graphics and the GPU

22.0 What this chapter gives you

1. You will be able to explain why a graphics chip is built differently from a processor, and what “throughput” buys that “speed” does not.
2. You will be able to draw a line, fill a shape and blend two transparent images by hand, using the same arithmetic the machine uses.
3. You will be able to describe how a three-dimensional shape is stored, and trace one point of it through every coordinate space to a pixel on screen.
4. You will be able to multiply a point by a 4x4 matrix and say exactly why the fourth row and column are there.
5. You will be able to name every stage of the rasterisation pipeline, in order, and say which stages you can write code for.
6. You will be able to read a short shader program and say what runs once per corner and what runs once per pixel.
7. You will be able to explain mipmaps, depth testing, shadow maps and physical material parameters without reaching for magic words.
8. You will be able to say how ray tracing differs from rasterisation, why it was impossible in real time for forty years, and what changed in 2018.
9. You will be able to read a real graphics card specification sheet line by line and predict which number will limit your program.
10. You will be able to explain why the same chip that draws a game also trains a neural network, which is the bridge into Chapters 46 to 51.

22.1 Why a separate chip

■ 22.1.1 PLAIN – in simple words

1. A screen at 1920 by 1080 has 2,073,600 little coloured squares called pixels.
2. At 60 pictures per second, that is about 124 million pixel values to work out every second. At 4K and 120 pictures per second it is nearly a billion.
3. Every one of those pixels is worked out almost independently of its neighbours. Nobody has to wait for anybody else.
4. A main processor is built to do one job very quickly, one step after another, with clever tricks to guess what comes next.
5. That cleverness is expensive in chip area and power, and it is wasted when the same simple sum has to run two million times.
6. So we build a second chip full of very simple workers instead of a few very clever ones.

7. A graphics chip in 2026 has thousands of these simple workers. A desktop processor has eight to twenty-four clever ones.
8. Each simple worker is slower than a clever one. Together they finish the pile of work far sooner.

■ 22.1.2 PLAIN – a picture in your head

1. Imagine a post office that has to stamp one million envelopes today.
2. Option one: hire four brilliant clerks. Each can read handwriting, fix wrong addresses, phone customers and think around problems.
3. Each brilliant clerk stamps four envelopes a second. Four clerks give sixteen a second. The million envelopes take seventeen hours.
4. Option two: hire two thousand school students. Each can only do one thing: press the stamp where the arrow points.
5. Each student stamps one envelope a second. Two thousand students give two thousand a second. The million envelopes take about eight minutes.
6. If a student has to wait for a new box of envelopes, the supervisor simply turns to another student who already has a box. Nobody stands idle.
7. This is the whole idea. The brilliant clerks are a processor. The students are a graphics chip.

Where this comparison breaks: the students are not independent. They are lined up in rows of thirty-two, and every student in a row must press the stamp at the same instant on the same command. If half the row needs a different action, the other half must stand still and wait its turn. A processor core has no such restriction, and this is the single biggest reason some tasks are hopeless on a graphics chip even though they contain a lot of work.

■ 22.1.3 PLAIN – a worked example

1. Suppose we need to add 1 to every number in a list of 16,384 numbers.
2. A processor core running one instruction per cycle at 5 GHz, with no vector tricks, needs 16,384 cycles, about 3.3 microseconds.
3. With vector instructions it handles 16 numbers per instruction, so 1,024 instructions, about 0.2 microseconds.
4. A graphics chip with 16,384 lanes gives every number its own lane. One pass. At 2.4 GHz that pass is under a microsecond including overheads.
5. But the work has to get to the chip first. Sending 16,384 four-byte numbers over the connection to the card takes about 65 kilobytes of transfer.
6. Over a PCIe 5.0 x16 link at roughly 55 gigabytes per second of real throughput, 65 kilobytes takes about 1.2 microseconds each way.
7. So for this tiny job, the copying costs more than the computing. The graphics chip loses.
8. Change the list to 100 million numbers and the picture inverts completely. The chip wins by a factor of tens.
9. This is the rule: a graphics chip pays a fixed toll and then charges very little per item. It only wins when there are a great many items.

■ 22.1.4 PLAIN – what is really happening inside

1. Inside a processor core, most of the silicon is not doing arithmetic.
2. It is guessing which way a branch will go, reordering instructions, keeping large caches full, and undoing work that turned out to be wrong.
3. All of that exists to make one chain of instructions finish sooner. We call this design latency-optimised: it minimizes the waiting time of one task.

4. A graphics chip does almost none of that. It is throughput-optimised: it maximizes how many tasks finish per second, and does not care that any single one is slow.
5. When a processor core asks memory for a value and memory takes 300 cycles to answer, the core tries to find other work in the same instruction stream. Often it cannot, and it stalls.
6. When a graphics chip asks memory for a value, it simply parks that group of threads and runs a different group that already has its data.
7. Because every group's working values sit in a huge on-chip register file, the switch costs nothing. There is no saving and restoring.
8. So a processor hides waiting with caches and guessing. A graphics chip hides waiting with sheer numbers of ready threads.
9. The threads are not free-running. They are grouped, and each group shares one instruction pointer. Thirty-two threads, one instruction, thirty-two different pieces of data.
10. When threads in one group need to take different branches, the hardware runs both paths and switches off the lanes that should not be active. This is called divergence and it wastes time.

■ 22.1.5 TECHNICAL – the engineer's version

1. **SIMD** means Single Instruction, Multiple Data. One instruction operates on a fixed-width vector register. Michael Flynn defined the category in 1966.
2. On x86, AVX-512 gives 512-bit registers, that is 16 lanes of 32-bit float per instruction. The width is visible in the instruction set.
3. **SIMT** means Single Instruction, Multiple Threads. NVIDIA coined the term for the G80 architecture in 2006. It is a programming model over SIMD hardware.
4. In SIMT you write scalar code for one thread. The hardware bundles threads into fixed groups and issues one instruction for the whole group.
5. NVIDIA calls a group a **warp**, and it has been 32 threads on every NVIDIA architecture since G80. That is an implementation detail, not a standard, but it has not changed in twenty years.
6. AMD calls a group a **wavefront**. GCN used 64 threads. RDNA, from 2019, defaults to 32 (wave32) and can also run wave64. Intel Xe issues SIMD8, SIMD16 or SIMD32 depending on the compiled kernel.
7. Divergence is handled by an execution mask. Both sides of a branch execute in sequence with inactive lanes predicated off. Worst case cost is the sum of both paths.
8. Since Volta in 2017, NVIDIA hardware keeps a per-thread program counter, called independent thread scheduling. Threads in a warp can reconverge more flexibly, but they still issue as a warp.
9. Latency hiding is quantified by **occupancy**: resident warps divided by the maximum the hardware supports. Registers and shared memory per thread limit it. NVIDIA ships an occupancy calculator with the CUDA toolkit.
10. Approximate access latencies on recent NVIDIA parts, measured by microbench marks rather than published by the vendor, and therefore approximate:

Level	Typical latency	Note
Register	~1 cycle	Per-thread, huge file
Shared memory	~20–30 cycles	Software managed
L1 cache	~30–40 cycles	Per SM
L2 cache	~200–300 cycles	Chip-wide
GDDR7 VRAM	~400–600 ns	Hundreds of cycles

11. The register file on an NVIDIA H100 streaming multiprocessor is 256 KB. Across 132 of them that

is 33 MB of registers, larger than the L2 cache of most processors. That inversion is the signature of a throughput design.

12. Tools to observe this: `nvidia-smi` for utilization and clocks, Nsight Compute for per-kernel occupancy and stall reasons, `rocm-smi` and `rocprow` on AMD, Xcode's Metal debugger on Apple silicon.

■ 22.1.6 WORDS – remember these

1. Latency — how long one job takes — the delay from issue to completion of a single operation or memory request.
2. Throughput — how much work finishes per second — sustained operations or bytes per unit time, independent of any one job's delay.
3. SIMD — one order, many hands — one instruction applied across a fixed-width vector register file.
4. SIMT — scalar code, grouped execution — NVIDIA's model where per-thread programs are executed in lockstep warps on SIMD hardware.
5. Warp / wavefront — a row of workers — the hardware scheduling group: 32 threads on NVIDIA, 32 or 64 on AMD.
6. Divergence — the row splits — threads in one group taking different branches, forcing serialized execution with lane masking.
7. Occupancy — how many rows are waiting to work — resident warps per streaming multiprocessor as a fraction of the architectural maximum.

22.2 Two-dimensional graphics first

■ 22.2.1 PLAIN – in simple words

1. Before any three-dimensional trickery there is a very simple idea: a big block of memory, one entry per pixel.
2. That block is called the **framebuffer**. Write a number into a slot and that pixel changes colour. That is all drawing ever is.
3. For a 1920 by 1080 screen with four bytes per pixel, the block is 1920 times 1080 times 4, which is 8,294,400 bytes, about 7.9 mebibytes.
4. Drawing a line means deciding which slots to write. There is no such thing as a diagonal line in a grid, only a staircase of squares that looks like one.
5. Filling a shape means finding, for each row, where the shape starts and stops, and writing every slot between.
6. Copying a rectangle of pixels from one place to another is called a **blit**. It is the workhorse of all older graphics.
7. A small movable picture, drawn over the background without disturbing it, is called a **sprite**.
8. Making one image show through another is **blending**, and it uses a fourth number per pixel telling how solid that pixel is.
9. Refusing to draw outside a rectangle is **clipping**, and it is what stops one window scribbling over another.

■ 22.2.2 PLAIN – a picture in your head

1. Think of a very large sheet of squared paper, 1920 squares wide and 1080 squares tall.
2. You have coloured pencils and one rule: you may only fill in whole squares. You may never draw between them.

3. To draw a line from one corner to another you hold a ruler over the paper and fill the square nearest the ruler in each column.
4. To draw a triangle you find, in each row, the leftmost and rightmost squares inside it, and fill everything between.
5. To move a picture you rub out the squares it was on and fill in a new set. The paper does not remember anything, so you must repaint the background.
6. To make something look like coloured glass you mix the new colour with what was already in the square instead of replacing it.

Where this comparison breaks: real drawing never rubs anything out. The machine paints the whole sheet again from scratch, sixty or more times a second, into a fresh sheet while you are looking at the previous one. There are always at least two sheets, and they swap. Nothing is ever edited in place while it is visible, because you would see the edit happening.

■ 22.2.3 PLAIN – a worked example

1. Let us draw a line from pixel (0,0) to pixel (6,4) using Bresenham's algorithm, published by Jack Bresenham of IBM in 1965.
2. The idea: step one column at a time, keep a running error, and step down a row only when the error says we have drifted too far.
3. Set $dx = 6$, $dy = 4$. Set the decision value $D = 2$ times dy minus dx , so $D = 8 - 6 = 2$. Start at $y = 0$.
4. Rule at each column: plot the pixel, then if D is greater than 0, increase y by 1 and add $(2dy - 2dx)$ to D ; otherwise add $2dy$ to D .
5. Here is every step, with the true mathematical line for comparison:

x	D before	pixel plotted	true y
0	2	(0,0)	0.00
1	-2	(1,1)	0.67
2	6	(2,1)	1.33
3	2	(3,2)	2.00
4	-2	(4,3)	2.67
5	6	(5,3)	3.33
6	2	(6,4)	4.00

6. Every chosen pixel is the one nearest the true line. No division, no floating point, no multiplication inside the loop. Only integer addition.
7. Here is the whole thing in C, for the case where the line goes right and down and is wider than it is tall:

```
void line(int x0, int y0, int x1, int y1) {
    int dx = x1 - x0, dy = y1 - y0;
    int D = 2 * dy - dx;
    int y = y0;
    for (int x = x0; x <= x1; x++) {
        plot(x, y);
        if (D > 0) { y++; D += 2 * dy - 2 * dx; }
        else      { D += 2 * dy; }
    }
}
```

8. Now blending. Suppose a red marker at 25 percent opacity is drawn over a pale grey background.

9. Source colour is (200, 30, 30) with alpha 0.25. Destination is (240,240,240).
10. The standard source-over formula is: result = source times alpha, plus destination times (1 minus alpha).
11. Red: $200 \times 0.25 + 240 \times 0.75 = 50 + 180 = 230$.
12. Green: $30 \times 0.25 + 240 \times 0.75 = 7.5 + 180 = 187.5$, which rounds to 188.
13. Blue is the same as green: 188. The final pixel is (230, 188, 188), a pale pink. Exactly what a thin red wash over grey looks like.

■ 22.2.4 PLAIN - what is really happening inside

1. The framebuffer is ordinary memory. On a modern machine it lives in the graphics card's own memory, not in main memory.
2. Pixels are stored row by row. The distance in bytes from one row to the next is called the **pitch** or stride, and it is often larger than width times bytes per pixel, because rows are padded for alignment.
3. A blit is a rectangle copy that respects both pitches. Dan Ingalls wrote the first widely copied version, BitBLT, as a microcoded routine on the Xerox Alto in 1975. The Commodore Amiga shipped a famous dedicated engine for it in 1985.
4. A sprite, on very old hardware, was not drawn into the framebuffer at all. A separate circuit overlaid it as the picture was being sent to the screen.
5. Today a sprite is simply a small textured rectangle drawn by the normal pipeline, and the word survives only as a habit.
6. Alpha is a fourth channel next to red, green and blue. 0 means fully see through, 1 (or 255) means fully solid.
7. Clipping is done by comparing each pixel's coordinates with a rectangle before writing. Modern hardware does it with a scissor test that costs nothing.
8. Your window manager does not let applications write to the screen at all. Each window draws into its own private buffer.
9. A separate program, the **compositor**, then draws all those buffers onto the real screen in the right order, with the right transparency, at the right moment.
10. This is why a frozen application leaves a stale image rather than a hole, and why shadows and rounded corners on windows are cheap.

■ 22.2.5 TECHNICAL - the engineer's version

1. The first true framebuffer is usually credited to Richard Shoup's SuperPaint at Xerox PARC, working in 1973, holding 640 by 486 pixels at 8 bits each in 307 kilobytes of shift-register memory that cost a fortune.
2. Bresenham's algorithm appeared as "Algorithm for computer control of a digital plotter", IBM Systems Journal volume 4 number 1, 1965. He wrote it in 1962.
3. The compositing algebra everyone uses comes from Thomas Porter and Tom Duff, "Compositing Digital Images", SIGGRAPH 1984. It defines twelve operators; the one you meet daily is over.
4. Straight alpha stores colour and alpha separately. **Premultiplied** alpha stores colour already multiplied by alpha, which makes over cheaper and makes filtering correct:

```

straight:      out = src.rgb * src.a + dst.rgb * (1 - src.a)
premultiplied: out = src.rgb          + dst.rgb * (1 - src.a)
out alpha:     out.a = src.a + dst.a * (1 - src.a)

```

5. Filtering straight-alpha images produces dark or bright halos, because the colour of fully transparent texels leaks in. Premultiplied alpha does not have this fault. This is a convention question that bites everybody once.
6. Blending must happen in linear light, not in the sRGB values you store. Blending sRGB numbers directly is wrong and produces dark edges. This is a standard requirement, defined in IEC 61966-2-1 for sRGB, and routinely ignored.
7. The hardware blend stage is configurable but not programmable. In OpenGL you set it with `glBlendFuncSeparate` and `glBlendEquation`; in Vulkan through `VkPipelineColorBlendAttachmentState`.
8. Real framebuffer sizes, with a 32-bit colour buffer and a 32-bit depth buffer:

Resolution	Colour buffer	Colour + depth
1280x720	3.5 MiB	7.0 MiB
1920x1080	7.9 MiB	15.8 MiB
2560x1440	14.1 MiB	28.1 MiB
3840x2160	31.6 MiB	63.3 MiB

9. Compositors in the wild: Quartz Compositor on macOS since 2001, the Desktop Window Manager on Windows since Vista in 2006, Mutter and KWin on Wayland, SurfaceFlinger on Android.
10. Tools: `xwd` and `import` capture X11 windows; `weston-info` reports Wayland compositor capabilities; `RenderDoc` captures the actual draw calls of any frame on Windows, Linux and Android.

■ 22.2.6 WORDS – remember these

1. Framebuffer — the memory that is the picture — a linear array of pixel values with a defined format and pitch, scanned out to the display.
2. Pitch — bytes per row — the stride between the start of consecutive rows, including alignment padding.
3. Blit — copy a rectangle — a two-dimensional block transfer between memory regions, historically by a dedicated engine.
4. Alpha — how solid a pixel is — a per-pixel coverage or opacity channel used by the Porter-Duff compositing operators.
5. Premultiplied alpha — colour already dimmed by alpha — storage where RGB is pre-scaled by A, required for correct filtering.
6. Clipping — do not draw outside here — restricting rasterisation to a region, implemented as a scissor rectangle test.
7. Compositor — the program that assembles the desktop — the system process that blends per-window buffers into the scanout surface.

22.3 The three-dimensional problem

■ 22.3.1 PLAIN – in simple words

1. A screen is flat. A world is not. So we need a way to write down a solid shape as numbers, and a recipe for flattening it.
2. A shape is stored as a list of points in space. Each point is three numbers: how far right, how far up, how far forward.
3. A point on its own is invisible. Points are joined into flat patches, and almost every patch in every game is a triangle.

4. Three points always lie on one flat surface. Four points may not. That single fact is why the whole industry uses triangles.
5. A whole object is a **mesh**: a list of points plus a list of which three points make each triangle.
6. Each point also carries extra facts, most importantly a **normal**: an arrow saying which way the surface faces at that point.
7. Without normals, lighting is impossible, because light depends entirely on the angle between the surface and the lamp.
8. To get from a stored shape to a picture we move the numbers through several different ways of measuring, one after another.
9. First relative to the object itself, then to the world, then to the camera, then to the flat screen, then to actual pixel positions.

■ 22.3.2 PLAIN – a picture in your head

1. Think of a paper model of a house that you built on your desk.
2. When you measured the pieces you measured them from a corner of the house. The chimney is 3 centimetres up from the roof. That is the house's own ruler.
3. Then you put the house on a model railway board. Now the chimney is 40 centimetres from the left edge of the board. That is the board's ruler.
4. Then you kneel down and look at the board through a cardboard tube. Now what matters is how far the chimney is in front of your eye and how far off to the side. That is your ruler.
5. Then you trace what you see onto the flat end of the tube. Things far away trace smaller. That is the flattening.
6. Finally you photocopy the tracing onto a page of a particular size. That is turning it into pixels.
7. Five rulers, one after the other. The house never moved. Only the way we measured it changed.

Where this comparison breaks: the tracing step is not a passive act of looking. It is a specific piece of arithmetic that also throws away everything behind you, everything too close, and everything too far, and it records how far away each traced point was so that nearer things can hide farther ones later. A cardboard tube does none of that bookkeeping.

■ 22.3.3 PLAIN – a worked example

1. Here is the smallest useful mesh: a square made of two triangles.
2. Four corners, listed once each. Position, then the normal arrow, then a pair of numbers for where to sample a picture (we meet those in section 22.7).

index	position (x,y,z)	normal	uv
0	(-1, -1, 0)	(0, 0, 1)	(0, 0)
1	(1, -1, 0)	(0, 0, 1)	(1, 0)
2	(1, 1, 0)	(0, 0, 1)	(1, 1)
3	(-1, 1, 0)	(0, 0, 1)	(0, 1)

triangles: (0, 1, 2) and (0, 2, 3)

3. Notice corners 0 and 2 are used by both triangles. That is the point of the index list: store each corner once, refer to it many times.
4. A naive format would store 6 corners for 2 triangles. This stores 4. On a real mesh the saving is close to a factor of three, because in a closed surface each vertex is shared by about six triangles.
5. Storage cost here: 4 vertices times (3 + 3 + 2) floats times 4 bytes = 128 bytes, plus 6 indices times 2 bytes = 12 bytes. Total 140 bytes.

6. A detailed game character is roughly 50,000 to 150,000 triangles. At about 32 bytes per vertex and roughly as many vertices as half the triangle count, a 100,000-triangle character costs on the order of 2 to 3 megabytes before textures.
7. The order of the three indices matters. (0,1,2) walked round the triangle is anticlockwise when seen from the front. That is how the hardware later works out which side you are looking at.

■ 22.3.4 PLAIN – what is really happening inside

1. A model file on disk holds, at minimum: vertex positions, indices, and one or more sets of the extra per-vertex facts.
2. Common extras: normal, tangent (needed for surface-detail textures), one or two sets of texture coordinates, vertex colour, and for animated models a list of which bones move this vertex and by how much.
3. It also holds material references: which texture files, which shader, which numeric settings.
4. Now the five coordinate spaces, each reached by multiplying by a matrix.
5. **Model space**, also called object space. Coordinates measured from the object's own origin. This is what the artist made and what the file stores.
6. **World space**. Multiply by the model matrix. This places, rotates and scales the object into the shared scene. Every object in the level now shares one ruler.
7. **View space**, also called eye or camera space. Multiply by the view matrix. This re-measures everything relative to the camera, which now sits at the origin looking down one axis.
8. **Clip space**. Multiply by the projection matrix. This applies perspective and defines a box: anything outside it is not visible and is thrown away.
9. **Normalized device coordinates**. Divide every component by the fourth component, w. This is the step that actually makes distant things smaller. The visible region becomes a tidy cube from -1 to 1.
10. **Screen space**. Apply the viewport transform, which stretches that tidy range onto the real pixel grid, for example 0 to 1919 across.
11. Every one of these steps except the divide is a matrix multiply, and matrices can be multiplied together beforehand, so in practice three matrices are combined into one and applied once per vertex.

■ 22.3.5 TECHNICAL – the engineer's version

1. Triangles win for four concrete reasons: three points are always coplanar; three points are always convex; barycentric interpolation across a triangle is well defined and cheap; and any polygon can be triangulated.
2. Quads and higher polygons are used in modelling tools for editing convenience and subdivision. They are triangulated before reaching the hardware. This is a convention, not a limit of the mathematics.
3. Index buffers are usually 16-bit (up to 65,535 vertices) or 32-bit. The post-transform vertex cache, an implementation detail typically holding 16 to 32 entries, rewards index orders that reuse recent vertices. Tools such as Tom Forsyth's linear-speed vertex cache optimizer and meshoptimizer reorder indices for this.
4. Winding order defines facing. OpenGL defaults to counter-clockwise front faces (GL_CCW); Direct3D historically defaults to clockwise. Both are configurable. Getting it wrong makes objects vanish.
5. Vertex normals are usually the area-weighted or angle-weighted average of the adjacent face normals. Hard edges are made by splitting the vertex so the two sides carry different normals, which is why a cube needs 24 vertices, not 8.

6. Common interchange formats:

Format	Year	Nature
OBJ (Wavefront)	1980s	Text, no animation
FBX (now Autodesk)	1996	Binary, closed, rich
glTF 2.0 (Khronos)	2017	JSON + binary, open
USD (Pixar)	2016	Scene graph, open

- glTF 2.0 is deliberately close to what a graphics API wants: buffers, buffer views, accessors, and a physically based material model. It is called “the JPEG of 3D” for that reason.
- The matrix chain, written the way most maths texts and OpenGL write it, with column vectors on the right:

```
clip = P * V * M * v_model
ndc  = clip.xyz / clip.w      (the perspective divide)
win  = viewport(ndc)
```

- Handedness and depth range are the two places every engine disagrees. OpenGL traditionally uses a right-handed view space and clip depth from -1 to 1. Direct3D, Vulkan and Metal use depth 0 to 1. Vulkan also flips the Y axis of clip space relative to OpenGL. These are specification differences, not opinions, and porting code without fixing them produces upside-down or inside-out images.
- Reversed-Z, storing near objects at depth 1.0 and far at 0.0 with a floating-point depth buffer, drastically improves precision. It is now standard practice in serious engines and requires a 0-to-1 depth range.

■ 22.3.6 WORDS - remember these

- Vertex — a corner point — a record holding a position plus arbitrary attributes, consumed by the vertex shader.
- Mesh — a shape as a list of corners and triangles — indexed geometry: a vertex buffer plus an index buffer plus a topology.
- Normal — which way the surface faces — a unit vector perpendicular to the surface, used in all lighting terms.
- Winding order — the direction you walk round a triangle — the vertex order that defines front and back faces for culling.
- Model space — measured from the object — object-local coordinates as authored.
- View space — measured from the camera — coordinates after the view matrix, with the eye at the origin.
- Clip space — the pre-divide box — homogeneous coordinates where visibility is tested against $-w \leq x, y \leq w$.
- Normalized device coordinates — the tidy cube — post-divide coordinates in $[-1,1]$ for x and y, resolution independent.

22.4 The mathematics, taught gently

■ 22.4.1 PLAIN - in simple words

- Three ideas carry all of three-dimensional graphics: the vector, the dot product, and the matrix. Everything else is decoration.

2. A **vector** is just a list of numbers. In graphics it is usually three or four. It can mean a position, or a direction, or a colour.
3. The **dot product** of two vectors is: multiply matching entries, add up the results. One number comes out.
4. That one number tells you how much two directions agree. If they point the same way it is large. At right angles it is exactly zero. Opposite, negative.
5. The **cross product** of two vectors gives a third vector at right angles to both. It is how you find which way a surface faces.
6. A **matrix** is a grid of numbers that describes a transformation: a rotation, a scaling, a shear, a move.
7. Multiplying a point by a matrix gives the moved point. Multiplying two matrices gives one matrix that does both jobs.
8. For three-dimensional work we use a 4x4 grid, not 3x3, and the reason is surprisingly simple: a 3x3 matrix cannot move something.
9. The same multiply, done on much bigger grids, is what a neural network does. The chip does not know or care which one you meant.

■ 22.4.2 PLAIN – a picture in your head

1. Think of a matrix as a machine that eats an arrow and spits out a different arrow, always in a consistent way.
2. Feed it the arrow pointing one step east. It gives you back some arrow. Write that down as the first column.
3. Feed it the arrow pointing one step up. Write that answer down as the second column. Same for one step forward, third column.
4. Now you know everything the machine does, because any arrow is just so many easts plus so many ups plus so many forwards.
5. That is literally what a matrix is: the columns are where the basic direction arrows land.
6. And here is the problem. Feed such a machine the arrow of length zero, the origin. Multiply zero by anything and you get zero. The origin can never move.
7. So a 3x3 machine can turn, stretch and squash, but never shift. To allow shifting we add a fourth number to every arrow and a fourth row and column to every machine.

Where this comparison breaks: the fourth number is not a fake. It is a real coordinate in a real four-dimensional space, and the division by it at the end is the whole of perspective. Treating it as a bookkeeping trick works right up until you meet the projection matrix, at which point you need the honest story, which is in the technical block below.

■ 22.4.3 PLAIN – a worked example

1. First the dot product. Take a surface facing straight up, normal $N = (0,1,0)$. Take a direction to the lamp, $L = (0.6, 0.8, 0)$.
2. Check L is length 1: 0.6 squared is 0.36 , 0.8 squared is 0.64 , they sum to 1.00 . Good.
3. Dot product: $(0 \times 0.6) + (1 \times 0.8) + (0 \times 0) = 0.8$.
4. So this surface receives 80 percent of the light it would receive if the lamp were straight overhead. That is Lambert's cosine law in one multiplication.
5. If the lamp were below the surface, the dot product would come out negative, and we clamp it to zero. Surfaces do not receive negative light.
6. Now the full journey of one point. Take a vertex at $(1, 1, 0)$ in model space.

7. The model matrix moves the object 2 units right and 5 units away from the camera. So world position is (3, 1, -5).
8. The camera sits at the origin looking down the negative z axis, so the view matrix is the identity and view position is also (3, 1, -5).
9. The projection matrix: 90 degree vertical field of view, 16:9 aspect, near plane 0.1, far plane 100. Its non-zero entries are

```

P[0][0] = 1 / (tan(45 deg) * (16/9)) = 0.5625
P[1][1] = 1 / tan(45 deg)           = 1.0
P[2][2] = (far+near)/(near-far)      = -1.002002
P[2][3] = (2*far*near)/(near-far)    = -0.2002002
P[3][2] = -1

```

10. Apply it to the homogeneous point (3, 1, -5, 1): $x = 0.5625 \times 3 = 1.6875$. $y = 1.0 \times 1 = 1.0$. $z = (-1.002002 \times -5) + (-0.2002002 \times 1) = 5.01001 - 0.2002 = 4.80981$. $w = -1 \times -5 = 5$.
11. Clip space point is (1.6875, 1.0, 4.80981, 5). Note w is now 5, not 1. The projection matrix stashed the distance into w.
12. Divide everything by w. Normalized device coordinates are (0.3375, 0.2, 0.961962). All three are inside -1 to 1, so the point is visible.
13. Viewport transform onto a 1920 by 1080 window, with y counted downwards: screen $x = (0.3375 + 1) / 2 \times 1920 = 0.66875 \times 1920 = 1284$. screen $y = (1 - 0.2) / 2 \times 1080 = 0.4 \times 1080 = 432$.
14. The vertex lands on pixel (1284, 432), and its stored depth is 0.981. Done.
15. Now move the same object twice as far away, to world $z = -10$. Redo it: w becomes 10, x stays 1.6875, so ndc $x = 0.16875$ and screen $x = 1082$.
16. The point moved from 1284 to 1082, that is 202 pixels closer to the centre at
960. Half the distance from centre, for twice the depth. That is perspective, and it fell out of one division.

■ 22.4.4 PLAIN – what is really happening inside

1. A 4x4 matrix has a clean structure. The top-left 3x3 block does rotation, scale and shear. The rightmost column does translation. The bottom row is usually (0,0,0,1) and does nothing.
2. When the bottom row is not (0,0,0,1), you get perspective. The projection matrix puts -1 in the bottom row so that w ends up holding the view-space distance.
3. A point is written with $w = 1$. A direction is written with $w = 0$.
4. That single difference makes translation apply to points and not to directions, which is exactly right: moving a house does not change which way north is.
5. Matrix multiplication is not commutative. Rotate then move is a different result from move then rotate. Ninety percent of beginner bugs are this.
6. Because matrices combine, an engine computes one combined matrix on the processor per object per frame, and the graphics chip applies it once per vertex.
7. For a 100,000-vertex model that is one 4x4 by 4x4 multiply on the processor and 100,000 4x4 by 4x1 multiplies on the graphics chip.
8. Each of those is 16 multiplies and 12 adds. That is 28 arithmetic operations with no branching, no dependence on neighbours, and perfectly predictable memory access.
9. That shape of work is exactly what thousands of simple lanes are for. It is also, and this is the point of the chapter, exactly the shape of the work in a neural network.

■ 22.4.5 TECHNICAL – the engineer’s version

1. Homogeneous coordinates were introduced by August Ferdinand Möbius in 1827, long before computers, in his work on projective geometry.
2. Larry Roberts applied them to computer graphics in his 1963 MIT doctoral thesis on machine perception of solids, which is where the projection matrix as we use it comes from.
3. The geometric meaning of the dot product: $a \cdot b$ equals $|a||b| \cos(\theta)$. For unit vectors it is simply $\cos(\theta)$.
4. The geometric meaning of the cross product: $a \times b$ is perpendicular to both, with magnitude $|a||b| \sin(\theta)$, which equals the area of the parallelogram they span. Direction follows the right-hand rule in a right-handed system.
5. Triangle face normal from vertices A, B, C is $\text{normalize}(\text{cross}(B - A, C - A))$. Half the magnitude of that cross product is the triangle’s area, which is why the same computation drives area-weighted normal averaging.
6. Column-major versus row-major is a storage question, not a mathematical one. OpenGL and GLSL store matrices column-major; Direct3D’s older maths libraries are row-major and write vectors on the left. The same transform then appears transposed. This is a convention clash and a permanent source of bugs.
7. The general perspective projection matrix, OpenGL convention, depth mapped to $[-1, 1]$:

$$\begin{bmatrix} f/\text{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & (\text{far}+\text{near})/(\text{near}-\text{far}) & 2*\text{far}*\text{near}/(\text{near}-\text{far}) \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

where $f = 1 / \tan(\text{fovy} / 2)$

8. Vulkan and Direct3D map depth to $[0, 1]$, which changes the third row. Reversed -Z swaps near and far in that row and pairs with a GREATER depth test.
9. Depth precision: with a 24-bit fixed-point depth buffer, near = 0.1 and far = 1000, 90 percent of the stored range is used up within the first metre. Raising the near plane helps far more than raising the buffer’s bit depth. Reversed-Z with a 32-bit float buffer largely solves it.
10. The connection to machine learning, stated plainly. A vertex transform is a 4×4 matrix times a 4×1 vector. A fully connected neural network layer is an $N \times M$ weight matrix times an $M \times 1$ activation vector, usually batched into an $N \times M$ times $M \times B$ matrix multiply.
11. Both are the same primitive: multiply-accumulate over contiguous data with no control flow. The hardware unit that does one does the other. Tensor cores, introduced on NVIDIA Volta in 2017, are simply matrix-multiply units with a fixed small tile size, and they are used by both graphics and machine learning code. Chapter 46 picks this up in detail.
12. Libraries: GLM for C++ (header-only, mirrors GLSL), numpy for experimenting, cglm for C. To check a matrix by hand, print it and verify that the fourth column is your translation.

■ 22.4.6 WORDS – remember these

1. Vector — a list of numbers with a direction meaning — an element of a real vector space, here R^3 or R^4 .
2. Dot product — how much two directions agree — the scalar product, which equals $|a||b|\cos(\theta)$ and is zero for perpendicular vectors.
3. Cross product — a direction at right angles to two others — the vector product, magnitude $|a||b|\sin(\theta)$, used for face normals.
4. Matrix — a grid that transforms — a linear map expressed in a chosen basis; its columns are the images of the basis vectors.

5. Homogeneous coordinates — the extra fourth number — projective coordinates where (x,y,z,w) and (kx,ky,kz,kw) name the same point, enabling translation and perspective as linear operations.
6. Perspective divide — dividing by w — the non-linear step that converts clip space to normalized device coordinates and makes distance shrink things.
7. Model-view-projection matrix — the three rulers combined — the product $P * V * M$, uploaded as a uniform and applied per vertex.

22.5 The rasterisation pipeline, stage by stage

■ 22.5.1 PLAIN – in simple words

1. Rasterisation is the answer to one question, asked once per triangle: which pixels does this triangle cover?
2. The whole machine is a production line. Corners go in one end, coloured pixels come out the other.
3. First the machine reads the corners out of memory.
4. Then it runs your program on each corner, which is where the matrices from the last section get applied.
5. Then, optionally, it can create extra corners to add detail, or make whole new shapes.
6. Then it throws away anything outside the visible box, and cuts triangles that are half in and half out.
7. Then it does the perspective divide and maps everything onto real pixel coordinates.
8. Then it works out, for each triangle, exactly which pixel centres fall inside it.
9. For each of those it can check the depth early and skip the expensive part.
10. Then it runs your second program once per surviving pixel, which decides the colour.
11. Then it blends that colour with what is already there and writes it into memory.
12. Some of those stages you write yourself. The rest are fixed circuits you can only configure.

■ 22.5.2 PLAIN – a picture in your head

1. Think of a car factory line.
2. At the start, a robot picks parts out of bins. That is vertex fetch.
3. The first station is staffed by a worker who can do anything you train them to do. That is the vertex shader.
4. The next stations can add extra parts or duplicate the chassis. Those are tessellation and the geometry shader, and most cars skip them.
5. Then a quality gate throws out anything that will not fit in the showroom. That is clipping.
6. Then a stamping press converts the shape into a fixed grid pattern. That is rasterisation, and it is a fixed machine, not a person.
7. Then another trainable worker paints each square. That is the fragment shader.
8. Then the final fixed machine decides whether the new paint covers the old and files the result. That is depth test and blending.

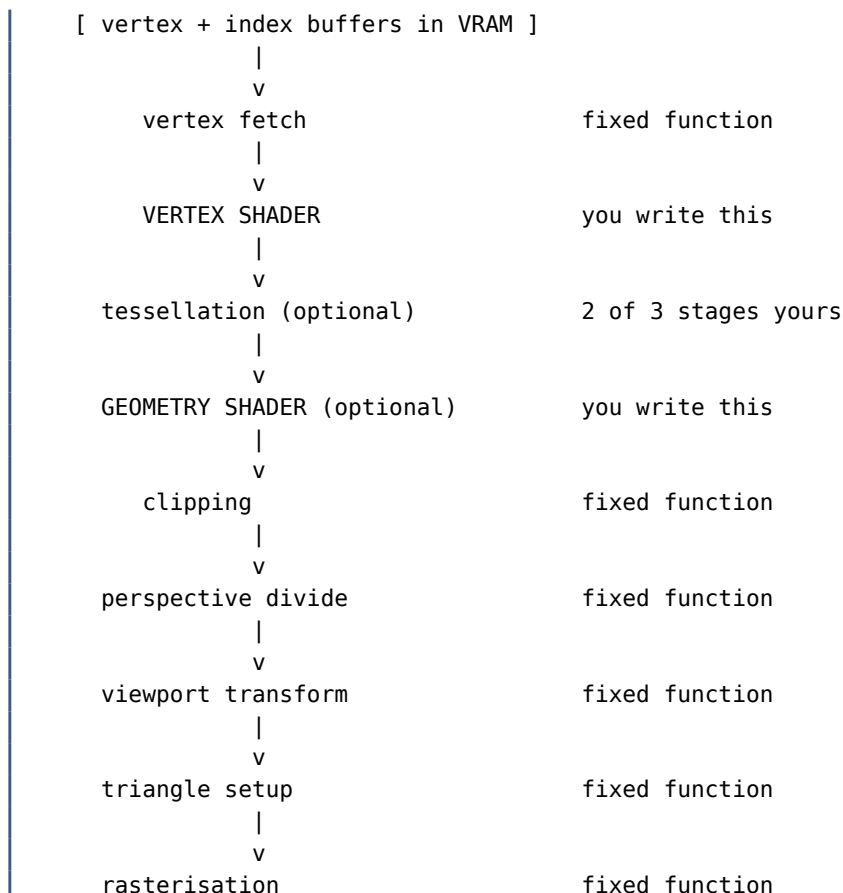
Where this comparison breaks: a factory line handles one car at a time per station. This line has thousands of cars at every station simultaneously, and the stations are not physically separate. On real hardware the same shader cores run the vertex program and the fragment program, switching between them as work arrives. The line is a logical order, not a physical layout.

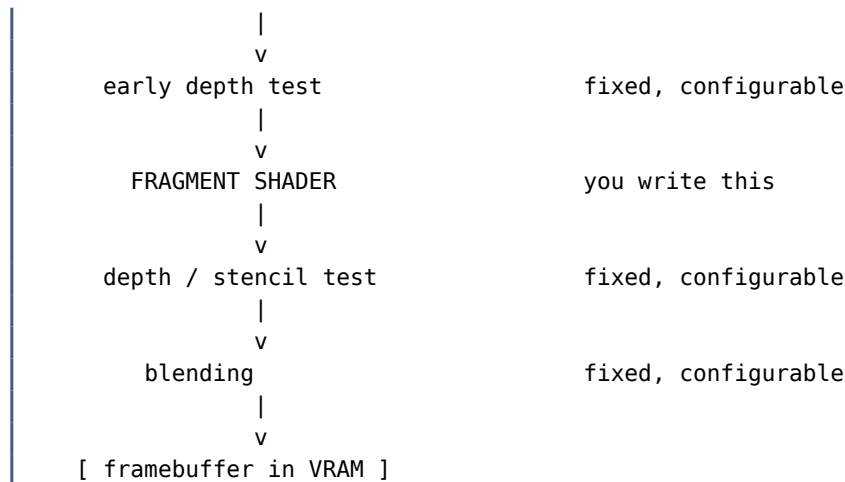
■ 22.5.3 PLAIN – a worked example

1. Draw one triangle whose screen positions after the divide and viewport transform are $A = (100, 100)$, $B = (300, 100)$, $C = (100, 300)$.
2. The bounding box is x from 100 to 300 and y from 100 to 300. That is 201 by 201, so 40,401 candidate pixels.
3. The triangle is half of that box, so about 20,200 pixels are actually inside.
4. Each candidate is tested with three edge functions. For edge AB the function is $E(x,y) = (B.y - A.y)(x - A.x) - (B.x - A.x)(y - A.y)$.
5. Substituting A and B: $E(x,y) = 0 \times (x - 100) - 200 \times (y - 100)$, which is $-200(y - 100)$. It is negative for y above 100 and zero on the line.
6. A pixel is inside when all three edge functions have the same sign. Test the pixel centre at $(150.5, 150.5)$ and it passes all three.
7. Once inside, the hardware computes barycentric weights. For the pixel $(150.5, 150.5)$: $u = 0.2525$ toward B, $v = 0.2525$ toward C, and 0.495 toward A.
8. Those weights interpolate every attribute. If A, B and C had depths 0.5, 0.7 and 0.9, this pixel's depth is $0.495 \times 0.5 + 0.2525 \times 0.7 + 0.2525 \times 0.9 = 0.6515$.
9. That depth is compared against what is already in the depth buffer. If the stored value is nearer, this pixel is discarded before the fragment shader ever runs.
10. Fragments are always processed in 2×2 groups, called quads, even at the triangle edge where one or two of the four are outside. The wasted ones are called helper lanes. On thin triangles this can waste half the work, which is why very high triangle counts stop paying off.

■ 22.5.4 PLAIN – what is really happening inside

1. Here is the full order, with which parts you can program:





2. **Vertex fetch** reads the index buffer, looks up each index in the vertex buffer, and assembles a struct for each vertex.
3. **Vertex shader** runs once per vertex. Its only required output is the clip space position. It may output anything else you want passed along.
4. **Tessellation** is three stages: a control shader you write, a fixed tessellator that subdivides a patch by the factors you gave, and an evaluation shader you write that positions each new vertex.
5. **Geometry shader** can emit zero, one or many primitives per input primitive. It is flexible and, on most hardware, slow. Modern engines avoid it.
6. **Clipping** removes primitives fully outside the view volume and cuts those that straddle it, producing new vertices with interpolated attributes.
7. **Perspective divide** divides x , y and z by w .
8. **Viewport transform** maps the -1 to 1 range onto the pixel rectangle and maps depth into the depth buffer's range.
9. **Triangle setup** computes the edge functions and the per-attribute interpolation gradients once per triangle, so the per-pixel work is cheap.
10. **Rasterisation** walks tiles of the bounding box and tests pixel centres against the edge functions, emitting fragments.
11. **Early depth test** compares depth before the fragment shader. It is only safe when the shader does not change depth and does not discard pixels.
12. **Fragment shader** runs once per fragment and outputs colour, and optionally depth.
13. **Blending** combines the shader's output with the existing pixel, then the result is written to the framebuffer.

■ 22.5.5 TECHNICAL - the engineer's version

1. The programmable stages, in Direct3D 11 and OpenGL 4 terms, are: vertex, hull (tessellation control), domain (tessellation evaluation), geometry, and pixel (fragment). Compute exists outside the graphics pipeline.
2. Everything else is fixed function. Fixed does not mean unconfigurable. The depth test comparison, blend factors, cull mode, scissor rectangle, viewport and stencil operations are all set through pipeline state.
3. Modern GPUs are **tile-based** or use binning even on desktop. AMD RDNA's Draw Stream Binning Rasterizer and NVIDIA's tiled caching, first observed on Maxwell in 2014, sort triangles into screen tiles to improve cache locality. These are implementation details and are not exposed in the APIs.
4. Rasterisation rules are specified precisely so that adjacent triangles neither double-draw nor leave

gaps. Direct3D and Vulkan both define a top-left fill rule and fixed-point vertex snapping, typically 8 subpixel bits.

5. Early depth test is called Early-Z or Hi-Z. Writing to `gl_FragDepth`, calling `discard`, or using certain blend modes disables it. A single stray `discard` in a shader can cost a large fraction of frame time.
6. Attribute interpolation must be perspective correct: the hardware interpolates `attribute/w` and `1/w` linearly in screen space, then divides. Doing it without the divide produces the warped textures of 1990s consoles. The original PlayStation of 1994 famously lacked perspective correction.
7. The mesh shader pipeline, introduced with NVIDIA Turing in 2018 and standardized in Direct3D 12 Ultimate in 2020 and Vulkan's `VK_EXT_mesh_shader` in 2022, replaces vertex, tessellation and geometry with two programmable stages: task and mesh. It is a genuine architectural change, not a rename.
8. Fragment quads: derivatives such as `dFdx` and `dFdy`, used for texture mipmap selection, only exist because fragments are shaded in `2x2` groups. This is why the quad is a hardware invariant and not an optimization.
9. Observation tools: RenderDoc and NVIDIA Nsight Graphics let you step through a captured frame stage by stage, inspect the post-vertex-shader output, and see per-draw pixel counts. `GL_ARB_pipeline_statistics_query` and Direct3D's pipeline statistics queries report primitives in, primitives clipped, and fragment shader invocations.

■ 22.5.6 WORDS – remember these

1. Rasterisation — deciding which pixels a shape covers — converting primitives into fragments by testing sample positions against edge functions.
2. Fragment — a candidate pixel — a potential contribution to a pixel, carrying interpolated attributes, before depth and blend decide its fate.
3. Barycentric coordinates — how far toward each corner — the three weights that express a point inside a triangle as a combination of its vertices.
4. Early-Z — check depth before painting — a depth test executed before the fragment shader, disabled by depth writes or `discard` in the shader.
5. Fixed function — a circuit you configure, not program — a hardware stage with settable state but no user code.
6. Quad — the `2x2` shading group — the minimum granularity of fragment shading, required for screen-space derivatives.
7. Mesh shader — the new front end — a compute-like stage that emits meshlets directly, replacing vertex, tessellation and geometry stages.

22.6 Shaders

■ 22.6.1 PLAIN – in simple words

1. A shader is a small program that runs on the graphics chip instead of the processor.
2. The name is misleading. A shader does not only handle shading. It is just the name for any program that runs at one of the programmable stages.
3. A **vertex shader** runs once for each corner. Its job is to say where that corner ends up.
4. A **fragment shader** runs once for each candidate pixel. Its job is to say what colour that pixel should be.
5. Both are written in a small C-like language. The most common are GLSL for OpenGL and Vulkan, HLSL for Direct3D, and MSL for Apple's Metal.

6. Values that are the same for every invocation, such as the camera matrix or the time, are called **uniforms**. You set them from the processor.
7. Values that the vertex shader computes and that get smoothly blended across the triangle for the fragment shader are the varying outputs.
8. Pictures the shader can look things up in are **textures**, and the object describing how to look them up is a **sampler**.
9. Thousands of copies of the same shader run at the same instant on different data. No copy can see another copy's variables.
10. A **compute shader** is the same machinery without any drawing at all. You just say how many invocations you want and they run. That is what opened the chip to work that has nothing to do with pictures.

■ 22.6.2 PLAIN – a picture in your head

1. Imagine a huge examination hall with twenty thousand desks.
2. Every candidate has the exact same question paper. That is the shader program.
3. On the front board are some facts everyone needs: today's date, the exchange rate. Those are uniforms. Nobody may change them.
4. In each candidate's envelope is their own individual data: a corner position, or a pixel coordinate. That is the per-invocation input.
5. There is a reference library at the side of the hall that anyone may consult. That is texture memory.
6. Candidates may not talk to each other. Each writes one answer and leaves.
7. The invigilator does not wait for candidate 1 before starting candidate 2. Everyone works at once.

Where this comparison breaks: candidates in the same row of thirty-two must turn the page at the same moment. If one candidate needs to read a different section of the paper, the whole row goes there and the others sit idle. And in a compute shader, small groups of candidates do get a shared scratch table they can all write on, which is the one exception to the no-talking rule and the reason compute shaders can do things fragment shaders cannot.

■ 22.6.3 PLAIN – a worked example

1. Here is a complete, working vertex shader in GLSL. It transforms a position and passes a texture coordinate and a normal onward.

```
#version 330 core
layout(location = 0) in vec3 aPos;
layout(location = 1) in vec3 aNormal;
layout(location = 2) in vec2 aUV;

uniform mat4 uMVP;
uniform mat3 uNormalMatrix;

out vec3 vNormal;
out vec2 vUV;

void main() {
    vNormal    = normalize(uNormalMatrix * aNormal);
    vUV        = aUV;
    gl_Position = uMVP * vec4(aPos, 1.0);
}
```

2. Line by line: the three in variables are read from the vertex buffer. The two uniform values were set once by the processor.

3. `gl_Position` is the one required output: the clip space position from section 22.4.
4. `vNormal` and `vUV` are outputs that the hardware will interpolate across the triangle.
5. Here is the matching fragment shader. It looks up a texture and applies one diffuse light.

```
#version 330 core
in vec3 vNormal;
in vec2 vUV;

uniform sampler2D uAlbedo;
uniform vec3 uLightDir;

out vec4 fragColour;

void main() {
    vec3 base = texture(uAlbedo, vUV).rgb;
    float ndotl = max(dot(normalize(vNormal), uLightDir), 0.0);
    fragColour = vec4(base * (0.1 + 0.9 * ndotl), 1.0);
}
```

6. `vNormal` and `vUV` arrive already interpolated. They are not the same values the vertex shader wrote; they are a weighted blend of three vertices' values.
7. `sampler2D` is the combined texture and sampler. `texture()` does the lookup, including filtering and mipmap selection, in one hardware instruction.
8. `dot(N, L)` is the Lambert term from section 22.4. The 0.1 is a crude ambient floor so that unlit sides are not pure black.
9. Now count the work. At 1920 by 1080 with every pixel covered once, this fragment shader runs 2,073,600 times per frame. At 60 frames per second that is 124 million invocations per second, of a program with one texture fetch and about ten arithmetic operations.
10. A compute shader, for comparison, invents its own grid. This one inverts an image:

```
#version 430
layout(local_size_x = 16, local_size_y = 16) in;
layout(rgba8, binding = 0) uniform image2D uImage;

void main() {
    ivec2 p = ivec2(gl_GlobalInvocationID.xy);
    vec4 c = imageLoad(uImage, p);
    imageStore(uImage, p, vec4(1.0 - c.rgb, c.a));
}
```

11. There is no triangle, no vertex, no framebuffer. You dispatch a grid of workgroups and each of the 256 invocations per group handles one pixel.

■ 22.6.4 PLAIN – what is really happening inside

1. Your shader text is not what the chip runs. It goes through two compilations.
2. First, the source is compiled to a portable intermediate form. For Vulkan that is SPIR-V, a binary format standardized by Khronos in 2015. For Direct3D it is DXIL, based on LLVM bytecode.
3. Second, the graphics driver compiles that intermediate form into the actual machine code of the specific chip installed in your machine.
4. That second step is why the same game runs on chips whose instruction sets are completely different and undocumented.
5. Uniforms are not registers. They live in a small buffer in memory that every invocation reads. Because every invocation reads the same address, the read is broadcast, and it is close to free.

6. Varying outputs are written into on-chip storage by the vertex shader, then the triangle setup stage turns them into interpolation gradients, and the fragment shader reads interpolated values.
7. A texture fetch is not a normal memory read. Dedicated texture units handle address wrapping, format decoding, filtering between neighbouring texels, and mipmap level choice, all in hardware.
8. Because a texture fetch may take hundreds of cycles, the scheduler issues it and immediately switches to another warp. This is exactly the latency hiding from section 22.1.
9. The number of registers your shader needs directly limits how many warps can be resident. A shader with too many live variables reduces occupancy and can run slower despite doing less arithmetic.

■ 22.6.5 TECHNICAL – the engineer’s version

1. History, briefly. Fixed-function transform and lighting arrived on the NVIDIA GeForce 256 in August 1999. Programmable vertex and pixel shaders were specified in Direct3D 8 in November 2000 and first shipped on the GeForce 3 in February 2001, written in assembly with instruction-count limits in the low tens.
2. High level languages followed: Cg and HLSL in 2002, GLSL in OpenGL 2.0 in 2004. Unified shader hardware, where one pool of cores runs all stages, arrived with the ATI Xenos in the Xbox 360 in 2005 and NVIDIA G80 in November 2005.
3. Compute shaders were standardized in Direct3D 11 in 2009 and OpenGL 4.3 in 2012. Metal and Vulkan had them from the start.
4. Shader stage inputs and outputs are declared with explicit locations, and the interface between stages must match or linking fails. In Vulkan this is validated at pipeline creation, not at draw time.
5. Uniform buffer objects have a guaranteed minimum size of 16 KiB in OpenGL and Vulkan; shader storage buffers are much larger and writable. Push constants in Vulkan give a small, fast path, with a guaranteed minimum of 128 bytes.
6. Interpolation qualifiers change behaviour: `smooth` is perspective correct and is the default, `noperspective` is linear in screen space, `flat` takes the value from the provoking vertex with no interpolation.
7. Workgroup limits are guaranteed minimums, not typical values. OpenGL 4.3 guarantees at least 1024 invocations per workgroup and 32 KiB of shared memory; Vulkan guarantees only 128 invocations and 16 KiB. Real desktop hardware offers 1024 and 32 KiB or more, but relying on that is not portable.
8. Subgroup or wave intrinsics let invocations in one warp exchange values directly through registers. `subgroupAdd`, `subgroupBallot` and friends were added in Vulkan 1.1 in 2018. They are the fastest reduction primitive available and are heavily used in machine learning kernels.
9. Tools: `glslangValidator` and `glslc` compile GLSL to SPIR-V; `spirv-cross` converts between shader languages; `dxc` compiles HLSL to DXIL and SPIR-V; NVIDIA’s `nvdiasm` and AMD’s Radeon GPU Analyzer show the real machine code and register counts.

■ 22.6.6 WORDS – remember these

1. Shader — a program that runs on the graphics chip — user code executed at a programmable pipeline stage in SIMT fashion.
2. Uniform — a value that is the same everywhere — read-only per-draw constant data, uploaded from the host into a constant or uniform buffer.
3. Varying — a value blended across the triangle — a vertex shader output interpolated per fragment, usually perspective correct.
4. Sampler — the rules for reading a picture — the object holding filter mode, wrap mode, mip bias and comparison state used by a texture fetch.
5. SPIR-V — the portable half-compiled shader — the Khronos binary intermediate representation consumed by Vulkan and OpenCL, standardized in 2015.
6. Compute shader — a shader with no drawing — a general kernel dispatched over an explicit grid of workgroups with shared local memory.
7. Workgroup — a small team with a shared table — a block of invocations that share on-chip memory and can synchronize with a barrier.

22.7 Textures and surface detail

■ 22.7.1 PLAIN – in simple words

1. A triangle mesh gives shape. It does not give pattern, colour or fine detail.
2. Those come from **textures**: ordinary images wrapped onto the surface.
3. To wrap an image you need to say, for each corner of each triangle, which point of the image it corresponds to.
4. That pair of numbers is called a **UV coordinate**, running from 0 to 1 across the image in each direction.
5. Between corners, the machine blends UV values, so every pixel of the triangle knows where to look in the image.
6. The image pixel you land on is called a **texel**, to keep it separate from a screen pixel. They are almost never the same size.
7. If a texel is bigger than a pixel the image looks blocky. If it is much smaller, distant surfaces sparkle and crawl horribly.
8. The fix for the second problem is to store the same image several times, each half the size of the last. That stack is a **mipmap**.
9. Textures do more than colour. One can store which way each tiny bump of the surface faces, giving fine detail with no extra triangles.
10. Textures are the single biggest consumer of graphics memory in any real game.

■ 22.7.2 PLAIN – a picture in your head

1. Think of wrapping a present with patterned paper.
2. The box is the mesh. The paper is the texture. Deciding which part of the paper touches which face of the box is UV mapping.
3. To do it properly you first cut the paper into a flat pattern that unfolds to fit the box. Artists really do this, and call it unwrapping.
4. Now step back across the room. The pattern is too fine to see and appears to shimmer. So instead you use a version of the paper printed with a coarser pattern for viewing from far away.
5. You keep a whole set: full detail, half detail, quarter detail, down to a single average-coloured square. You choose by distance.

Where this comparison breaks: paper has no thickness and no lighting behaviour. A modern surface uses four or five separate images at once for the same patch: one for colour, one for the direction of tiny bumps, one for how metallic it is, one for how rough it is, and often one for how much light gets trapped in crevices. They are all sampled with the same UV coordinates.

■ 22.7.3 PLAIN – a worked example

1. Take a 4096 by 4096 texture. That is 16,777,216 texels.
2. Stored as four bytes per texel, uncompressed, it needs 64 mebibytes.
3. Now add the mipmap chain: 2048 squared, then 1024, then 512, and so on to 1x1. Each level is a quarter of the previous, so the whole chain adds one third. Total 85.3 mebibytes.
4. Now compress it. Here are the real costs for the same image:

Format	Bits per texel	With full mipmaps
RGBA8 uncompressed	32	85.3 MiB
BC7 (high quality)	8	21.3 MiB
BC1 (old, no alpha)	4	10.7 MiB
ASTC 8x8 (mobile)	2	5.3 MiB

5. A game character might use five such textures. Uncompressed that is 426 mebibytes for one character. In BC7 it is 107 mebibytes. Compression is not optional.
6. Now filtering. Suppose a pixel lands at UV (0.30012, 0.50008) on a 1024 by 1024 texture. That is texel position (307.32, 512.08).
7. Nearest filtering picks texel (307, 512) and ignores the fractions.
8. Bilinear filtering blends four texels: (307,512), (308,512), (307,513), (308,513), with weights $(1-0.32)(1-0.08) = 0.6256$, $0.32 \times 0.92 = 0.2944$, $0.68 \times 0.08 = 0.0544$, and $0.32 \times 0.08 = 0.0256$. Those four sum to 1.
9. Trilinear filtering does that twice, on the two nearest mipmap levels, then blends between them by the fractional level. Eight texels, one result.
10. Anisotropic filtering takes several trilinear samples along the direction the surface is stretched. At 16x it may take up to sixteen of them. It is the single best-value image quality setting in any game.

■ 22.7.4 PLAIN – what is really happening inside

1. Mipmaps exist because of sampling theory, not because of laziness. If the texture has more detail than the pixel grid can represent, that detail folds back as false patterns. That is aliasing.
2. The hardware picks a mipmap level automatically. It compares the UV of the pixel to the right and the pixel below, using the 2x2 quad from section 22.5, and takes the base-2 logarithm of the larger change.
3. A **normal map** stores, in the red, green and blue channels, the x, y and z of a tiny surface normal, in a coordinate frame attached to the surface.
4. Because it is only a normal, not real geometry, the silhouette of the object never changes. Look along a bumpy wall and the edge is still perfectly flat.
5. A **bump map** or height map stores only height, and the normal is derived from the slope. It is older, smaller, and less exact.
6. A **cube map** is six square textures forming a box around a point. Look up with a direction vector instead of a UV pair. This is how skies and reflections are stored.
7. Block compression works on 4x4 blocks. Each block stores two endpoint colours and a small index per texel that says how far between them the texel lies.

- That fixed block size is the point: any texel's address can be computed directly, so the texture unit can decompress a single block on demand without touching the rest of the image.

■ 22.7.5 TECHNICAL – the engineer's version

- Texture mapping was introduced in Edwin Catmull's 1974 University of Utah doctoral thesis, alongside the depth buffer. Mipmapping came from Lance Williams, "Pyramidal Parametrics", SIGGRAPH 1983. The name is from the Latin "multum in parvo", much in little.
- Bump mapping is James Blinn, "Simulation of Wrinkled Surfaces", SIGGRAPH 1978. Environment mapping is Blinn and Newell, 1976.
- Compression families by platform, all standards:

Family	Where used	Typical rate
BC1-BC7 (DXT/S3TC)	PC, consoles	4 or 8 bpp
ETC2 / EAC	Android, OpenGL ES 3	4 or 8 bpp
ASTC	Mobile, Vulkan	0.89 to 8 bpp
BC6H	HDR textures	8 bpp

- BC6H stores high dynamic range data and is the correct choice for environment and light probe maps. BC5 stores two channels and is the standard choice for normal maps, with z reconstructed in the shader as $\sqrt{1 - x^2 - y^2}$.
- Anisotropic filtering degree is capped by hardware, commonly 16x on desktop. The cost is extra texture fetches, so it is bandwidth-bound, not arithmetic-bound. Measured cost in modern games is typically under 3 percent of frame time at 16x.
- Real memory budgets, observed rather than specified. In 2026, a demanding PC title at 4K with ray tracing and the highest texture setting commonly reports 10 to 14 gigabytes of video memory allocated, of which the large majority is textures. At 1080p with medium textures the same title fits in 6 to 8 gigabytes.
- This is why 8 gigabyte cards became controversial from 2023 onward. The argument is not about raw speed; it is about what happens when the working set does not fit and textures must stream over PCIe every frame, producing stutter and visibly blurred surfaces.
- Virtual texturing, sparse textures and the Direct3D 12 Sampler Feedback feature, added in 2020, let an engine load only the mip levels and tiles actually sampled. Unreal Engine 5's Nanite and Virtual Shadow Maps rely on this class of technique.
- Tools: texconv and compressionator for offline compression; RenderDoc shows every bound texture with its exact format and mip chain; nvidia-smi and radeontop report video memory in use, though allocated is not the same as actively needed.

■ 22.7.6 WORDS – remember these

- Texel — a pixel of a texture — an addressable element of a texture image, distinct from a screen pixel.
- UV coordinate — where on the picture this corner sits — normalized texture coordinates in $[0,1]$, interpolated per fragment.
- Mipmap — the stack of shrinking copies — a prefiltered image pyramid, each level half the linear size, selected by screen-space derivatives.
- Bilinear filtering — blend the four nearest dots — weighted average of the 2x2 texel neighbourhood within one mip level.
- Trilinear filtering — blend between two shrink levels too — bilinear in each of two adjacent mip levels, then linearly interpolated between them.

6. Anisotropic filtering — extra samples along the stretched direction — multiple trilinear taps along the axis of greatest UV derivative, up to the hardware cap.
7. Normal map — a picture of which way each speck faces — a texture encoding tangent-space normals, giving lighting detail without geometry.
8. Block compression — fixed-size squeezed tiles — lossy fixed-rate texture formats decoded in the texture unit, allowing random access.

22.8 Making it look real

■ 22.8.1 PLAIN – in simple words

1. Once you can put a coloured triangle on screen, everything else is about deciding what colour, and which triangle wins.
2. Which triangle wins is settled by the **depth buffer**: a second image that stores, for every pixel, how far away the nearest thing drawn so far is.
3. Draw order stops mattering. Each pixel simply keeps the nearest.
4. When two surfaces are at almost the same distance, rounding error makes the winner flicker between them. That is **z-fighting**.
5. You can also skip any triangle whose back is facing you, since you cannot see the inside of a solid object. That is **back-face culling**, and it removes about half of all triangles for free.
6. Colour comes from a lighting model: a formula combining a base ambient level, a part that depends on the angle to the lamp, and a shiny highlight.
7. Shadows are handled by rendering the scene once from the lamp's position, storing only depth, then checking each pixel against that.
8. Modern materials are described not by made-up numbers but by physical ones: base colour, whether it is metal, and how rough it is.
9. Light that has bounced off other surfaces is called global illumination and is the hardest part of the whole subject.
10. Finally, effects applied to the finished image, such as blur and colour grading, are called post-processing.

■ 22.8.2 PLAIN – a picture in your head

1. Think of painting a stage set with an assistant holding a tape measure.
2. Every time you are about to paint a square, the assistant tells you how far away the last thing painted on that square was.
3. If your new object is nearer, you paint and the assistant writes down the new distance. If it is further, you skip it.
4. That is the depth buffer, and it means you can paint the scenery in any order at all.
5. Now for shadows: you walk to where the lamp is, look at the set from there, and write down how far the nearest surface is in every direction.
6. Back at your easel, for each square you ask: is this point further from the lamp than the nearest thing the lamp could see in that direction? If yes, it is in shadow.

Where this comparison breaks: the tape measure has limited precision. Two surfaces a millimetre apart at a hundred metres get the same reading, and the painter flickers between them. That is z-fighting, and it is not a bug in the idea, it is a limit of storing distance in a finite number of bits.

■ 22.8.3 PLAIN – a worked example

1. Take a surface with normal $N = (0, 1, 0)$, light direction $L = (0.6, 0.8, 0)$, and view direction $V = (0, 1, 0)$. Shininess exponent 32.
2. Diffuse term: $N \cdot L = 0.8$. Lambert's law, straight from section 22.4.
3. Phong specular uses the mirror-reflected light direction: $R = 2(N \cdot L)N - L = (0, 1.6, 0) - (0.6, 0.8, 0) = (-0.6, 0.8, 0)$.
4. $R \cdot V = (-0.6 \times 0) + (0.8 \times 1) = 0.8$. Raise to the power 32: $0.8^{32} = 0.00079$. Almost no highlight.
5. Blinn-Phong instead uses the halfway vector between light and view: $H = \text{normalize}(L + V) = \text{normalize}(0.6, 1.8, 0) = (0.3162, 0.9487, 0)$.
6. $N \cdot H = 0.9487$. Raise to the power 32: $0.9487^{32} = 0.185$. A clearly visible highlight.
7. Those two answers differ by a factor of 230. The models are not interchangeable at the same exponent.
8. To match Blinn-Phong to Phong you need roughly four times the exponent. Check: $0.9487^{128} = 0.00118$, which is close to Phong's 0.00079. The rule of thumb holds.
9. Full Blinn-Phong for this pixel, with ambient 0.05, diffuse strength 0.8, specular strength 0.5: result $= 0.05 + 0.8 \times 0.8 + 0.5 \times 0.185 = 0.05 + 0.64 + 0.0925 = 0.7825$.

■ 22.8.4 PLAIN – what is really happening inside

1. The formulas, written out plainly. Lambert diffuse: $k_d * \max(\text{dot}(N, L), 0)$. Phong specular: $k_s * \text{pow}(\max(\text{dot}(R, V), 0), \text{shininess})$ with $R = \text{reflect}(-L, N)$. Blinn-Phong specular: $k_s * \text{pow}(\max(\text{dot}(N, H), 0), \text{shininess})$ with $H = \text{normalize}(L + V)$.
2. Blinn-Phong is cheaper, because computing H is fewer operations than reflecting a vector, and it behaves better at grazing angles where Phong produces an unnatural hard cut-off.
3. Shadow mapping in exact steps: render the scene from the light, store depth only; then when shading, transform the pixel's world position into the light's space, look up the stored depth, and compare.
4. Two errors follow immediately. If the comparison is exact, surfaces shadow themselves in a stripey pattern called shadow acne, fixed with a small bias. Too much bias and shadows detach from their objects, called peter-panning.
5. Physically based rendering replaces made-up shininess with measurable parameters: **albedo** (the base colour a surface reflects diffusely), **metalness** (0 for non-metal, 1 for metal, nothing in between in the standard model), and **roughness** (0 mirror, 1 completely scattered).
6. The reason it works is energy conservation: a surface may not reflect more light than it receives. Older models happily broke that and artists corrected it by eye, which is why old games look different under different lighting.
7. Global illumination is light that has bounced at least once. Direct lighting is one lamp to one surface to the eye. Indirect is lamp to wall to floor to eye, and it is what makes a room look like a room.
8. Ambient occlusion is a cheap stand-in: darken creases and contact points, because less bounced light reaches them.

■ 22.8.5 TECHNICAL – the engineer's version

1. The depth buffer, or z-buffer, is from Edwin Catmull's 1974 thesis, with Wolfgang Strasser describing the same idea independently in 1974.
2. Z-fighting is a precision problem. With a 24-bit fixed-point buffer, near 0.1 and far 1000, the gap between representable depths is about 6 millimetres at 100 metres and about 60 centimetres at 1000 metres. Two surfaces closer together than that gap cannot be told apart.

3. Fixes, in order of preference: move the near plane out; use reversed-Z with a 32-bit float depth buffer; use a depth bias with `glPolygonOffset`; last resort, physically separate the surfaces.
4. Back-face culling uses the sign of the signed screen-space area of the triangle after projection. It removes about half the triangles of a closed mesh and costs nothing, so it is on by default in every engine.
5. Lighting model history: Henri Gouraud published per-vertex interpolated shading in 1971. Bui Tuong Phong published his reflection model and per-pixel normal interpolation in Communications of the ACM in June 1975, dying of leukaemia in the same year at 32. James Blinn published the halfway-vector variant at SIGGRAPH in 1977.
6. Cook and Torrance published the first physically motivated microfacet model in 1982. The parameterization the industry actually uses comes from Brent Burley's "Physically Based Shading at Disney", SIGGRAPH 2012, usually called the Disney principled BRDF.
7. The standard real-time specular term today is the Cook-Torrance form $D * F * G / (4 (N \cdot L)(N \cdot V))$, with GGX for the normal distribution D (Walter and others, 2007), the Schlick approximation for Fresnel F (1994), and a Smith height-correlated term for G .
8. Shadow map filtering: percentage closer filtering (Reeves, Salesin and Cook, 1987) samples several nearby depths and averages the comparison results, not the depths. Cascaded shadow maps split the view frustum into 3 or 4 ranges, each with its own map, and are the standard for outdoor sunlight.
9. Screen space ambient occlusion was introduced by Crytek in Crysis in 2007. Ground truth ambient occlusion (GTAO, 2016) is the common modern choice.
10. Post-processing effects and their real costs at 4K, order of magnitude, on a mid-range 2025 card: tone mapping under 0.1 ms, bloom 0.3 to 0.8 ms, motion blur 0.5 to 1.5 ms, depth of field 0.5 to 2 ms, screen space reflections 1 to 3 ms. In a 16.7 ms frame budget these add up quickly.

■ 22.8.6 WORDS – remember these

1. Depth buffer — a per-pixel record of nearest distance — the z-buffer, tested and optionally written per fragment, making draw order irrelevant.
2. Z-fighting — flickering between two surfaces — depth quantization error when two surfaces fall within one representable depth step.
3. Back-face culling — skip triangles facing away — discarding primitives by the sign of their projected signed area.
4. Diffuse — brightness by angle to the lamp — the Lambertian term, proportional to $\max(N \cdot L, 0)$.
5. Specular — the shiny highlight — the mirror-like lobe, modelled by Phong, Blinn-Phong or a microfacet distribution such as GGX.
6. Shadow map — a depth picture taken from the lamp — a depth buffer rendered from the light, compared against per fragment to test occlusion.
7. Albedo, metalness, roughness — colour, is-it-metal, how scattered — the three core parameters of the metallic-roughness physically based material model.
8. Ambient occlusion — darkening in the creases — an estimate of how much of the surrounding hemisphere is blocked at each point.

22.9 Ray tracing

■ 22.9.1 PLAIN – in simple words

1. Rasterisation asks: for each triangle, which pixels does it cover?
2. Ray tracing asks the opposite: for each pixel, which triangle does it see?
3. To answer that, you shoot an imaginary straight line from the eye through that pixel and find the first thing it hits.
4. Once you have hit something, you can shoot more lines: toward the lamp to test for shadow, or in the mirror direction to find a reflection.
5. That is why ray tracing gets reflections, refraction and correct shadows almost for free, while rasterisation needs a separate trick for each.
6. The cost is brutal. Testing one line against every triangle in a scene of ten million triangles is ten million tests, for every one of two million pixels.
7. The fix is to sort the triangles into a tree of nested boxes, so that one test against a big box can eliminate a million triangles at once.
8. Even with the tree, this was far too slow for real time for about forty years.
9. In 2018 graphics chips gained circuits dedicated to walking that tree and testing lines against triangles, and real-time ray tracing became possible.
10. Games do not use it for everything. They rasterise most of the image and use rays only for shadows, reflections or bounced light. That is hybrid rendering.

■ 22.9.2 PLAIN – a picture in your head

1. Imagine finding which shop in a city a laser pointer is aimed at.
2. The stupid method is to visit all forty thousand shops and check each one.
3. The sensible method is a nested set of boxes. Is the beam inside the city boundary? Yes. Which district? North. Which street in that district? Which building on that street? Which shop in the building?
4. Five or six questions instead of forty thousand. Each question halves or better the remaining candidates.
5. That nested set of boxes is called a bounding volume hierarchy.

Where this comparison breaks: the city does not move. A game scene does, every frame, and the hierarchy has to be rebuilt or repaired for anything that moved, by the processor or by the chip itself. That rebuild cost is a real and significant part of the frame time and is invisible in screenshots.

■ 22.9.3 PLAIN – a worked example

1. A ray is written as $P(t) = O + tD$, where O is the origin, D is a unit direction, and t is distance travelled.
2. A sphere of radius r centred at C is every point where the distance to C equals r .
3. Substituting one into the other gives a quadratic in t : $t^2 + 2t(D \cdot (O - C)) + |O - C|^2 - r^2 = 0$.
4. Real numbers. Eye at the origin $O = (0,0,0)$. Direction $D = (0,0,-1)$. Sphere at $C = (0,0,-10)$ with radius 2.
5. $O - C = (0,0,10)$. $D \cdot (O - C) = -10$. So $b = -20$ and $c = 100 - 4 = 96$.
6. Discriminant $= b^2 - 4c = 400 - 384 = 16$. Its square root is 4.
7. $t = (20 \pm 4) / 2$, so $t = 8$ or $t = 12$.
8. The nearer hit is $t = 8$, at the point $(0, 0, -8)$. Correct: the front of the sphere is at $z = -10 + 2 = -8$.
9. A negative discriminant means the ray misses. A discriminant of exactly zero means it grazes the surface.

10. For triangles the standard method is Moller-Trumbore, published in 1997. It solves for the barycentric coordinates and the distance in one go, using two cross products and needing no precomputed plane equation.

■ 22.9.4 PLAIN – what is really happening inside

1. A bounding volume hierarchy is a tree. Each node holds a box that contains everything below it. Leaves hold a handful of triangles.
2. Traversal keeps a small stack. Test the ray against the two child boxes; descend into the nearer one first; if you find a hit closer than the far box, you never open the far box at all.
3. Ray-box tests are cheap: six comparisons using the slab method. Ray-triangle tests are more expensive, so the tree exists to minimize how many you do.
4. In a well-built tree, a ray touches roughly 20 to 60 nodes and 2 to 10 triangles in a scene of millions.
5. The 2018 hardware does two specific things. One unit walks the tree and does box tests. Another does triangle intersection. Both run while the shader cores do something else.
6. Path tracing is ray tracing taken to its logical end: at every hit, pick a random new direction, keep going, and average over many random paths. It converges to the true answer given enough samples.
7. Real time can afford about one to two paths per pixel. That gives a violently noisy image.
8. So the last stage is a **denoiser**: an algorithm, today usually a small neural network, that turns a noisy one-sample image plus depth and normal information into a clean one.
9. Without the denoiser, real-time path tracing would be useless. It is not a polish step, it is a load-bearing part of the system.

■ 22.9.5 TECHNICAL – the engineer’s version

1. Arthur Appel described ray casting for shading solids in 1968. Turner Whitted added recursive reflection and refraction in “An Improved Illumination Model for Shaded Display”, Communications of the ACM, June 1980. His famous test image took 74 minutes on a VAX 11/780.
2. James Kajiya formalized the rendering equation and path tracing in “The Rendering Equation”, SIGGRAPH 1986. Everything since is a method of approximating that one integral.
3. Bounding volume hierarchies for ray tracing go back to Rubin and Whitted in 1980 and Kay and Kajiya in 1986. The surface area heuristic, the standard build quality metric, is from Goldsmith and Salmon, 1987.
4. Hardware acceleration arrived with NVIDIA Turing, announced 20 August 2018 and shipping 20 September 2018, in the RTX 2080 and 2080 Ti. NVIDIA quoted 10 billion rays per second for the 2080 Ti.
5. AMD added ray accelerators with RDNA 2 in November 2020; RDNA 4, in March 2025, doubled ray-triangle throughput per compute unit and added oriented bounding boxes. Intel shipped ray tracing units with Arc Alchemist in 2022.
6. API support is a standard: DirectX Raytracing (DXR) announced March 2018, Vulkan Ray Tracing extensions ratified November 2020, and Metal ray tracing from 2020. All expose a two-level acceleration structure: a bottom level per mesh, a top level of instances.
7. The shader types DXR defines are ray generation, intersection, any-hit, closest-hit and miss. Traversal itself is fixed function and deliberately not programmable, which is what allows it to be a dedicated circuit.
8. Denoising: the practical breakthrough was Chaitanya and others, “Interactive Reconstruction of Monte Carlo Image Sequences Using a Recurrent Denoising Autoencoder”, SIGGRAPH 2017. NVIDIA’s Ray Reconstruction, shipped in DLSS 3.5 in September 2023, replaces the hand-written denoiser with a trained network.

9. Honest performance picture, as of 2026. Full path tracing at 4K on the fastest consumer card of the day still typically renders internally at 1080p or lower and relies on both denoising and upscaling to reach the output resolution. Native-resolution real-time path tracing is not a solved problem. Vendors showing “4K path traced” numbers are almost always showing an upscaled image, and independent testers routinely point this out.

■ 22.9.6 WORDS – remember these

1. Ray casting — shoot a line, find what it hits — solving for the nearest surface intersection along a parameterized ray.
2. Bounding volume hierarchy — nested boxes for fast searching — a tree of axis-aligned boxes over primitives, traversed to prune intersection tests.
3. Path tracing — follow random bounces and average — Monte Carlo integration of the rendering equation.
4. Hybrid rendering — rasterise most, trace some — using rasterisation for primary visibility and rays only for selected effects.
5. RT core — the circuit that walks the tree — fixed-function hardware for ray-box and ray-triangle intersection, introduced in 2018.
6. Denoiser — clean up a noisy estimate — a spatial and temporal filter, now usually a neural network, reconstructing an image from few samples.

22.10 Upscaling and frame generation

■ 22.10.1 PLAIN – in simple words

1. Rendering costs roughly scale with pixel count. Going from 1080p to 4K is four times the pixels and roughly four times the cost.
2. So there is an obvious cheat: render at a lower resolution and enlarge the result cleverly.
3. Naïve enlargement looks soft and blurry, so for a long time nobody did it.
4. The clever version uses information from previous frames. Slightly shift the camera by a fraction of a pixel each frame, and over several frames you have sampled the scene far more finely than one frame allows.
5. That trick, on its own, is called temporal anti-aliasing. It reuses history to remove jagged edges.
6. Upscalers use the same history but also target a higher output resolution, and the modern ones use a trained neural network to decide how to combine old and new samples.
7. There is a second, more controversial idea: instead of rendering more frames, invent them. Look at two real frames and manufacture the picture in between.
8. That makes the counter say a bigger number and makes motion look smoother, but it cannot make the game respond faster. It usually makes it respond slower.
9. Neither trick is free. Both introduce specific, visible artefacts.

■ 22.10.2 PLAIN – a picture in your head

1. Think of photographing a page of small print in poor light.
2. One photo is grainy and unreadable. But take eight photos, each shifted by a fraction of a millimetre, and stack them, and the text becomes sharp.
3. That is temporal accumulation. You bought detail with time instead of with sensor resolution.
4. Now suppose the page is being pulled away while you shoot. To stack the photos you must first work out how far the page moved between each one, and slide each photo back into place.
5. That correction is called reprojection, and the arrows telling you how far things moved are motion vectors.

- When a corner of the page was hidden behind your thumb in the earlier photos, there is no history for that region, and the stack has nothing to add.

Where this comparison breaks: a page moves rigidly. A game scene has objects moving in front of each other, transparent surfaces with no motion vectors at all, shadows that move differently from the objects casting them, and particles. Each of these breaks the correction in its own way, which is why upscaler artefacts cluster around fast movement, thin objects, and things seen through glass.

■ 22.10.3 PLAIN – a worked example

- Output resolution 3840 by 2160, which is 8,294,400 pixels.
- The standard quality presets and their internal render sizes:

Preset	Scale factor	Internal at 4K
Quality	0.667	2560 x 1440
Balanced	0.58	2227 x 1253
Performance	0.50	1920 x 1080
Ultra Performance	0.33	1280 x 720

- Performance mode renders 2,073,600 pixels instead of 8,294,400. That is 25 percent of the work for the parts of the frame that scale with resolution.
- In practice the measured speed-up is smaller, typically 1.6x to 2.2x rather than 4x, because vertex work, shadow map rendering and the upscaler itself do not shrink.
- Now frame generation, and the latency arithmetic that people get wrong.
- Suppose the game natively runs at 60 frames per second, one frame every 16.7 milliseconds.
- To insert a frame between real frames N and N+1, the system must already have rendered N+1. So it holds N+1 back, shows the invented frame first, then shows N+1.
- The counter now reads 120 frames per second. But the newest real image reaches your eye about one frame later than it would have, plus the time to generate the fake one.
- Measured end-to-end click-to-photon latency typically rises by 8 to 15 milliseconds with single frame generation, before any latency-reduction feature claws some back.
- So the honest statement is: frame generation improves smoothness, not responsiveness, and slightly harms responsiveness. It is at its best when the base frame rate is already comfortable, and at its worst below about 45 frames per second where it is most tempting.

■ 22.10.4 PLAIN – what is really happening inside

- The renderer jitters the projection matrix by a sub-pixel offset each frame, following a low-discrepancy sequence such as Halton.
- It also outputs a motion vector for every pixel: where that surface point was in the previous frame, in screen coordinates.
- The upscaler takes this frame's low-resolution colour, its depth, its motion vectors, and the previous output frame.
- It warps the previous output using the motion vectors, then decides per pixel how much of that warped history to trust.
- The old approach used hand-written rules: colour clamping to a neighbourhood box, rejecting history that looks too different. It causes ghosting when it trusts too much and flicker when it trusts too little.
- The new approach feeds all of that into a small neural network trained on matched pairs of low-resolution input and high-quality reference images.

7. Frame generation additionally computes an optical flow field between two real frames, warps both toward the midpoint, and fills disocclusions with a network.
8. The generated frame is never used as history for the next real frame. It is displayed and discarded.

■ 22.10.5 TECHNICAL – the engineer’s version

1. Timeline, all verifiable release dates. DLSS 1.0, February 2019, per-game trained, widely judged poor. DLSS 2.0, March 2020, generic temporal network, the version that made the idea work. DLSS 3, October 2022, adds Frame Generation, RTX 40 only. DLSS 3.5, September 2023, adds Ray Reconstruction. DLSS 4, January 2025, replaces the convolutional network with a transformer model and adds Multi Frame Generation on RTX 50.
2. As of August 2026 NVIDIA’s current release is DLSS 4.5, with a second generation transformer model and a dynamic multi-frame multiplier advertised up to 6x, meaning five generated frames per rendered frame.
3. AMD FidelityFX Super Resolution: FSR 1.0, June 2021, spatial only. FSR 2.0, May 2022, temporal, open source, runs on competitors’ hardware. FSR 3, September 2023, adds frame generation. FSR 4, March 2025, machine learning based and limited to RDNA 4.
4. Intel XeSS launched August 2022 with two code paths: XMV matrix instructions on Intel Arc, and a DP4a integer path elsewhere that is measurably weaker. XeSS 2, December 2024, added frame generation and a low-latency mode.
5. Latency reduction features are separate technologies: NVIDIA Reflex (2020), AMD Anti-Lag, Intel Xe Low Latency. They shorten the render queue. They do not remove the frame-holding cost of frame generation.
6. Honest quality comparison, as of 2026. Independent testing outlets including Digital Foundry and Hardware Unboxed consistently report the transformer-based DLSS as the most stable and detailed at a given preset, FSR 4 as close behind on supported hardware and a large jump over FSR 3, and XeSS as good on Intel hardware and weaker on its fallback path. This is expert consensus from controlled comparison, not a measured standard, and reasonable people disagree about specific titles.
7. The known artefact list, which applies to all of them in some degree: ghosting trails behind fast movement; smearing where an object uncovers background; shimmer on thin geometry such as fences and wires; loss of fine texture detail at aggressive presets; and, for frame generation, warped heads-up-display elements and text in the generated frames.
8. The cost of running the upscaler itself is real: roughly 1 to 2 milliseconds per frame at 4K output on recent hardware. Below about 40 frames per second this overhead eats a noticeable share of the gain.
9. There is a genuine industry disagreement worth stating. One camp argues these techniques are legitimate reconstruction, no different in kind from mipmaps or texture compression. The other argues they let publishers ship games that cannot run acceptably at native resolution, and that quoting frame rates with generated frames included is misleading. Both positions are held by serious people.

■ 22.10.6 WORDS – remember these

1. Temporal anti-aliasing — reuse earlier frames to smooth edges — accumulation of sub-pixel jittered samples across frames with motion-vector reprojection.
2. Motion vector — where this pixel was last frame — a per-pixel screen-space displacement produced by the renderer for reprojection.
3. Reprojection — slide the old frame into place — warping history buffers by motion vectors before reuse.

4. Ghosting — trails behind moving things — an artefact of trusting stale history whose motion vectors were wrong or missing.
5. Disocclusion — newly revealed area with no history — a region uncovered this frame, which temporal methods must invent.
6. Frame generation — invent a picture between two real ones — interpolating an intermediate frame from optical flow, raising frame counters and latency together.

22.11 Inside a real graphics chip

■ 22.11.1 PLAIN – in simple words

1. A graphics chip is not one big pool of lanes. It is many small identical blocks, each a small computer in its own right.
2. NVIDIA calls a block a streaming multiprocessor. AMD calls it a compute unit. Intel calls it an Xe core. They are the same idea.
3. Each block has its own instruction scheduler, its own arithmetic lanes, its own registers, its own small fast memory, and its own texture unit.
4. The marketing number, “21,760 CUDA cores”, is the total count of arithmetic lanes, not the number of independent computers.
5. The real count of independent computers on that chip is 170.
6. Alongside the general lanes sit two kinds of special circuit: matrix units for machine learning, and ray tracing units for the tree walking of section 22.9.
7. All the blocks share a large cache and a memory controller that talks to the card’s own memory chips.
8. That memory is the real bottleneck of almost every workload, and its speed is the number you should look at first.

■ 22.11.2 PLAIN – a picture in your head

1. Think of a large open-plan office split into 170 identical pods.
2. Each pod has 128 desks, one supervisor per 32 desks, a shared whiteboard, and a shelf of reference books.
3. The whiteboard is shared memory: fast, tiny, and only visible inside the pod.
4. Down the corridor is the building library, shared by all pods. That is the L2 cache.
5. Outside the building is the city archive. That is the card’s memory. Fetching from there takes a long walk, so you send someone and get on with other work.

Where this comparison breaks: the desks in a pod are not independent workers. Groups of 32 share one supervisor who reads out one instruction at a time. The pod is more like a rowing eight than an office.

■ 22.11.3 PLAIN – a worked example

1. Here is a real specification sheet, the NVIDIA GeForce RTX 5090, launched 30 January 2025 at 1,999 US dollars, decoded line by line.

Line on the sheet	Value	What it really means
Architecture	Blackwell	Design generation, 2025
GPU die	GB202	The specific silicon
CUDA cores	21,760	FP32 lanes in total

Line on the sheet	Value	What it really means
SM count	170	Independent blocks
Tensor cores	680	4 per SM, matrix units
RT cores	170	1 per SM, tree walkers
Boost clock	2.41 GHz	Peak, not sustained
Memory	32 GB GDDR7	Card's own memory
Memory bus	512-bit	Wires to the memory
Bandwidth	1,792 GB/s	Bytes readable per second
L2 cache	96 MB	Chip-wide shared cache
Transistors	92.2 billion	Complexity
Die size	750 mm ²	Physical area of silicon
Interface	PCIe 5.0 x16	Link to the processor
Total board power	575 W	Heat you must remove

- Now the arithmetic that the sheet does not show you.
- 21,760 divided by 170 gives 128 FP32 lanes per block. 680 divided by 170 gives 4 matrix units per block.
- Peak FP32 rate = lanes $\times$ 2 (a multiply-add counts as two) $\times$ clock = 21,760 $\times$ 2 $\times$ 2.41 GHz = 104.8 TFLOPS. This matches the published figure exactly, which tells you it is a paper number from a formula, not a measurement.
- Bandwidth = bus width $\times$ per-pin data rate / 8 = 512 $\times$ 28 Gbps / 8 = 1,792 GB/s. The 28 gigabits per second per pin is the GDDR7 speed grade used.
- A 512-bit bus with 32-bit memory chips means 16 chips. 32 GB across 16 chips is 2 GB each. The board layout follows directly from the bus width.
- Now the ratio that matters most: 104.8 TFLOPS against 1,792 GB/s is about 58 floating-point operations available for every byte you can read.
- So any calculation doing fewer than 58 operations per byte of data is limited by memory, not by arithmetic. Most real ones are. This ratio is the whole of the roofline model in one line.
- Power: 575 watts over 750 square millimetres is about 0.77 watts per square millimetre, which is why these cards carry three-slot coolers and vapour chambers.

■ 22.11.4 PLAIN – what is really happening inside

- Inside one block, the lanes are split into four partitions of 32. Each partition has one warp scheduler that issues one instruction per cycle to its own group.
- The register file is enormous by processor standards: 256 KB per block on recent NVIDIA parts, so more register storage than L1 cache.
- That is deliberate. Registers are what let dozens of thread groups stay resident so the scheduler always has someone ready to run.
- Shared memory is a scratchpad the programmer controls, not a cache. On recent NVIDIA consumer parts it shares a 128 KB pool with L1, split as you choose.
- Tensor cores do not do one multiply. They do a whole small matrix multiply per instruction, for example a 16 $\times$ 8 by 8 $\times$ 16 tile, accumulating into a result.
- RT cores do box and triangle intersection while the general lanes are free to do other work.
- The memory controller is split into many channels. Each GDDR chip has its own path, and requests from many blocks are coalesced and reordered to keep every channel busy.
- The PCIe link is not for the drawing. It is for uploading geometry and textures, sending commands, and reading results back.

■ 22.11.5 TECHNICAL – the engineer’s version

1. Memory technologies and real bandwidth, all from published specifications:

Part and memory	Bus width	Bandwidth
Arc B580, GDDR6 19 Gbps	192-bit	456 GB/s
RX 9070 XT, GDDR6 20 Gbps	256-bit	640 GB/s
RTX 5080, GDDR7 30 Gbps	256-bit	960 GB/s
RTX 5090, GDDR7 28 Gbps	512-bit	1,792 GB/s
H100 SXM, HBM3	5120-bit	3,350 GB/s
B200, HBM3e	wide stacks	about 8,000 GB/s

2. JEDEC published the GDDR7 standard, JESD239, in March 2024. It is the first JEDEC DRAM standard to use PAM3 signalling, three voltage levels carrying 1.5 bits per symbol, and it specifies up to 32 Gb/s per pin.
3. JEDEC published HBM4, JESD270-4, in April 2025, with revision 4A in December 2025. It doubles the per-stack interface to 2048 bits, runs up to 8 Gb/s per pin, reaches up to 2 TB/s per stack, and allows up to 64 GB per stack with 16-high 32 Gb dies.
4. HBM is stacked vertically on an interposer beside the die, which is why it is confined to expensive datacentre parts. GDDR is separate packages on the board, which is why it is on everything else.
5. Naming honesty. A “CUDA core” is one FP32 lane inside a SIMD datapath with no independent program counter. AMD’s “stream processor” is the same thing. Neither is a core in the sense used for a processor. This is marketing vocabulary that stuck.
6. TFLOPS honesty. AMD publishes 48.7 TFLOPS FP32 for the RX 9070 XT, which has 4,096 stream processors at up to 2.97 GHz. The plain formula gives 24.3. The published figure counts dual-issue FP32, which only applies when the compiler can pair instructions. NVIDIA’s 104.8 TFLOPS for the RTX 5090 uses the plain formula. Comparing the two numbers directly is not valid.
7. PCIe 5.0 x16 has a raw signalling rate of 32 GT/s per lane, giving a theoretical 63 GB/s each way after 128b/130b encoding, and roughly 50 to 55 GB/s achievable in practice.
8. Power delivery on high-end 2025 cards uses the 12V-2x6 connector, the revised form of 12VHPWR, rated at 600 watts. The original 12VHPWR connector had well-documented melting failures on RTX 4090 cards in 2022, traced to incomplete insertion and uneven current sharing between pins.
9. Tools: `nvidia-smi -q` dumps clocks, power, temperature and memory; `nvt` and `radeontop` give live views; Nsight Compute reports achieved occupancy, memory throughput as a percentage of peak, and the exact stall reasons per kernel.

■ 22.11.6 WORDS – remember these

1. Streaming multiprocessor — one pod of lanes — NVIDIA’s independent scheduling and execution block; AMD’s compute unit, Intel’s Xe core.
2. CUDA core — one arithmetic lane — an FP32 lane within a SIMD datapath, not an independently scheduled core.
3. Tensor core — the matrix multiplier — a fixed-tile matrix multiply-accumulate unit, first shipped on NVIDIA Volta in 2017.
4. Shared memory — the pod’s whiteboard — programmer-managed on-chip scratchpad shared by a workgroup.
5. GDDR — fast memory chips on the board — graphics DDR DRAM; GDDR7 uses PAM3 and reaches 32 Gb/s per pin under JESD239.

6. HBM — memory stacked next to the die — high bandwidth memory on an interposer; HBM4 under JESD270-4 reaches 2 TB/s per stack.
7. Roofline — the arithmetic-per-byte break-even — the ratio of peak compute to peak bandwidth that decides whether a kernel is compute or memory bound.

22.12 The software stack

■ 22.12.1 PLAIN – in simple words

1. Your program never talks to the graphics chip directly. It talks to a library with a defined set of functions, called a graphics API.
2. Behind that library sits the **driver**, written by the chip maker, which turns those calls into the chip's own commands.
3. Old APIs let you say “draw this” one thing at a time, and the driver did an enormous amount of guessing and bookkeeping.
4. New APIs make you state everything up front, in explicit objects, and then record long lists of commands yourself.
5. That is more work for the programmer and much less guessing for the driver, which means less overhead and more predictable timing.
6. Commands are not executed as you write them. They are recorded into a buffer, submitted in a batch, and executed later by the chip.
7. Because the chip runs behind the processor, at any moment one of them is waiting for the other. Which one is waiting decides how you make the program faster.
8. Shaders must be turned into the chip's real machine code at some point. If that happens the first time you enter a new area, the game stutters.

■ 22.12.2 PLAIN – a picture in your head

1. Think of ordering from a kitchen through a waiter.
2. The old API is a waiter who takes one instruction at a time, remembers the whole table's preferences, and silently fixes contradictions.
3. That waiter is convenient and unpredictable. You never know when they will disappear to sort something out.
4. The new API makes you write the entire order on a card, in full, and hand it over. The waiter only carries cards.
5. It is more effort. But nothing surprising happens, and several people can write cards at the same time.

Where this comparison breaks: the kitchen is always about one full order behind. By the time your card is being cooked you are already writing the next one. If you ever ask “is my food ready yet” and wait for the answer, both of you stop. That is what a synchronization stall costs, and it is the single most common performance mistake in graphics programming.

■ 22.12.3 PLAIN – a worked example

1. A frame is CPU-bound or GPU-bound. Here is how to tell, with numbers.
2. Measure two things: how long the processor takes to prepare the frame, and how long the chip takes to draw it.

case A	CPU 12.0 ms	=====	
	GPU 6.0 ms	=====	-> CPU-bound, 83 fps

case B	CPU 4.0 ms	====	
	GPU 15.0 ms	=====	-> GPU-bound, 67 fps

3. In case A, buying a faster graphics card changes nothing. The chip already finishes early and waits.
4. In case A, lowering the resolution also changes nothing, because resolution costs the chip, not the processor.
5. What helps in case A: fewer draw calls, fewer objects, better batching, a lower-overhead API.
6. In case B, resolution and shader quality are exactly the right dials.
7. A quick field test: drop the resolution by half. If the frame rate barely moves, you are CPU-bound.

■ 22.12.4 PLAIN – what is really happening inside

1. The API validates your calls, tracks state, and hands work to the driver.
2. The driver contains a shader compiler that turns the portable intermediate form into the specific chip's instructions, and a command translator that builds the hardware command packets.
3. Those packets go into a ring buffer in memory. The chip reads the ring buffer independently through direct memory access.
4. Work is submitted to **queues**. A modern chip has separate queues for graphics, for compute, and for copying, and they can run at the same time.
5. Because they run at the same time, you need explicit fences and barriers to say "this must finish before that starts". Getting those wrong causes corruption that only appears on some hardware.
6. Shader compilation is the classic stutter cause. A game may contain tens of thousands of shader variants. Compiling one takes a few milliseconds to a few hundred. Doing it at the moment an effect first appears drops a frame.
7. The fix is to compile everything ahead of time and cache it, which is why many games now show a "compiling shaders" screen on first run or after a driver update.
8. A driver update invalidates the cache, because the generated machine code may change. That is why the stutter returns after updating a driver.

■ 22.12.5 TECHNICAL – the engineer's version

1. API history, with dates. OpenGL 1.0 from Silicon Graphics in 1992. Direct3D in 1996. OpenGL 2.0 with GLSL in 2004. Direct3D 11 in 2009. AMD Mantle in 2013, which was donated and became the basis of Vulkan. Apple Metal in June 2014. Direct3D 12 with Windows 10 in July 2015. Vulkan 1.0 on 16 February 2016. Vulkan 1.4 in December 2024. WebGPU shipped in Chrome 113 in May 2023 and its W3C specification was still at Candidate Recommendation stage in January 2026.
2. What changed with the low-level APIs, precisely: pipeline state is baked into immutable objects instead of being set piecemeal; resource lifetime and residency become the application's job; memory is allocated and sub-allocated by the application; synchronization is explicit; and command buffers can be recorded from many threads at once.
3. The measured payoff is draw call overhead. A Direct3D 11 draw call costs roughly 1 to 10 microseconds of processor time depending on how much state changed with it; Direct3D 12 and Vulkan reduce this to well under 1 microsecond and let the cost be spread across threads.
4. The cost is code volume. A minimal triangle in OpenGL is about 100 lines. In Vulkan it is 800 to 1,500. This is a real and widely acknowledged trade, and it is why most people use an engine rather than the API directly.
5. Pipeline state objects are the mechanism behind first-run stutter. Direct3D 12 added Pipeline State Object libraries and Vulkan added pipeline caches; both Vulkan's

VK_EXT_graphics_pipeline_library (2022) and Direct3D 12's state object work aim to reduce combinatorial explosion. Valve's Steam Deck distributes precompiled pipeline caches to users precisely for this reason.

6. Queue families in Vulkan are advertised by the device. A typical discrete card exposes one graphics-and-compute queue family, one or more compute-only, and one transfer-only queue using the copy engines.
7. Diagnosis tools: RenderDoc for frame capture and per-draw timing; PIX on Windows; Nsight Systems and Radeon GPU Profiler for timeline views showing exactly where the processor and the chip wait for each other; VK_LAYER_KHRONOS_validation for correctness.

■ 22.12.6 WORDS – remember these

1. Graphics API — the agreed set of drawing functions — the specified interface between application and driver, such as Vulkan or Direct3D 12.
2. Driver — the translator to real hardware — vendor software containing the shader compiler, memory manager and command packet builder.
3. Command buffer — a written list of orders — a recorded sequence of GPU commands, submitted to a queue for later execution.
4. Queue — a lane of work into the chip — an independent submission channel; graphics, compute and transfer queues can run concurrently.
5. Pipeline state object — the whole draw configuration frozen — an immutable object bundling shaders, blend, depth and raster state, compiled once.
6. CPU-bound — the processor is the limit — frame time dominated by host-side preparation, unaffected by resolution.
7. GPU-bound — the chip is the limit — frame time dominated by device execution, responsive to resolution and shader cost.

22.13 Graphics chips for everything else

■ 22.13.1 PLAIN – in simple words

1. For thirty years these chips did one thing. Then people noticed the shape of the work, not the subject of the work, was what suited them.
2. Anything made of millions of independent, identical, arithmetic-heavy steps fits. Weather models, molecular simulation, oil exploration, cryptography and, above all, neural networks.
3. Before 2007 you had to disguise your problem as a drawing problem. People really did pack scientific data into textures and read the answer out of a picture.
4. In 2007 NVIDIA released CUDA, which let you write ordinary C-like code for the chip with no pretence of drawing. That is the moment the field opened.
5. Speed is quoted in FLOPS, floating-point operations per second. A TFLOP is a trillion of them per second.
6. But the number depends entirely on how precise the numbers are. Halve the precision and the same silicon roughly doubles the rate.
7. And the FLOPS number is usually not what limits you. Memory bandwidth is. Most real work spends its life waiting for data.
8. The chips sold for datacentres and the chips sold for games are the same family with very different memory, very different precision support, and very different prices.

■ 22.13.2 PLAIN – a picture in your head

1. Think of a factory that can assemble 100,000 items per hour.
2. The loading bay can only bring in enough parts for 20,000 items per hour.
3. The factory's rated capacity is a fiction. You will build 20,000 items per hour, and buying faster assembly robots will change nothing.
4. The loading bay is memory bandwidth. The robots are FLOPS.
5. The only real fixes are a bigger loading bay, or redesigning the product so each part gets used in more items before it is thrown away.

Where this comparison breaks: you can genuinely redesign the work. Reusing data that is already on the chip, by tiling a matrix multiply so each loaded block feeds many operations, is the single most important optimization in the field, and it is why a well-written matrix multiply reaches near peak while a naive one reaches a few percent.

■ 22.13.3 PLAIN – a worked example

1. Real throughput of one NVIDIA H100 SXM, at each precision, dense (that is, not counting the sparsity doubling that vendors like to quote):

Precision	Dense throughput
FP64 tensor	67 TFLOPS
FP32 vector	67 TFLOPS
TF32 tensor	494 TFLOPS
FP16 / BF16 tensor	989 TFLOPS
FP8 tensor	1,979 TFLOPS

2. From FP64 to FP8 is a factor of about 30 on the same chip. Nothing changed except how many bits each number uses.
3. This is why the AI field moved to lower precision so aggressively. It is not a small saving.
4. Now the bandwidth check. The H100 SXM has 3,350 GB/s. At FP16 it can do 989 trillion operations per second.
5. That is $989 \times 10^{12} / 3350 \times 10^9 =$ about 295 operations for every byte read. Any calculation with a lower ratio than that is bandwidth-limited.
6. Generating one token from a 70-billion-parameter model in 16-bit precision requires reading roughly 140 gigabytes of weights, and does about two operations per weight. That is 1 operation per byte. Hopelessly bandwidth-limited.
7. At 3,350 GB/s, reading 140 GB takes about 42 milliseconds. That is the physical floor for one token on one such chip, no matter how fast its arithmetic is.
8. A gaming card against a datacentre accelerator, same generation family:

Item	RTX 5090	B200 SXM
Memory	32 GB GDDR7	180 GB HBM3e
Bandwidth	1.79 TB/s	about 8 TB/s
FP4 tensor, sparse	about 3.4 PFLOPS	18 PFLOPS
Board power	575 W	up to 1,000 W
Launch price	1,999 US dollars	not sold singly

■ 22.13.4 PLAIN – what is really happening inside

1. A general-purpose program for a graphics chip is called a kernel. You launch a grid of thread blocks; each block runs on one streaming multiprocessor.
2. The programming model is deliberately the same as a compute shader. Same hardware, different front door.
3. Data must be copied to the card's memory first, unless the system has a shared memory design. That copy crosses PCIe and is often the slowest part.
4. Matrix multiply is the workhorse. The efficient version splits the matrices into tiles that fit in shared memory, loads each tile once, and reuses it across many multiply-accumulate operations.
5. Tensor cores make this even more extreme: one instruction consumes a whole tile and produces a whole tile of results.
6. Lower precision helps twice: the arithmetic units go faster, and the same bytes of bandwidth carry twice as many numbers.
7. That second effect is often the larger one, which is exactly the point about bandwidth being the true limit.

■ 22.13.5 TECHNICAL – the engineer's version

1. Timeline. NVIDIA announced CUDA with the G80 architecture in November 2006; the public toolkit shipped in 2007. Khronos released OpenCL 1.0 in December 2008, initiated by Apple. Both are still maintained; CUDA has by far the larger ecosystem, which is a business fact as much as a technical one.
2. The scientific computing shift came first. GPU-accelerated entries appeared in the TOP500 supercomputer list from 2008 onward, and Tianhe-1A took the number one spot in November 2010 using NVIDIA Tesla parts.
3. The machine learning shift is usually dated to AlexNet, Krizhevsky, Sutskever and Hinton, which won the ImageNet competition in 2012 trained on two GeForce GTX 580 cards with 3 GB each. That result is established fact and is the hinge of the modern field.
4. Precision formats in use, all standards or de facto standards: IEEE 754 binary 64 and binary 32; IEEE 754 binary16; bfloat16, from Google Brain, which keeps FP32's exponent range with 8 fewer mantissa bits; TF32, NVIDIA's 19-bit internal tensor format from 2020; FP8 in the E4M3 and E5M2 variants, standardized by an Arm, Intel and NVIDIA joint proposal in 2022; and FP4 from Blackwell in 2024.
5. Marketing claim, flagged as such: every headline TOPS figure from every vendor in this space is quoted at the lowest supported precision, with structured sparsity assumed. NVIDIA's 3,352 AI TOPS for the RTX 5090 is FP4 with sparsity. Dividing by four gives a fairer comparison against an older FP16 figure.
6. Active research, flagged as such: whether FP4 and lower training is generally viable rather than viable for specific layers is not settled as of 2026. Inference at 4 bits with careful quantization is established practice; 4-bit training is not.
7. The roofline model, from Williams, Waterman and Patterson in 2009, is the standard tool: plot achievable performance against arithmetic intensity, with a slanted bandwidth roof and a flat compute roof. Nsight Compute draws it for you per kernel.
8. Chapter 46 takes this forward into what a neural network actually is and why these numbers govern what can be trained and served.

■ 22.13.6 WORDS – remember these

1. FLOPS — sums per second — floating-point operations per second; TFLOPS is 10^{12} and PFLOPS is 10^{15} .
2. Kernel — a program launched over a grid — a device function executed by many threads organized into blocks.
3. Arithmetic intensity — sums per byte fetched — the ratio of operations to bytes moved, which determines the binding limit under the roofline model.
4. Sparsity — assume half the weights are zero — NVIDIA's 2:4 structured sparsity, which doubles quoted tensor throughput when applicable.
5. bfloat16 — 16 bits with FP32's range — a truncated FP32 with 8 exponent bits and 7 mantissa bits, from Google Brain.
6. CUDA — the first non-graphics door into the chip — NVIDIA's proprietary parallel computing platform and C-like language, public from 2007.

22.14 Integrated, discrete and mobile chips

■ 22.14.1 PLAIN – in simple words

1. A **discrete** graphics card is a separate board with its own memory chips and its own power supply, plugged into the processor over a link.
2. An **integrated** graphics chip is built into the same package as the processor and has no memory of its own. It borrows the system's memory.
3. That borrowing is the whole story of integrated graphics. System memory is several times slower than a graphics card's memory.
4. Apple's approach is different: one large pool of fast memory that both the processor and the graphics chip address directly, with no copying.
5. Phone and tablet chips use a further trick. Instead of drawing the whole screen at once, they cut it into small tiles and finish each tile completely in tiny on-chip memory.
6. That saves enormous amounts of memory traffic, which saves power, which is why a phone can render a detailed scene on a battery.
7. A phone chip can do almost everything a desktop chip can do, including ray tracing on recent models. What it cannot do is sustain it, because of heat.

■ 22.14.2 PLAIN – a picture in your head

1. Imagine painting a large mural.
2. A desktop chip paints the whole wall at once, walking back to the paint store for every colour.
3. A phone chip masks off one square metre, keeps that square's paints on a small tray at hand, finishes it completely, then moves the tray to the next square.
4. Far fewer trips to the store. The store is memory, and every trip costs battery.

Where this comparison breaks: the tile has to know which parts of the mural touch it before it starts, so all the shapes must be sorted into tiles first. That sorting step is real work and real memory traffic, and it is why tile-based designs do best on scenes with moderate geometry and lose their advantage as triangle counts explode.

■ 22.14.3 PLAIN – a worked example

1. Memory bandwidth, the number that decides everything here:

System	Memory	Bandwidth
Desktop, DDR5-6000	System DDR5	96 GB/s
Apple M5	Unified LPDDR5X	153.6 GB/s
Apple M5 Max, 40-core	Unified LPDDR5X	614 GB/s
Intel Arc B580	12 GB GDDR6	456 GB/s
RTX 5090	32 GB GDDR7	1,792 GB/s

2. An integrated chip on a normal desktop shares that 96 GB/s with the processor, which is also using it. In practice it sees perhaps 60 to 70 GB/s.
3. That is about 4 percent of an RTX 5090's bandwidth. No amount of arithmetic capability rescues that.
4. Apple's M5 Max at 614 GB/s is in the same range as a mid-range discrete card, and unlike a discrete card it can address up to 128 GB of it.
5. That combination is why Apple laptops became popular for running large models locally: not raw speed, but a large pool of reasonably fast memory with no copying.

■ 22.14.4 PLAIN - what is really happening inside

1. Integrated graphics carve a region out of system memory. Some is reserved at boot, more is allocated on demand. The processor and graphics chip may share a last-level cache, which helps.
2. Unified memory on Apple silicon means one physical pool and one address space. A buffer written by the processor is visible to the graphics chip with no copy and no PCIe transfer.
3. Tile-based deferred rendering, the mobile approach, works in two passes. The first pass runs all the vertex work and records which triangles fall in which tile, writing that list to memory.
4. The second pass takes one tile at a time, typically 16x16 or 32x32 pixels, loads nothing, rasterises every triangle in that tile's list into on-chip memory, and writes only the final colours out.
5. Because the depth buffer for a tile lives on-chip, depth traffic to main memory is almost eliminated. So is blending traffic.
6. "Deferred" here refers to deferring fragment shading until visibility within the tile is resolved, so hidden surfaces are never shaded at all.
7. What a phone cannot do is sustain power. A phone chip may draw 10 watts for a few seconds and then must fall back to 3 to 5 watts or the case becomes too hot to hold. Frame rates fall accordingly.

■ 22.14.5 TECHNICAL - the engineer's version

1. Tile-based rendering came from Imagination Technologies' PowerVR, whose first parts shipped in 1996 and which powered the Sega Dreamcast in 1998 and every iPhone until Apple's own designs from the A11 in 2017.
2. Arm Mali and Qualcomm Adreno both use tiling; Adreno can also switch to immediate mode per render pass, which Qualcomm calls FlexRender.
3. Vulkan and Metal expose tiling directly through render pass load and store operations. Declaring `LOAD_OP_DONT_CARE` and `STORE_OP_DONT_CARE` correctly is the single biggest mobile optimization, because it tells the driver a tile never needs to be read from or written to main memory.
4. Apple silicon current figures, from Apple's own specifications: M5, launched 15 October 2025, has an 8 or 10 core GPU with a neural accelerator in each core and 153.6 GB/s of unified bandwidth. M5 Pro and M5 Max followed on 3 March 2026, with up to 20 and up to 40 GPU cores, and 307 GB/s and 614 GB/s respectively.
5. AMD's integrated designs in handheld consoles, and Apple's designs, blur the old boundary: both are integrated in the strict sense yet perform like entry-level to mid-range discrete parts.

6. What a phone GPU genuinely cannot do, as of 2026: hold large working sets, because bandwidth and thermal budget both punish it; sustain high clocks for more than a few minutes; and match the fixed-function ray tracing throughput of a 300-watt desktop part, though recent Arm, Qualcomm, Imagination and Apple designs all have hardware ray tracing.
7. Tools: Arm Performance Studio and Qualcomm Snapdragon Profiler report tile counts, overdraw and bandwidth per render pass; Xcode's Metal debugger shows per-tile memory traffic on Apple silicon.

■ 22.14.6 WORDS – remember these

1. Discrete GPU — a separate card with its own memory — a device with dedicated VRAM connected over PCIe.
2. Integrated GPU — built into the processor package — a device sharing system DRAM and its bandwidth with the host.
3. Unified memory — one pool, no copying — a single physical memory and address space shared coherently by CPU and GPU, as on Apple silicon.
4. Tile-based deferred rendering — finish one small square at a time — binning primitives into screen tiles and shading each tile entirely in on-chip memory.
5. Binning pass — sorting shapes into squares — the first pass of a tiled renderer, producing per-tile primitive lists.
6. Thermal throttling — slowing down to stay cool — clock reduction driven by sustained power and temperature limits, dominant on mobile devices.

22.98 Common wrong ideas

1. Wrong: a GPU is just a faster CPU. Right: it is a slower processor repeated thousands of times. Single-thread performance on a GPU lane is several times worse than on a modern CPU core; the win is entirely in parallel count.
2. Wrong: 21,760 CUDA cores means 21,760 independent processors. Right: it means 21,760 arithmetic lanes across 170 independently scheduled blocks, with groups of 32 lanes forced to execute the same instruction.
3. Wrong: more TFLOPS means a faster card. Right: TFLOPS is a formula, not a measurement, vendors count it differently, and most real workloads are limited by memory bandwidth long before they run out of arithmetic.
4. Wrong: the fourth number in a 4x4 matrix is a mathematical trick with no meaning. Right: it is a real projective coordinate, and the division by it is literally what produces perspective.
5. Wrong: a normal map adds geometric detail. Right: it only changes the direction used in lighting. Silhouettes stay perfectly flat, and the illusion collapses at grazing angles.
6. Wrong: ray tracing replaced rasterisation. Right: every shipping game in 2026 rasterises primary visibility and uses rays for selected effects. Full path tracing runs at reduced internal resolution and depends on denoising.
7. Wrong: frame generation makes a game more responsive. Right: it raises the frame counter and smoothness while increasing click-to-photon latency by roughly 8 to 15 milliseconds, because a real frame must be held back.
8. Wrong: upscaling from a lower resolution always looks worse than native. Right: because native rendering has no anti-aliasing history, a good temporal upscaler often resolves fine detail and edges better than native with simple anti-aliasing, while being worse in fast motion.
9. Wrong: the depth buffer stores distance in metres. Right: it stores a non-linear function of distance, which is why precision is concentrated near the camera and why reversed-Z with a float buffer is

a large improvement.

10. Wrong: shader stutter means the game is badly optimized in general. Right: it usually means shader machine code is being generated on first use, which precompilation and a persistent cache fix without changing anything else.

22.99 Chapter summary in 20 lines

1. A screen is millions of independent small calculations per frame, which is a different shape of problem from a single fast instruction stream.
2. A processor is latency-optimised: a few clever cores with caches, branch prediction and reordering, built to finish one task quickly.
3. A graphics chip is throughput-optimised: thousands of simple lanes that hide memory waiting by switching to another ready group of threads.
4. Threads run in fixed groups, warps of 32 on NVIDIA and wavefronts of 32 or 64 on AMD, sharing one instruction at a time; divergence costs real time.
5. All drawing ends in a framebuffer. Lines come from integer stepping algorithms such as Bresenham's from 1965, and transparency from the Porter-Duff over operator of 1984.
6. Three-dimensional shapes are meshes of triangles with per-vertex normals, because three points are always flat, always convex, and always interpolatable.
7. A vertex travels through model, world, view, clip, normalized device and screen space, one matrix multiply per step plus one division by w.
8. Four-by-four matrices with homogeneous coordinates, an idea from Möbius in 1827, are used because a 3x3 matrix cannot express translation or perspective.
9. That same matrix multiply, on larger matrices, is exactly what a neural network layer computes, which is why one chip serves both fields.
10. The rasterisation pipeline has programmable stages, vertex, tessellation, geometry and fragment, separated by fixed-function clipping, divide, viewport transform, triangle setup, rasterisation, depth test and blending.
11. Shaders are compiled twice: once to a portable form such as SPIR-V, once by the driver into the installed chip's own machine code.
12. Textures supply detail, and mipmaps exist to prevent aliasing, not to save memory; block compression is what makes real texture budgets possible.
13. The depth buffer, from Catmull in 1974, makes draw order irrelevant, and its finite precision is the sole cause of z-fighting.
14. Lighting moved from Phong in 1975 and Blinn in 1977 to physically based models parameterized by albedo, metalness and roughness, following Disney's 2012 formulation.
15. Ray tracing answers the opposite question from rasterisation, needs a bounding volume hierarchy to be tractable, and became real time only with dedicated hardware from 2018.
16. Path tracing at real-time rates produces a noisy image that a neural denoiser reconstructs; the denoiser is load-bearing, not cosmetic.
17. Upscaling and frame generation trade image quality and latency for pixel throughput; the trades are real, measurable and worth stating honestly.
18. Inside a chip: independent blocks with their own schedulers, registers, shared memory and texture units, sharing an L2 cache and a wide memory controller feeding GDDR7 or HBM.
19. The ratio of peak arithmetic to peak bandwidth, about 58 operations per byte on an RTX 5090, decides whether any given program is compute or memory bound.
20. Integrated chips are limited by shared system bandwidth, Apple's unified memory sidesteps copying entirely, and mobile chips save power by finishing the picture one small on-chip tile at a

time.



PART F

Networks

Signals, addresses, names, routes and encryption. Worked
throughout on one real network failure.

Networking From Zero — Signals, Ethernet, MAC, ARP and DHCP

23.0 What this chapter gives you

1. You will be able to say what a network actually is, and name the four jobs every network must do no matter how it is built.
2. You will be able to explain how a bit becomes a voltage on copper, and why a plain high-low signal is not good enough.
3. You will be able to name the line codes that fix it, from Manchester in 1973 to PAM-4 today, and say what each one buys.
4. You will be able to read a cable category number and state its real bandwidth and distance limit instead of guessing.
5. You will be able to tell bandwidth, throughput, latency and jitter apart, and compute a bandwidth-delay product on paper.
6. You will be able to draw an Ethernet frame field by field with correct byte lengths, and explain where the number 1500 came from.
7. You will be able to read a MAC address, find its vendor, and say why your phone lies about it on purpose.
8. You will be able to describe how a switch learns, why a loop is a disaster, and what spanning tree does about it.
9. You will be able to walk through ARP and DHCP message by message, using your own router at 192.168.0.1 as the example.
10. You will be able to explain everything that happens on the wire before a single IP packet is sent, which is what the next six chapters build on.

This chapter opens Part F. Part F uses one real event as its running example: a real network outage on a real home broadband line in India, on a macOS laptop, where `curl -v https://github.com` printed `Trying 20.207.73.82:443...` and then sat in complete silence for fifteen seconds. We will take that event apart layer by layer across the whole of Part F. This chapter covers everything underneath the IP layer: the signal, the cable, the frame, the address, and the two small protocols that get a machine onto a network at all. IP addressing, NAT, DNS, routing, traceroute, TCP and TLS each get a full chapter of their own later in Part F. Where they come up here, we name them and move on.

23.1 What a network is: two machines and a shared medium

■ 23.1.1 PLAIN – in simple words

1. A network is two or more machines that can send each other messages.
2. To send a message you need something in between. A wire, a fibre, or air.
3. That in-between thing is called the **medium**. It is the road the message travels on.
4. If only two machines share the medium, life is easy. Anything you send can only arrive at one place.
5. As soon as three or more machines share it, four new problems appear at once.
6. Problem one is **addressing**. If the message can reach everyone, it must say who it is for.
7. Problem two is **framing**. The medium carries a stream of signals. Something must mark where one message stops and the next starts.
8. Problem three is **error detection**. Wires pick up noise. The receiver must be able to tell that what arrived is not what was sent.
9. Problem four is **sharing**. If two machines talk at the same time on one shared medium, both messages are ruined. Somebody has to take turns.
10. Every network technology ever built is an answer to those same four problems. The answers differ. The problems do not.

■ 23.1.2 PLAIN – a picture in your head

1. Picture a long corridor in a hostel with many rooms opening onto it.
2. If you shout from your doorway, everyone in the corridor hears you. The corridor is the shared medium.
3. So you begin with a name: “Anita, the parcel is at the desk.” That is addressing. Everyone hears it. Only Anita acts on it.
4. You say the whole sentence in one go, then stop. The pause tells listeners the message has ended. That is framing.
5. If a bike goes past outside and drowns out three words, Anita asks you to repeat. That is error detection.
6. If two people shout at once, nobody can make out either. So people listen first, and wait if they hear a voice. That is sharing the medium.
7. Now picture a different arrangement: a private phone line from your room to Anita’s room only. No names needed. No shouting. No waiting.
8. That is the difference between a **broadcast** medium and a **point-to-point** link.

Where this comparison breaks: a corridor is slow and forgiving. Humans hear a clash and naturally back off. On a wire, two signals overlapping do not sound wrong, they simply produce a third voltage that looks like valid data. The machine cannot notice by ear. It has to be told to check, with a checksum, or to detect the clash electrically within a fixed number of microseconds. Also, a corridor has no fixed speed. A wire does, and that fixed speed is what makes all the timing rules later in this chapter possible.

■ 23.1.3 PLAIN – a worked example

1. Take a home with a laptop, a phone, a smart TV and a router.
2. The router is at 192.168.0.1. That is the real address of the reader’s router, and we will use it throughout this chapter.
3. All four devices are on one logical shared network, even though the laptop and phone reach it over radio and the TV over a cable.

4. The laptop wants to send something to the router. It cannot just push the bytes out and hope.
5. It must attach the router's local hardware address, so the network knows the message is for the router and not the TV. That is addressing.
6. It must wrap the bytes in a start marker and an end marker so the router knows where the message begins and ends. That is framing.
7. It must append a check number computed from the bytes, so the router can confirm nothing was corrupted. That is error detection.
8. Over radio, it must wait until the air is quiet before transmitting, because the phone might be talking. That is sharing.
9. Over the cable to the TV, none of the waiting is needed, because a modern switched cable link is private in both directions.
10. Same four problems. Two different sets of answers, chosen by the medium.

■ 23.1.4 PLAIN – what is really happening inside

1. Networks are built in layers, and each layer solves one of those problems and hands the rest to the layer above.
2. The bottom layer is the **physical layer**. Its only job is to turn a 1 or a 0 into something the medium can carry, and back again.
3. Above it sits the **data link layer**. This is where framing, local addressing, error detection and medium sharing live.
4. Above that sits the network layer, which is where IP lives, and which we deliberately leave for the next chapter.
5. A **topology** is the shape of the connections. There are four classic shapes.
6. A **bus** is one long shared cable with every machine tapped onto it. Cheap, and one cut kills everything. This was early Ethernet.
7. A **star** has every machine wired to one central box. One cut kills one machine. This is what every home and office uses now.
8. A **ring** passes messages around a loop, each machine to the next. Token Ring and FDDI worked this way. Fibre backbones still use rings for resilience.
9. A **mesh** connects many machines to many others, so there are several paths between any two. The internet core is a mesh. So is a modern WiFi mesh kit.
10. In practice a home network is a star of stars: devices star into the router, and the router is one point in the ISP's tree.

■ 23.1.5 TECHNICAL – the engineer's version

1. The reference model in textbooks is the OSI seven-layer model, published as ISO/IEC 7498-1, with the first version issued in 1984.
2. Real systems follow the four-layer TCP/IP model described in RFC 1122 (1989): link, internet, transport, application.
3. This chapter is entirely layer 1 and layer 2 in OSI terms, and entirely the link layer in TCP/IP terms.
4. The four link-layer duties have formal names: physical signalling, delineation, frame check sequence, and medium access control.
5. **Medium access control** (MAC) is the sublayer that decides who transmits when. Its name is also why hardware addresses are called MAC addresses.
6. Access methods in use: CSMA/CD (classic Ethernet), CSMA/CA (802.11 WiFi), token passing (802.5 Token Ring, FDDI), TDMA and OFDMA (cellular, WiFi 6).
7. The topology table below gives the practical trade-off.

Topology	Failure of one link	Where used now
Bus	Whole segment dies	Obsolete, CAN in cars
Star	One node dies	All LANs, home WiFi
Ring	Ring wraps, survives	SONET, metro fibre
Mesh	Reroute around it	Internet core, WiFi mesh

8. A **collision domain** is the set of ports where two simultaneous transmissions would interfere. A **broadcast domain** is the set of ports a broadcast frame reaches. Section 23.8 pulls these apart properly.
9. The distinction between a standard and a convention matters here. IEEE 802.3 is a standard. Calling the central box a “switch” rather than a “bridge” is a convention; the standard still says bridge.

■ 23.1.6 WORDS – remember these

1. Medium — the stuff a message travels through — the physical channel: copper, fibre or free space.
2. Point-to-point — a private line between two machines — a link with exactly two endpoints and no contention.
3. Broadcast medium — a shared space everyone hears — a multiple-access channel requiring a MAC protocol.
4. Framing — marking where a message starts and stops — delineation of a bit stream into protocol data units.
5. Topology — the shape of the wiring — the graph of nodes and links, physical or logical.
6. Collision — two machines talking at once — overlapping transmissions on a shared medium producing an undecodable signal.
7. Layer — one job in the stack — an abstraction level with a defined service interface to the layer above.

23.2 Bits on a wire, physically

■ 23.2.1 PLAIN – in simple words

1. A wire cannot carry a number. It can only carry a voltage, which is an electrical push.
2. So we agree on a code. High voltage means 1. Low voltage means 0. That is the simplest possible scheme.
3. This simple scheme has a name: **NRZ**, short for non-return-to-zero. The voltage just stays where it is for the whole bit.
4. NRZ has a serious problem, and the problem is time.
5. The receiver has to know exactly when each bit starts and ends. Otherwise it cannot tell one long 1 from four short 1s.
6. It could use its own clock, but no two clocks in the world run at exactly the same speed. They drift apart.
7. So the receiver needs the signal itself to tell it the time. It does this by watching for the moments the voltage changes.
8. Every change is a tick it can use to correct its clock. This is called **clock recovery**.
9. Send a hundred zeros in a row and there are no changes for a hundred bit times. The receiver’s clock drifts, and it loses count. That is the failure.
10. There is a second problem. A long run of the same voltage builds up a steady electrical offset in the transformers and capacitors along the path.

11. That offset is called **DC bias**, and it slowly ruins the receiver's idea of what counts as high and what counts as low.
12. So we need a code where the voltage keeps changing no matter what the data says, and where highs and lows come out roughly equal over time. That balance is called **DC balance**.

■ 23.2.2 PLAIN – a picture in your head

1. Think of a drummer keeping time for a dancer who cannot see the drummer.
2. The dancer knows the tempo roughly, but not perfectly. Over a long silence, the dancer slips.
3. If the drummer hits the drum on every single beat, the dancer never slips, because every beat is confirmed.
4. Plain NRZ is a drummer who only hits the drum when the music changes. During a long steady passage there is nothing to hear.
5. Manchester coding is a drummer who hits the drum in the middle of every single beat, always, and uses the direction of the hit to carry the message.
6. It costs twice the effort, but the dancer never loses the beat.
7. The clever later codes are drummers who found a way to guarantee a hit at least every few beats, without hitting on every beat.

Where this comparison breaks: a dancer who slips can recover by watching other dancers. A receiver has nothing else to watch. Also, a drum hit is either there or not, while a real signal on a long cable arrives smeared, rounded and mixed with echoes of earlier bits. The receiver is not detecting a clean edge, it is running an equalizer that reconstructs a plausible edge from a blurred one. At 10 Gbit/s over 100 m of copper, the raw received waveform looks nothing like the transmitted one until that equalizer has done its work.

■ 23.2.3 PLAIN – a worked example

1. We will encode one byte, the ASCII letter K, which is hex 0x4B and binary 01001011.
2. We use the IEEE 802.3 Manchester convention: a 0 is high then low, a 1 is low then high. Each bit becomes two half-bits.
3. Below, - is high voltage and _ is low. Each bit gets four characters, so two per half-bit.

bit	0	1	0	0	1	0	1	1
NRZ	_____	_____	_____	_____	_____	_____	_____	_____
MAN	--____	--____	--____	--____	--____	--____	--____	--____

4. Look at the NRZ line. Bits 3 and 4 are both 0, so eight characters pass with no change at all. Bits 7 and 8 are both 1, same problem.
5. Now look at the Manchester line. There is a transition in the middle of every single bit, guaranteed, whatever the data is.
6. Count the highs and lows in the Manchester line. There are sixteen of each. Perfectly balanced. That is DC balance, guaranteed by construction.
7. The price is visible too. Manchester needs 32 half-bits to send 8 bits. It uses twice the signal changes per bit, so twice the bandwidth.
8. That is exactly why 10BASE-T, which used Manchester, needed 20 MHz of cable bandwidth to carry 10 Mbit/s.

■ 23.2.4 PLAIN – what is really happening inside

1. A twisted pair is two copper wires twisted around each other along the whole length of the cable.
2. The transmitter does not put a voltage on one wire and use earth as the other. It drives the two wires in opposite directions.
3. If one goes up by one volt, the other goes down by one volt. The receiver only looks at the difference between them. This is **differential signalling**.
4. Now suppose a motor nearby throws out electrical noise. That noise hits both wires almost equally, pushing both up by the same amount.
5. The difference between them is unchanged. The noise cancels. Noise that affects both wires equally is called **common mode**, and differential receivers ignore it.
6. The twisting is what makes the noise hit both wires equally. Over each half twist, the wire nearer the noise source swaps places with the other one.
7. Averaged over the length, both wires have been equally near the noise. So the induced noise on the two is equal, and cancels.
8. Twisting also stops the pairs inside one cable from disturbing each other. That interference has a name, **crosstalk**, and each pair is twisted at a different rate so their fields do not line up.
9. Higher cable categories are, in large part, tighter and more precisely varied twist rates, better copper, and sometimes metal foil around each pair.
10. Foil or braid around the pairs is called **shielding**. It blocks outside interference, but only works if the shield is properly grounded at the ends.

■ 23.2.5 TECHNICAL – the engineer's version

1. **NRZ** maps one bit to one symbol with two voltage levels. Zero coding overhead, no guaranteed transitions, no DC balance.
2. **Manchester** (IEEE 802.3 convention: 0 is high-to-low at mid-bit, 1 is low-to-high) is 50 percent efficient. It doubles the baud rate. Used by 10BASE-T, and still used by consumer infrared, RFID and NFC.
3. There are two Manchester conventions. G. E. Thomas defined the opposite mapping in 1949; inverting a Manchester signal converts one convention into the other. Always state which one you mean.
4. **4B/5B** maps each 4 data bits to a 5-bit symbol chosen so no symbol has more than three consecutive zeros. 80 percent efficient. Used by FDDI and by 100BASE-TX, where it feeds an MLT-3 line driver.
5. **MLT-3** is a three-level code that cycles low, mid, high, mid on each 1 and holds on each 0. It cuts the fundamental frequency of 100BASE-TX from 125 MHz to about 31.25 MHz, which is why Cat5 could carry Fast Ethernet.
6. **8b/10b** was described by Al Widmer and Peter Franaszek at IBM in 1983, US patent 4,486,739 granted December 1984. It maps 8 bits to 10, 80 percent efficient, guarantees run length and DC balance through a running disparity of at most plus or minus two, and supplies comma symbols for alignment. Used by 1000BASE-X, PCI Express 1.x and 2.x, SATA, USB 3.0, DisplayPort 1.x and Fibre Channel.
7. **64b/66b** arrived with 10 Gigabit Ethernet. It adds a 2-bit sync header to 64 bits of scrambled payload. Overhead is about 3.1 percent instead of 20 percent. It does not guarantee DC balance or run length; it makes them statistically overwhelmingly likely via a self-synchronous scrambler.
8. **PAM-4** sends two bits per symbol using four voltage levels instead of two. It halves the symbol rate for a given bit rate, at the cost of about 9.5 dB of signal-to-noise margin, so it is always paired with forward error correction. It is the basis of 50G, 100G, 200G and 400G electrical lanes.
9. **PAM-16**, sixteen levels, with 64B/65B framing and LDPC coding, is what 10GBASE-T uses over

copper.

Code	Bits per symbol	Overhead	Example use
NRZ	1	0 percent	Short backplanes
Manchester	0.5	100 percent	10BASE-T
4B/5B + MLT-3	0.8	25 percent	100BASE-TX
8b/10b	0.8	25 percent	1000BASE-X
64b/66b	0.97	3.1 percent	10GBASE-R
PAM-4	2	needs FEC	400G lanes

10. Twisted-pair cabling categories are defined by ANSI/TIA-568 in North America and ISO/IEC 11801 internationally. The numbers below are the tested frequency, not the data rate.

Category	Bandwidth	Ethernet at 100 m
Cat 3	16 MHz	10BASE-T
Cat 5	100 MHz	100BASE-TX
Cat 5e	100 MHz	1000BASE-T
Cat 6	250 MHz	5GBASE-T
Cat 6A	500 MHz	10GBASE-T
Cat 7	600 MHz	10GBASE-T
Cat 7A	1000 MHz	10GBASE-T
Cat 8	2000 MHz	40GBASE-T at 30 m

- Cat 5 was deprecated in favour of Cat 5e in 2001. Cat 7 and Cat 7A are ISO/IEC classes and were never adopted by TIA, and they use non-RJ45 connectors such as GG45 and TERA; this is why they are rare in practice.
- Cat 8 comes in Class I (Cat 8.1, U/FTP or F/UTP, RJ45 compatible) and Class II (Cat 8.2, F/FTP or S/FTP, GG45 or TERA). It supports 25GBASE-T and 40GBASE-T, standardized in IEEE 802.3bq in 2016, over a maximum channel of 30 m. It is a data centre top-of-rack cable, not a building cable.
- Shielding notation is X/YTP where X is the overall shield and Y is the per pair shield: UTP, F/UTP (foil overall), S/FTP (braid overall, foil per pair), U/FTP (foil per pair only).
- Every twisted-pair Ethernet variant from 10BASE-T through 10GBASE-T is specified to 100 m of channel, which is 90 m of solid horizontal cable plus 10 m of stranded patch cords.
- The honest version: category ratings are measured on a fully certified channel with proper terminations. A Cat 6A cable punched down badly, or run next to a fluorescent ballast, may fail at 30 m. Cable certification testers such as the Fluke DSX series measure insertion loss, NEXT, return loss and delay skew, and they exist precisely because the label on the jacket is a claim, not a measurement.

■ 23.2.6 WORDS - remember these

- NRZ — voltage just stays high or low — non-return-to-zero line code, one bit per symbol, no guaranteed transitions.
- Clock recovery — working out the beat from the signal — extracting a bit clock from data transitions with a phase-locked loop.
- DC balance — equal amounts of high and low — zero mean of the transmitted waveform, required for transformer and capacitor coupling.

4. Manchester — a change in the middle of every bit — biphas-L code, 50 percent efficient, self-clocking and DC balanced.
5. Line code — the rule turning bits into signals — the mapping from data bits to channel symbols.
6. Differential signalling — drive two wires oppositely — balanced transmission where the receiver takes the difference and rejects common-mode noise.
7. Crosstalk — one pair leaking into another — near-end and far-end coupling between pairs, measured as NEXT and FEXT in dB.
8. Twisted pair — two wires wound together — balanced copper pair with a controlled twist rate for noise and crosstalk rejection.
9. PAM-4 — four voltage levels instead of two — pulse amplitude modulation carrying two bits per symbol, always used with FEC.

23.3 Bits on fibre and on air

■ 23.3.1 PLAIN – in simple words

1. Copper carries electricity. Fibre carries light. Air carries radio waves.
2. A fibre optic cable is a very thin strand of extremely pure glass. Thinner than a human hair.
3. A laser or an LED at one end flashes light into it. A light detector at the other end sees the flashes.
4. Light on means 1, light off means 0, in the simplest fibre systems.
5. The light does not leak out of the sides, even when the fibre is bent, and that is the whole trick.
6. It stays in because of **total internal reflection**. Light hitting the edge of the glass at a shallow enough angle bounces back in instead of escaping.
7. To make that happen, the fibre has two layers of glass. An inner **core** and an outer **cladding** with a slightly lower ability to bend light.
8. Fibre beats copper in three ways. It goes much further, it carries much more, and it does not care about electrical noise at all.
9. It has no metal in the signal path, so a lightning strike or a motor next door does nothing to it.
10. Radio is the third medium. It needs no cable at all, which is its whole appeal and also every one of its problems.

■ 23.3.2 PLAIN – a picture in your head

1. Imagine a long straight corridor with mirrors on both walls.
2. Shine a torch down it at a shallow angle. The beam bounces from wall to wall and travels the whole length, still bright.
3. Now imagine the corridor is not straight but gently curved. The beam still bounces along, because the angles are still shallow.
4. Bend it sharply round a corner and some light hits the wall too steeply, passes through, and is lost. That is a **bend loss** in a real fibre.
5. Now imagine two kinds of corridor. A wide one, where beams entering at different angles take noticeably different path lengths.
6. Beams that started together arrive at different times, so a sharp flash at one end becomes a smeared blob at the other. That is **multi-mode** fibre.
7. And a very narrow corridor, so narrow that only one straight path fits. Every photon takes the same route and arrives together. That is **single-mode**.
8. Narrow costs more to build and needs a better torch. But the flash stays sharp over tens of kilometres.

Where this comparison breaks: light in a fibre is not a torch beam bouncing off mirrors. The refractive

index change at the core boundary reflects the wave, and in single-mode fibre the core is only a few wavelengths wide, so ray optics stops being the right description entirely. There the light is a guided wave whose field extends slightly into the cladding. The corridor picture also has no equivalent of chromatic dispersion, where different colours of light travel at different speeds through the same glass and smear the pulse even in single-mode fibre.

■ 23.3.3 PLAIN – a worked example

1. Compare three ways of getting 10 Gbit/s from one building to another.
2. Option one: 10GBASE-T over Cat 6A copper. Maximum 100 m. Cheap cable, cheap sockets, but the transceiver chip runs hot and burns several watts per port.
3. Option two: 10GBASE-SR over multi-mode fibre at 850 nm. Maximum 300 m on OM3 fibre, 400 m on OM4. Cheap laser, more expensive fibre.
4. Option three: 10GBASE-LR over single-mode fibre at 1310 nm. Maximum 10 km. More expensive laser, cheapest fibre per metre.
5. The buildings are 600 m apart. Copper is out at 100 m. Multi-mode is out at 400 m. Single-mode wins, and would still win at 6 km.
6. Notice that the fibre itself is not what sets the distance. The **optics** at each end do: the laser wavelength, its power, and the receiver's sensitivity.
7. Those optics live in a small plug-in module called an **SFP**. The switch port is a slot. You choose the distance by choosing the module.

Module type	Typical rate	Reach
SFP	1 Gbit/s	550 m to 80 km
SFP+	10 Gbit/s	300 m to 80 km
SFP28	25 Gbit/s	100 m to 10 km
QSFP28	100 Gbit/s	100 m to 40 km
QSFP-DD	400 Gbit/s	500 m to 40 km

■ 23.3.4 PLAIN – what is really happening inside

1. The transmitter is a laser diode, or for the cheapest short links an LED. It is switched on and off billions of times a second.
2. The receiver is a photodiode. Light landing on it frees electrons and makes a tiny current, which an amplifier turns back into a digital signal.
3. Three things degrade the signal along the way, and they are different from the copper problems.
4. **Attenuation** is the light simply getting dimmer, absorbed and scattered by the glass. Measured in decibels per kilometre.
5. **Modal dispersion** is the smearing caused by different paths in multi-mode fibre. It is the reason multi-mode has a short reach.
6. **Chromatic dispersion** is the smearing caused by different wavelengths in the pulse travelling at slightly different speeds. It limits long single-mode links.
7. Glass is not equally transparent at all colours. There are three windows where it is unusually clear, and every optical network uses them.
8. Those windows are around 850 nanometres, 1310 nanometres and 1550 nanometres. All are infrared. None are visible to the eye.
9. 1550 nm is the clearest of the three, which is why undersea cables use it.
10. Because different wavelengths do not interfere, you can send many separate channels down one fibre at once, each on its own colour. That is **wavelength division multiplexing**.

11. Radio works on the same idea of a carrier wave, but the medium is shared with everyone in range, and the signal fades, reflects and gets absorbed by walls and bodies.

■ 23.3.5 TECHNICAL – the engineer’s version

1. Standard telecom fibre has a 125 micrometre cladding diameter regardless of type. Only the core differs.
2. Single-mode fibre has a core of about 8 to 10 micrometres. Multi-mode has 50 or 62.5 micrometres.
3. Single-mode is classified OS1 (indoor, tight buffered) and OS2 (outdoor, loose tube, lower loss). Multi-mode is OM1 through OM5.
4. Typical attenuation figures: about 3 dB/km at 850 nm on multi-mode, about 0.35 dB/km at 1310 nm and about 0.2 dB/km at 1550 nm on single-mode. Corning’s Vascade EX2500 submarine fibre is specified at a nominal 0.148 dB/km at 1550 nm.
5. Total internal reflection occurs when light in the core meets the cladding at an angle greater than the critical angle, set by the two refractive indices. The **numerical aperture** is the sine of the largest input angle that will be guided; typical multi-mode NA is 0.2 to 0.29.
6. Single-mode fibre is used for essentially all links longer than about 1 km, and multi-mode inside buildings and data halls.

Fibre grade	Core	10G reach	100G reach
OM3	50 um	300 m	70 m
OM4	50 um	400 m	100 m
OM5	50 um	400 m	150 m
OS2	9 um	10 to 80 km	10 to 80 km

7. The pluggable optic form factors are defined by multi-source agreements (MSAs) between vendors, not by IEEE. SFP dates from 2001, SFP+ from 2006, QSFP+ from 2009, QSFP28 from 2014, QSFP-DD and OSFP from 2016 onward.
8. This is a good example of a **convention** rather than a standard: an MSA is an industry agreement with no standards body behind it, which is why vendor lock-in on optics is so common.
9. Coarse WDM (CWDM) uses 8 to 18 channels spaced 20 nm apart. Dense WDM (DWDM) uses 40 to 96 or more channels spaced 0.8 nm or 0.4 nm apart on the ITU-T G.694.1 grid. A single modern DWDM fibre pair carries tens of terabits per second.
10. Light travels through glass at about 200,000 km/s, roughly two-thirds of its speed in vacuum, because the refractive index of silica is about 1.47. That gives about 5 microseconds of delay per kilometre of fibre, and it is a hard physical floor on latency.
11. Radio is IEEE 802.11 for WiFi and 3GPP specifications for cellular. Chapter 31 covers radio fully: modulation, channels, the 2.4, 5 and 6 GHz bands, CSMA/CA, MIMO, OFDMA and why WiFi behaves so differently from a cable. We say only this much here: the reader’s laptop reached the router over radio, and the fault in this book’s running example was not in that radio hop.
12. Observation commands: on a switch, show interfaces transceiver reports per-lane optical transmit and receive power in dBm, which is how you tell a dirty connector from a dead laser.

■ 23.3.6 WORDS – remember these

1. Core — the middle of the fibre that carries light — the higher-refractive index region, 8 to 10 um single-mode, 50 or 62.5 um multi-mode.
2. Cladding — the outer glass that keeps light in — the lower-index layer, standard 125 um outer diameter.

3. Total internal reflection — light bouncing back instead of escaping — reflection above the critical angle at a refractive-index boundary.
4. Single-mode — one path only, goes far — small-core fibre supporting one propagation mode, no modal dispersion.
5. Multi-mode — many paths, goes less far — large-core fibre with modal dispersion limiting reach and bandwidth.
6. Attenuation — the signal getting weaker — optical power loss in dB per km.
7. Dispersion — the pulse smearing out — modal or chromatic pulse spreading that limits the bit rate over distance.
8. SFP — the plug-in module that does the light — small form-factor pluggable transceiver defined by an MSA, not by IEEE.
9. WDM — many colours down one fibre — wavelength division multiplexing on the ITU-T G.694.1 grid.

23.4 Bandwidth, throughput, latency and jitter

■ 23.4.1 PLAIN – in simple words

1. People say “my internet is slow” and mean four completely different things.
2. **Bandwidth** is the maximum rate the link can carry. It is a capacity. It is what you bought.
3. **Throughput** is the rate you actually get for a particular transfer, right now. It is always less than bandwidth, sometimes far less.
4. **Latency** is the waiting time before anything arrives. How long one message takes to get there.
5. **Jitter** is how much that waiting time wobbles from message to message.
6. A video call needs low latency and low jitter. It barely needs bandwidth.
7. A large download needs bandwidth. It does not care about latency at all.
8. A game needs low latency above everything.
9. These four are almost independent. A link can have enormous bandwidth and terrible latency, and a link can have tiny bandwidth and superb latency.
10. And there is a fifth state that is none of these: the link can be silent. Nothing comes back at all. That is what happened to the reader.

■ 23.4.2 PLAIN – a picture in your head

1. The classic picture is a water pipe.
2. Bandwidth is how wide the pipe is. A wide pipe moves more water per second.
3. Latency is how long the pipe is. A long pipe means water takes longer to appear at the far end, even if the pipe is very wide.
4. Throughput is how much water actually comes out per second, which depends on the pipe, the tap, the pump, and anything half-blocking the middle.
5. Jitter is the pulsing you feel when the flow is not steady.
6. Widening the pipe does nothing for the time the first drop takes to appear. That is the single most useful thing this picture teaches.

Where this comparison breaks: it breaks in three important places, and engineers get caught by all three. First, water in a pipe is continuous, but data is sent in discrete packets, so a packet that is one byte too big for the pipe is not trimmed, it is dropped whole. Second, a water pipe does not lose water and then resend it, but a network does, and the resend costs a full round trip of latency that the pipe picture cannot represent. Third, and most importantly here, a blocked water pipe still tells you something: pressure builds, or water comes back. A silently dropping network gives you

nothing at all. There is no equivalent of a packet vanishing without trace in the water picture, and that vanishing is precisely what the reader was looking at.

■ 23.4.3 PLAIN – a worked example

1. The reader's own case is the clearest example in this book.
2. The command was `curl -v https://github.com`. It printed `Trying 20.207.73.82:443...` and then nothing at all for fifteen seconds.
3. Consider what each of the four measures would have looked like if it were the problem.
4. Low bandwidth: the page would have loaded, slowly, byte by byte. It did not load at all.
5. High latency: the page would have loaded after a long pause. It never loaded.
6. Low throughput because of congestion: the transfer would have stalled and restarted. There was no transfer to stall.
7. High jitter: the page would have loaded unevenly. There was nothing to be uneven about.
8. What actually happened is that the connection attempt got no reply of any kind. No refusal, no error, no data. Silence.
9. That is not slowness. That is a packet going into the network and never producing any observable effect.
10. The same site loaded instantly on mobile data from the same phone. So the site was up, and the destination was reachable from India.
11. What that proves: the failure was specific to the path taken from the reader's broadband line, not to the destination.
12. What it merely suggests: something on that path was discarding packets silently. It does not tell us which device, or why. Later chapters in Part F narrow that down.

■ 23.4.4 PLAIN – what is really happening inside

1. Latency is made of four separate delays, and they add up.
2. **Propagation delay** is the time the signal takes to physically travel. In fibre, about 5 microseconds per kilometre. Nothing can beat this.
3. **Transmission delay** is the time to push all the bits of a message out of the interface. It equals message size divided by link rate.
4. **Processing delay** is the time each router takes to look at the packet and decide where to send it. Modern hardware does this in microseconds.
5. **Queuing delay** is the time the packet spends waiting in a buffer behind other packets. This is the one that varies, and it is the main source of jitter.
6. Now the key idea. A link with high bandwidth and high latency can hold a surprising amount of data that has left the sender but not yet arrived.
7. That quantity has a name: the **bandwidth-delay product**. It is the capacity of the pipe itself, measured in bytes in flight.
8. If the sender is only allowed a certain number of unacknowledged bytes at a time, and that number is smaller than the bandwidth-delay product, the link can never be filled, however fast it is.
9. This is why a 1 Gbit/s link from India to the United States can deliver 3 Mbit/s to a badly tuned application. The link is not slow. The window is too small.

■ 23.4.5 TECHNICAL – the engineer’s version

1. Bandwidth in networking means capacity in bits per second. In signal processing it means a frequency range in hertz. The two are related through the Shannon-Hartley theorem but are not the same quantity. Say which you mean.
2. Round-trip time (RTT) is what ping measures. One-way latency is roughly half of it, but only on a symmetric path, and many paths are not symmetric.
3. Bandwidth-delay product, worked in full:

```

Link rate      = 100 Mbit/s = 100,000,000 bits/s
RTT           = 180 ms      = 0.180 s
BDP (bits)    = 100e6 * 0.180 = 18,000,000 bits
BDP (bytes)   = 18,000,000 / 8 = 2,250,000 bytes
              = about 2.15 MiB in flight

```

4. 180 ms is a realistic round trip from an Indian home line to a server on the US east coast. 2.15 MiB must be in flight at all times to keep that link full.
5. The original TCP receive window field is 16 bits, so a maximum of 65,535 bytes. That is 3 percent of what this link needs.
6. RFC 1323 (1992), now superseded by RFC 7323 (2014), added the window scale option, which multiplies the window by a power of two up to 2^{14} . Without it, long fat networks cannot be filled.
7. Maximum achievable throughput with a fixed window is window / RTT. With 65,535 bytes and 180 ms that is about 2.9 Mbit/s, on a 100 Mbit/s link.
8. TCP and its congestion control get a full chapter later in Part F. The point here is only that link speed alone never predicts throughput.

Quantity	Unit	Typical home value
Bandwidth	Mbit/s	100 to 1000
Throughput	Mbit/s	40 to 900
Latency to gateway	ms	0.5 to 3
Latency India to US	ms	180 to 280
Jitter, good WiFi	ms	1 to 5
Jitter, congested	ms	30 to 200

9. Jitter is formally the variation in one-way delay. RFC 3550, the RTP specification (2003), defines an interarrival jitter estimator that media applications use to size their playout buffer.
10. **Bufferbloat** is the condition where oversized, unmanaged buffers in a modem or router hold hundreds of milliseconds of packets, so latency explodes under load while bandwidth tests still look fine. Jim Gettys named and characterized it in 2010. The fixes are active queue management algorithms: CoDel (2012) and fq_codel, and the CAKE qdisc on Linux.
11. Tools: ping for RTT, iperf3 for throughput, mtr for per-hop loss and latency, and flent running the RRUL test for bufferbloat under load.
12. The honest version: ping measuring 20 ms does not mean your data has 20 ms of latency. ICMP echo is often handled on a different, slower or faster path inside a router than forwarded traffic, and is frequently rate-limited or deprioritized. This matters enormously for reading a traceroute, which is a later chapter in Part F.
13. For the reader’s case none of these numbers apply, because there was no measurable transfer. The correct technical description is a connection attempt that received neither a SYN-ACK, nor a TCP RST, nor an ICMP unreachable message, within the 15 second observation window.

■ 23.4.6 WORDS – remember these

1. Bandwidth — how wide the pipe is — link capacity in bits per second, or a frequency range in hertz depending on context.
2. Throughput — what you actually get — achieved data rate for a given flow over a given interval, also called goodput when headers are excluded.
3. Latency — the waiting time before anything happens — one-way or round-trip delay, the sum of propagation, transmission, processing and queuing delays.
4. Jitter — how much the wait wobbles — variance in one-way delay, estimated per RFC 3550 in real-time media.
5. Propagation delay — the time light or electricity takes to travel — distance divided by signal velocity, about 5 us per km in fibre.
6. Bandwidth-delay product — how much data fits inside the link — link rate multiplied by RTT, the in-flight capacity in bytes.
7. Bufferbloat — big buffers making everything laggy — excess queuing delay from unmanaged FIFO buffers, fixed by AQM such as fq_codel.
8. Silent drop — nothing comes back at all — a discard with no ICMP or TCP notification, indistinguishable from a black hole from the sender's side.

23.5 Ethernet

■ 23.5.1 PLAIN – in simple words

1. Ethernet is the set of rules for sending data over a local network cable. It is more than fifty years old and it won completely.
2. It was invented in 1973 at a research lab in California called Xerox PARC, by Robert Metcalfe and David Boggs.
3. The idea it copied came from Hawaii. A radio network called ALOHAnet let many machines share one radio channel by simply transmitting and retrying after a clash.
4. Metcalfe's version used a cable instead of radio, and added one crucial rule: listen before you talk.
5. The name comes from "luminiferous ether", the invisible substance nineteenth century physicists thought light travelled through. It was a joke about the cable being the medium everything swims in.
6. In the beginning, every machine was tapped onto one thick coaxial cable running through the building. One cable, shared by all.
7. Because it was shared, two machines could talk at once and ruin each other's message. That is a **collision**.
8. The rule for handling it was: listen first, and if a collision happens anyway, stop, wait a random time, and try again.
9. That rule is CSMA/CD, which stands for carrier sense multiple access with collision detection.
10. Today, on almost every network you will ever touch, collisions cannot happen, because the shared cable is gone.

■ 23.5.2 PLAIN – a picture in your head

1. Picture a small meeting where everyone shares one microphone that is always on.
2. Before speaking, you listen. If someone is talking, you wait. That is carrier sense.
3. Sometimes two people start in the same instant, because each heard silence. Both voices go out together and neither is understandable. That is a collision.
4. Both notice the clash while it is happening, stop immediately, and each waits a different random number of seconds before trying again. That is collision detection and backoff.
5. Now replace the meeting with a room where every person has a private two-way headset to a chairperson.
6. Anyone can speak at any time, in both directions at once, and the chairperson passes each message only to the person it is for.
7. Nobody ever clashes with anybody. That is a modern switched, full-duplex network.

Where this comparison breaks: humans hear a clash instantly. On a long cable, electricity takes time to travel, so a station at one end can be halfway through transmitting before it hears the clash from the far end. This is why classic Ethernet had a minimum frame size and a maximum cable length that were mathematically linked. If a frame were too short, the sender could finish before the collision signal got back, and would never know it had failed.

■ 23.5.3 PLAIN – a worked example

1. Take the original 10 Mbit/s shared Ethernet and work out why the minimum frame is 64 bytes.
2. The rule is: a sender must still be transmitting when a collision from the far end reaches it.
3. Worst case, the signal must go all the way to the far end and the collision must come all the way back. That is a full round trip.
4. The standard fixes this round trip budget at **512 bit times**, which is the time to send 512 bits.

512 bits at 10 Mbit/s	= 51.2 microseconds
512 bits / 8	= 64 bytes

5. So a frame must be at least 64 bytes long, or the sender might finish too early. 64 bytes counts from the destination address through the check sequence.
6. That 51.2 microsecond budget also caps the cable. Signals travel through coax at roughly two thirds the speed of light, and repeaters add delay, so the maximum end-to-end network was 2500 m with at most four repeaters.
7. If a frame arrives shorter than 64 bytes, it is called a **runt** and is discarded. Historically a runt meant a collision. Today it means a faulty cable, a bad NIC or a duplex mismatch.
8. On a switched full-duplex link there is no shared medium and no collision, so none of this timing matters. The 64-byte minimum survives anyway, purely for compatibility.

■ 23.5.4 PLAIN – what is really happening inside

1. A modern Ethernet link between a laptop and a switch is two independent one-way channels using different wire pairs.
2. Each end can transmit whenever it likes, because nobody else is on its transmit path. This is **full duplex**.
3. When the cable is plugged in, the two ends negotiate. They exchange short pulse bursts advertising every speed and duplex mode they support.
4. They then both pick the best mode that both support. This is **auto-negotiation**, and it is why plugging in a cable just works.

5. If negotiation fails or is disabled at one end, you can get a **duplex mismatch**: one side full duplex, the other half duplex.
6. That link will still pass traffic, but the half-duplex side will report collisions and late collisions and throughput will collapse under load. It is a classic slow-network fault.
7. The switch in the middle looks at the destination address of each frame and sends it out one port only. Section 23.8 explains exactly how it knows which.
8. Because each link is private and full duplex, the network has as many collision domains as it has ports, and each of those domains has exactly one possible talker in each direction. So there are no collisions.

■ 23.5.5 TECHNICAL – the engineer’s version

1. The experimental Ethernet first ran on 22 May 1973 at Xerox PARC at 2.94 Mbit/s. That odd rate was the Alto computer’s system clock divided down.
2. Metcalfe and Boggs published “Ethernet: Distributed Packet Switching for Local Computer Networks” in Communications of the ACM in July 1976.
3. US patent 4,063,220, “Multipoint data communication system with collision detection”, was granted in 1977 to Xerox, naming Metcalfe, Boggs, Thacker and Lampson.
4. The DIX consortium of Digital Equipment Corporation, Intel and Xerox published the Ethernet Blue Book in 1980 at 10 Mbit/s. Version 2, the frame format still called Ethernet II, was published in November 1982.
5. Metcalfe founded 3Com in June 1979 to commercialize it.
6. IEEE published 802.3 as a draft in 1983 and as a full standard in 1985. The competing standards, Token Ring (IEEE 802.5, 1985) and FDDI (ANSI X3T9.5, 1987), were technically respectable and lost on cost.
7. The 802.3 speed family, with the amendment letter and year:

Amendment	Year	What it added
802.3	1983	10BASE5, 10 Mbit/s coax
802.3i	1990	10BASE-T, twisted pair
802.3u	1995	100BASE-TX, Fast Ethernet
802.3z	1998	1000BASE-X, gigabit fibre
802.3ab	1999	1000BASE-T, gigabit copper
802.3ae	2002	10 Gbit/s over fibre
802.3an	2006	10GBASE-T over copper
802.3ba	2010	40 and 100 Gbit/s
802.3bz	2016	2.5G and 5GBASE-T
802.3bq	2016	25G and 40GBASE-T
802.3bs	2017	200 and 400 Gbit/s
802.3df	2024	800 Gbit/s, 100G lanes

8. Media naming is systematic: the number is the speed in Mbit/s or Gbit/s, BASE means baseband (no carrier modulation of the whole channel), and the suffix names the medium. T is twisted pair, S is short-wave multi-mode fibre, L is long-wave single-mode, C is twinaxial copper, R is a 64b/66b coded lane.
9. 10BASE5 was 10 mm thick coax, up to 500 m per segment, tapped with a vampire tap that pierced the jacket. 10BASE2 was thinner coax with BNC T-pieces, 185 m per segment, and was universally called thinnet or cheapernet.

10. The connector story: coax with N-type or BNC, then the 8P8C modular plug that everyone calls RJ45 for 10BASE-T onward. RJ45 is technically the wrong name, since RJ45 is a telephone registered jack specification, but the usage is universal. That is a convention, not a standard.
11. Pin assignments come from TIA-568A and TIA-568B, which differ only in swapping the green and orange pairs. A cable with one end A and one end B is a crossover cable.
12. Crossover cables are no longer needed because Auto-MDI-X, standardized as part of 802.3ab in 1999 and now universal, detects and swaps the pairs electronically.
13. Interframe gap is 96 bit times: 9.6 us at 10 Mbit/s, 0.96 us at 100 Mbit/s, 96 ns at 1 Gbit/s. It exists so receivers can resynchronize between frames.
14. Half-duplex gigabit exists in the standard, using carrier extension to pad the slot time to 4096 bit times, but no one ever deployed it.
15. Observation: `ethtool eth0` on Linux shows negotiated speed, duplex and auto-negotiation state; `networksetup -listallhardwareports` and `ifconfig` on macOS show the equivalent, and `netstat -i` shows error counters.

■ 23.5.6 WORDS – remember these

1. Ethernet — the standard way to wire a local network — the IEEE 802.3 family of physical and MAC layer specifications.
2. CSMA/CD — listen, talk, stop if you clash — carrier sense multiple access with collision detection, mandatory on half-duplex shared media.
3. Collision — two stations transmitting at once — overlapping transmissions within one collision domain, detected by the transceiver.
4. Backoff — waiting a random time before retrying — truncated binary exponential backoff over a doubling contention window.
5. Full duplex — both directions at once — simultaneous independent transmit and receive on separate pairs or wavelengths.
6. Auto-negotiation — the cable sorting itself out — link code word exchange using fast link pulses to select speed, duplex and flow control.
7. Duplex mismatch — one side full, one side half — a misconfiguration causing late collisions and severe throughput loss.
8. Runt — a frame that is too short — a frame under 64 octets, discarded and counted as an error.
9. Interframe gap — the compulsory pause between frames — 96 bit times of idle before the next preamble.

23.6 The Ethernet frame, field by field

■ 23.6.1 PLAIN – in simple words

1. Everything sent on an Ethernet network is wrapped in a **frame**.
2. A frame is like an envelope. The data you care about is the letter inside. The envelope carries the addressing and the checks.
3. The envelope has a fixed layout, and every network card in the world agrees on it. That agreement is what makes any card talk to any other.
4. It begins with a wake-up pattern so the receiver can lock onto the timing.
5. Then the address it is going to, then the address it came from.
6. Then a two-byte code saying what kind of thing is inside. Is this an IP packet, or an ARP message, or something else.
7. Then the data itself, between 46 and 1500 bytes.

8. Then a four-byte check number computed from everything before it.
9. The receiver recomputes that check number. If it does not match, the frame is thrown away silently. Ethernet does not ask for a resend. Something higher up must notice.
10. The whole frame, not counting the wake-up pattern, is between 64 and 1518 bytes.

■ 23.6.2 PLAIN – a picture in your head

1. Think of a postcard going through an old sorting office.
2. The first thing the machine needs is a run of identical marks along the edge so it can line the card up and match its rollers to the card's speed. That is the preamble.
3. Then one different mark saying "the card starts here". That is the start frame delimiter.
4. Then the delivery address, then the return address.
5. Then a small printed code saying what class of item this is: a letter, a parcel slip, a legal notice. That is the EtherType.
6. Then the message.
7. Then a checksum stamped at the end, computed from the whole card by the sending office. The receiving office recomputes it and bins the card if they disagree.
8. Postcards have a minimum size because the machine cannot grip anything smaller, and a maximum size because the slot is only so wide.

Where this comparison breaks: a sorting office that bins a damaged card usually tells someone. Ethernet tells nobody. A frame failing its check sequence is counted in a statistics register on the interface and then forgotten. Also, a postcard's return address is written by the sender and can be a lie; so can an Ethernet source address, and that is the root of several attacks in section 23.10.

■ 23.6.3 PLAIN – a worked example

1. Here is a real frame, an ARP request from the reader's laptop asking who owns 192.168.0.1. MAC addresses are illustrative, since the session record does not include them.

Preamble	AA AA AA AA AA AA AA	(7 bytes, alternating)
SFD	AB	(1 byte)
Dest MAC	FF FF FF FF FF FF	(6 bytes, broadcast)
Src MAC	3C 22 FB 8A 41 D7	(6 bytes, the laptop)
EtherType	08 06	(2 bytes, ARP)
Payload	00 01 08 00 06 04 00 01 3C 22 FB 8A 41 D7 C0 A8 00 6B 00 00 00 00 00 00 C0 A8 00 01	(28 bytes of ARP, padded to 46)
Padding	00 x 18	(to reach 46 bytes)
FCS	9C 4E 1B 33	(4 bytes, CRC-32)

2. C0 A8 00 6B is 192.168.0.107 in hexadecimal. That is the laptop.
3. C0 A8 00 01 is 192.168.0.1 in hexadecimal. That is the reader's router.
4. Count the frame from destination MAC to FCS: $6 + 6 + 2 + 46 + 4 = 64$ bytes. Exactly the minimum.
5. The ARP message itself is only 28 bytes, so 18 bytes of zero padding were added to reach the 46-byte minimum payload.
6. The preamble and SFD are not counted in the 64. They are physical-layer framing, stripped before the frame is handed up.
7. A packet capture tool will normally show you this frame as 60 bytes, because it also strips the 4-byte FCS after checking it.

■ 23.6.4 PLAIN – what is really happening inside

1. The preamble is 56 bits of alternating 1 and 0. On the wire this is a square wave, which is the easiest possible thing for a receiver's clock circuit to lock onto.
2. The start frame delimiter breaks the pattern with two 1 bits in a row. That single break is what says "data starts on the next bit".
3. The EtherType field does double duty, and this is a historical scar. Values of 1500 or below mean "this is a length in bytes", the original IEEE 802.3 meaning. Values of 1536 or above mean "this is a type code", the DIX Ethernet II meaning.
4. Because 1500 is below 1536, both interpretations can coexist unambiguously, which is why the maximum payload is 1500 and not something rounder.
5. The check sequence is a CRC-32, a cyclic redundancy check. It is not a simple sum. The frame is treated as one enormous binary number and divided by a fixed polynomial; the remainder is the check value.
6. CRC-32 catches all single-bit errors, all double-bit errors, all odd numbers of errors, and all burst errors up to 32 bits. Beyond that it catches all but about 1 in 4 billion.
7. The maximum payload of 1500 bytes is called the **MTU**, maximum transmission unit.
8. The number 1500 came from the 1980 DIX specification. Buffer memory was expensive in 1979, and a station holding the shared coax cable for too long starved everybody else. 1500 bytes at 10 Mbit/s is about 1.2 milliseconds of airtime, which was judged a fair maximum turn.
9. It has never changed, on any Ethernet speed, for compatibility. A 400 Gbit/s link still defaults to a 1500-byte MTU chosen for a 10 Mbit/s coax cable in 1979.
10. **Jumbo frames** raise the payload to around 9000 bytes. Nine thousand was chosen because CRC-32's error detection strength starts to weaken beyond about 12000 bytes, and 9000 comfortably holds an 8192-byte block plus headers.
11. Jumbo frames are not in the IEEE standard. They are a vendor convention. Every device on the path must agree, or frames are silently dropped.

■ 23.6.5 TECHNICAL – the engineer's version

1. The complete frame layout, in transmission order:

Field	Bytes	Purpose
Preamble	7	Clock synchronization
Start frame delimiter	1	Marks start of frame
Destination MAC	6	Receiver address
Source MAC	6	Sender address
802.1Q tag	4	Optional VLAN tag
EtherType or length	2	Payload type or size
Payload	46 to 1500	The data
Frame check sequence	4	CRC-32
Interframe gap	12	Idle before next frame

2. Preamble is 10101010 repeated seven times. The SFD is 10101011. Together they are 64 bits, often described as an 8-byte preamble ending in two 1 bits.
3. Minimum frame is 64 octets measured from destination MAC through FCS inclusive. Preamble, SFD and IFG are not counted.

4. Maximum untagged frame is 1518 octets: 6 + 6 + 2 + 1500 + 4. With one 802.1Q tag it is 1522. With two stacked tags (802.1ad, “QinQ”) it is 1526.
5. With an 802.1Q tag present the minimum payload is 42 bytes rather than 46, so the 64-octet minimum still holds.
6. The FCS is a left-shifting CRC-32 with polynomial 0x04C11DB7, initial value 0xFFFFFFFF, post-complemented, giving a residue check value of 0x38FB2284.
7. Common EtherType values, all in hexadecimal, assigned by the IEEE Registration Authority:

EtherType	Protocol
0x0800	IPv4
0x0806	ARP
0x8100	802.1Q VLAN tag
0x86DD	IPv6
0x8847	MPLS unicast
0x88CC	LLDP
0x8863	PPPoE discovery
0x8864	PPPoE session

8. MTU is the layer 3 payload limit, 1500 bytes for standard Ethernet. It is distinct from the maximum frame size, which includes the 18 bytes of header and FCS.
9. Fragmentation costs, and this is why MTU matters. If an IP packet is larger than the path MTU, it must be split. Every fragment carries a full IP header, so overhead rises. If any one fragment is lost, the whole original packet is lost. Firewalls and load balancers often handle fragments badly or drop them. Fragment reassembly consumes memory at the receiver and is a classic denial of service target.
10. IPv4 fragmentation is defined in RFC 791 (1981). IPv6 removed router fragmentation entirely in RFC 8200 (2017); only the source may fragment.
11. Path MTU discovery, RFC 1191 (1990) for IPv4 and RFC 8201 (2017) for IPv6, finds the smallest MTU on a path by sending packets with the Don't Fragment bit set and listening for ICMP “fragmentation needed” replies.
12. The honest version: PMTUD is fragile in the real world, because many networks block all ICMP, so the “fragmentation needed” message never arrives and the connection hangs after the handshake. This is the classic PMTUD black hole, and RFC 4821 (2007) defines Packetization Layer PMTUD to work around it by probing at the TCP layer instead.
13. PPPoE, used on many DSL lines, adds 8 bytes of header and leaves an MTU of 1492. VPNs and tunnels reduce it further, which is why tunnel interfaces often default to 1400 or lower.
14. The reader's macOS machine showed several utun interfaces. Those are tunnel interfaces, and each carries its own smaller MTU. Check with `ifconfig utun0` and look for the `mtu` value.
15. Jumbo frame support is queried with `ip link show` on Linux or `ifconfig` on macOS; on a switch it is often called “system jumbo mtu” or “maximum frame size” and applies per port or globally depending on the vendor. That is an implementation detail, not a standard.

■ 23.6.6 WORDS – remember these

1. Frame — the envelope around your data on a local network — a layer 2 protocol data unit delimited by preamble and interframe gap.
2. Preamble — the wake-up pattern at the start — 56 bits of alternating 1 and 0 for receiver clock recovery.
3. EtherType — the code saying what is inside — a 2-byte protocol identifier when the value is 1536 or greater.
4. Payload — the actual data being carried — 46 to 1500 octets in a standard untagged frame.
5. FCS — the check number at the end — a 32-bit CRC over the frame, polynomial 0x04C11DB7.
6. MTU — the largest chunk of data one frame can carry — maximum transmission unit, 1500 octets for standard Ethernet.
7. Jumbo frame — a bigger than normal frame — a non-standard payload of about 9000 octets, requiring agreement across the whole path.
8. Fragmentation — cutting a packet up to fit — splitting an IP datagram across multiple frames, with per-fragment header cost and all-or-nothing loss.
9. Runt and giant — too small and too big — frames below 64 or above the configured maximum, both counted as errors.

23.7 MAC addresses

■ 23.7.1 PLAIN – in simple words

1. Every network interface has a hardware address burned into it. It is called a **MAC address**.
2. MAC stands for media access control. The name is an accident of which sublayer of the standard defines it.
3. It is 48 bits long, which is 6 bytes. It is written as six pairs of hexadecimal digits.
4. Example: 3C:22:FB:8A:41:D7. Some systems use hyphens or dots instead of colons. Same address.
5. It is meant to be globally unique, so that no two interfaces anywhere have the same one.
6. The first three bytes identify the manufacturer. The last three are the manufacturer's own serial number for that interface.
7. So from a MAC address you can usually tell what kind of device it is, without asking the device.
8. An address is either for one device, for a group of devices, or for everyone on the network.
9. The everyone address is all ones: FF:FF:FF:FF:FF:FF. A frame sent there is delivered to every device on the local network.
10. Modern phones and laptops now often make up a fake MAC address on purpose, to stop shops and networks tracking you as you walk around.

■ 23.7.2 PLAIN – a picture in your head

1. Think of a MAC address like the serial number stamped on the back of an appliance.
2. The first part identifies the factory. The second part identifies the individual unit that factory made.
3. Two appliances from the same factory share the first part and differ in the second.
4. Unlike a postal address, it says nothing about where the device is. It travels with the device.
5. A postal address changes when you move house. A serial number does not. This is exactly the difference between an IP address and a MAC address.
6. That difference is why both exist, and it is the reason ARP has to exist at all.

Where this comparison breaks: a serial number is permanent, but a MAC address can be changed in software on almost every operating system in about one command. Uniqueness is administrative, not physical. Manufacturers have shipped duplicate addresses by mistake, and cheap unbranded network chips have shipped whole batches with identical addresses.

■ 23.7.3 PLAIN – a worked example

1. Take the address 3C:22:FB:8A:41:D7.
2. Split it: 3C:22:FB is the vendor part, 8A:41:D7 is the device part.
3. The vendor part is called the **OUI**, organizationally unique identifier. It is bought from the IEEE.
4. 3C:22:FB is registered to Apple. So this is very likely a Mac or an iPhone.
5. You can look up any OUI in the IEEE public registry, or with a local tool. On many Linux systems, `/usr/share/ieee-data/oui.txt` holds the whole list.
6. Now look at the first byte alone: 3C is 00111100 in binary.
7. The last bit of that first byte is 0. That means unicast: one device.
8. If it were 1, it would be multicast: a group. 01:00:5E: . . . is the IPv4 multicast group prefix, and 33:33: . . . is the IPv6 one.
9. The second-to-last bit of the first byte is 0. That means the address is globally assigned by a real manufacturer.
10. If it were 1, the address is **locally administered**, meaning somebody made it up. Randomized privacy addresses always have this bit set to 1.
11. So 3C ends in 00: globally unique, unicast, real Apple hardware.
12. Compare with 3E:22:FB:8A:41:D7. 3E is 00111110. Last bit 0, so still unicast, but second-to-last bit is 1, so this one was made up locally.

■ 23.7.4 PLAIN – what is really happening inside

1. The network card holds its address in a small memory chip, and the driver reads it when the interface starts.
2. When the card receives a frame, it compares the destination address against a short list without waking the CPU.
3. The list is: its own address, the broadcast address, and any multicast groups it has been told to join.
4. If the destination matches nothing on that list, the card discards the frame in hardware. The operating system never sees it.
5. Multicast matching is often done with a hash filter rather than exact matching, so the card sometimes lets through groups it did not join. The driver filters those out in software.
6. There is a special mode where the card stops filtering entirely and passes up everything it hears. That is **promiscuous mode**, and section 23.12 covers it.
7. Address randomization works by generating a random 46-bit value, setting the locally administered bit to 1 and the multicast bit to 0, and using that as the interface address.
8. On phones, a different random address is typically generated per network name, so the same shop sees a consistent device but two different shops cannot link you.

■ 23.7.5 TECHNICAL – the engineer's version

1. The 48-bit format is IEEE EUI-48. There is also a 64-bit EUI-64 used by IPv6 interface identifiers, FireWire and ZigBee.
2. The IEEE Registration Authority sells three block sizes: MA-L (a 24-bit OUI, 16.7 million addresses), MA-M (28-bit prefix, 1 million addresses) and MA-S (36-bit prefix, 4096 addresses).

3. Bit ordering matters and confuses everyone. Ethernet transmits each octet least-significant bit first, so the I/G bit is literally the first bit on the wire, which is why the hardware can begin filtering immediately.
4. Bits in the first octet:

Bit	Name	Meaning if 1
Least significant	I/G	Multicast or broadcast
Second least	U/L	Locally administered

5. Broadcast is FF:FF:FF:FF:FF:FF, all 48 bits set, which is simply the all-groups multicast address.
6. IPv4 multicast maps to 01:00:5E plus the low 23 bits of the group address, per RFC 1112 (1989). IPv6 multicast maps to 33:33 plus the low 32 bits, per RFC 2464 (1998).
7. Ranges reserved for local use without IEEE registration are defined in IEEE 802c (2017), which carves the locally administered space into structured quadrants.
8. MAC randomization history, with dates, since this changes fast: Apple iOS 8 randomized addresses during WiFi scanning in 2014; Android 10 made randomized addresses the default for connections in 2019; Apple shipped Private WiFi Address in iOS 14 and iPadOS 14 in 2020, and macOS gained a rotating variant later. Windows 10 has had a per-network random hardware address option since 2015.
9. Since iOS 18 and macOS 15 (2024) Apple's default is a rotating private address that changes periodically on networks the device does not consider trusted. Check current behaviour before relying on it; this area moves.
10. Consequences you must plan for: MAC-based DHCP reservations break, MAC filtering as a security control was always weak and is now useless, and network access control must use 802.1X with real credentials instead.
11. Commands to read your own address:

```
# macOS, all interfaces with hardware addresses
ifconfig | grep ether

# macOS, which port is which
networksetup -listallhardwareports

# macOS, the WiFi interface specifically
ifconfig en0 | grep ether

# Linux
ip link show
cat /sys/class/net/eth0/address
```

12. To see the vendor decoded for you, packet capture tools do it automatically: tcpdump -e prints Ethernet headers, and Wireshark shows Apple_8a:41:d7 style names using its own OUI database.
13. The honest version: MAC addresses do not travel across routers. A router strips the incoming frame and builds a new one with new source and destination MAC addresses for the next hop. The MAC address the reader's router sees is the laptop's. The MAC address GitHub's server sees is its own upstream router's. This is the single most important fact about MAC addresses and the reason section 23.10 exists.

■ 23.7.6 WORDS – remember these

1. MAC address — the hardware address of a network interface — a 48-bit EUI-48 identifier used for layer 2 delivery.
2. OUI — the manufacturer part of the address — organizationally unique identifier, the first 24 bits, allocated by the IEEE.
3. Unicast — addressed to one device — I/G bit clear in the first octet.
4. Multicast — addressed to a group — I/G bit set, with reserved prefixes 01:00:5E for IPv4 and 33:33 for IPv6.
5. Broadcast — addressed to everyone on the link — FF:FF:FF:FF:FF:FF, all bits set.
6. Locally administered — an address somebody made up — U/L bit set, not guaranteed unique outside the local link.
7. MAC randomization — your device using a fake address for privacy — rotating locally administered addresses, default on iOS 14+ and Android 10+.
8. EUI-64 — the 64-bit version — extended unique identifier used by IPv6 SLAAC and other technologies.

23.8 Hubs, switches and how a switch learns

■ 23.8.1 PLAIN – in simple words

1. In the middle of a star-shaped network sits a box that everything plugs into.
2. There have been two kinds of box, and the difference between them is the biggest single improvement in local networking history.
3. A **hub** is the dumb one. Anything arriving on any port is copied out of every other port, immediately, with no thought.
4. So every device hears everything. The whole hub is one shared medium, and collisions happen exactly as on the old coax cable.
5. A **switch** is the smart one. It looks at where the frame is going and sends it out of one port only.
6. To do that it needs to know which device is on which port. Nobody tells it. It works this out by itself, by watching.
7. Every frame carries a source address. When a frame comes in on port 3 saying it is from address X, the switch writes down “X is on port 3”.
8. It builds a table this way, purely by listening. This is called **learning**.
9. If a frame arrives for an address the switch has not learned yet, it does the only safe thing: it sends it out of every port except the one it came in on. That is **flooding**.
10. Hubs have been obsolete since roughly 2005. You will only meet one in a museum or a very old cupboard.

■ 23.8.2 PLAIN – a picture in your head

1. Picture a hotel with a single letterbox for the whole building, and a porter.
2. The hub porter takes every letter and photocopies it into every room’s pigeonhole. Everyone reads everything and throws away what is not theirs.
3. The switch porter starts on day one knowing nothing, with a blank notebook.
4. A letter comes out of room 12 with “from: Anita” on it. The porter writes “Anita, room 12” in the notebook.
5. Later a letter arrives addressed to Anita. The porter checks the notebook and walks it straight to room 12.

6. A letter arrives for Vikram, who is not in the notebook yet. The porter has no choice: a copy goes to every room. Somebody will be Vikram.
7. When Vikram replies, his letter has “from: Vikram” on it, and the porter writes down his room. From then on, letters to Vikram go straight there.
8. If a guest checks out and a new one moves in, the notebook goes stale. So each entry is rubbed out if it has not been seen for a while.

Where this comparison breaks: the porter walks at human speed and can think. A switch must make this decision in a few hundred nanoseconds, for millions of frames a second, simultaneously on every port. It does it with dedicated silicon and a content-addressable memory, not by looking things up in a list. Also, the porter would notice two guests claiming the same name. A switch does not object; it simply moves the entry, which is exactly how one class of attack works.

■ 23.8.3 PLAIN – a worked example

1. Three devices plug into a four-port switch. All addresses shortened for readability.

```
Port 1: laptop      AA:AA
Port 2: phone       BB:BB
Port 3: TV          CC:CC
Port 4: router      DD:DD   (192.168.0.1)
```

2. Step 0. The switch has just been powered on. Its table is empty.

```
MAC table: (empty)
```

3. Step 1. The laptop sends a frame to the router: source AA:AA, destination DD:DD, arriving on port 1.
4. The switch learns the source: AA:AA is on port 1. It looks up DD:DD, finds nothing, and floods the frame out of ports 2, 3 and 4.

```
MAC table:
AA:AA -> port 1
```

5. Step 2. The router replies: source DD:DD, destination AA:AA, arriving on port 4.
6. The switch learns DD:DD is on port 4. It looks up AA:AA, finds port 1, and sends the frame only to port 1. The phone and TV never see it.

```
MAC table:
AA:AA -> port 1
DD:DD -> port 4
```

7. Step 3. The phone sends to the router: source BB:BB, destination DD:DD, on port 2.
8. The switch learns BB:BB on port 2, looks up DD:DD, finds port 4, forwards only there. No flooding at all this time.

```
MAC table:
AA:AA -> port 1
BB:BB -> port 2
DD:DD -> port 4
```

9. The TV on port 3 has still never transmitted, so it is still unknown. Traffic for it will be flooded until it speaks once.

10. After three frames the switch has learned three of four devices and stopped flooding almost entirely. This is why switched networks scale and hubs do not.

■ 23.8.4 PLAIN – what is really happening inside

1. The table is called the **MAC address table**, or the forwarding database, or the CAM table depending on who is speaking.
2. Each entry holds an address, a port number, a VLAN identifier and a timestamp.
3. Entries expire. The default ageing time is 300 seconds on most vendors. Every time a frame is seen from that address, the timer resets.
4. Ageing is what lets you unplug a laptop from one port and plug it into another. Actually, the switch relearns instantly on the first frame from the new port, and ageing only cleans up devices that have gone away entirely.
5. There are two ways to forward a frame, and they trade latency against safety.
6. **Store-and-forward** reads the entire frame into memory, checks the FCS, and only then forwards it. Corrupt frames never leave the switch.
7. **Cut-through** starts forwarding as soon as it has read the destination address, six bytes in. Far lower latency, but it will happily forward a frame that turns out to be corrupt.
8. A **collision domain** is a set of ports that can interfere with each other. On a switch, each port is its own collision domain, and on full duplex that domain has no contention at all.
9. A **broadcast domain** is the set of ports a broadcast frame reaches. A whole switch is one broadcast domain by default, and so is a whole stack of switches wired together.
10. Splitting a broadcast domain requires either a router or a VLAN, which is section 23.9.

■ 23.8.5 TECHNICAL – the engineer’s version

1. The correct standards term for a switch is a **MAC bridge**, defined by IEEE 802.1D. “Switch” is universal industry usage, that is, a convention.
2. Learning is defined in 802.1D as the learning process, and flooding of unknown unicast is required behaviour, not an optimization.
3. Lookups use ternary content-addressable memory (TCAM) or hash-based exact-match tables. TCAM does a full parallel comparison in one clock cycle, at a substantial cost in silicon area and power.
4. Table sizes are a real limit. A small unmanaged switch may hold 1000 to 8000 entries. An access switch holds 16k to 32k. A data centre spine switch holds 256k or more.
5. When the table is full, new entries cannot be learned and all traffic to unknown addresses is flooded. **MAC flooding** is an attack that deliberately fills the table with bogus source addresses to turn a switch into a hub and enable eavesdropping. The defence is port security, limiting learned addresses per port.

Forwarding mode	Latency at 1 GbE	Corrupt frames
Store-and-forward	About 5 to 12 us	Dropped
Cut-through	About 1 to 3 us	Forwarded
Fragment-free	About 1 us plus 64B	Mostly dropped

6. Fragment-free is a compromise: wait for the first 64 bytes, which is where collision fragments would appear, then start forwarding.

7. Modern data centre switches quote port-to-port latency of 300 to 800 nanoseconds in cut-through mode. Store-and-forward latency scales with frame size, because the whole frame must arrive first; a 1518-byte frame at 1 Gbit/s takes about 12 us just to receive.
8. What a **managed** switch adds over an unmanaged one: VLANs (802.1Q), spanning tree (802.1D and successors), link aggregation (802.1AX, formerly 802.3ad), port mirroring for capture, quality of service queues (802.1p), 802.1X port authentication, LLDP neighbour discovery (802.1AB), SNMP monitoring, and per-port statistics and rate limiting.
9. Link aggregation is directly relevant to this book's running example. The traceroute in the reader's session showed interfaces named ae66-0, be23 and po22. Those prefixes mean aggregated Ethernet, bundle Ethernet and port-channel: three vendor words for several physical links bonded into one logical link. Those are naming conventions, not standards.
10. That bonding is also why hops 5, 6, 8, 9 and 11 in the reader's traceroute showed more than one address for a single hop. Packets in the same trace took different member links of the same bundle. That is load balancing working correctly, not a fault.
11. Observation commands, vendor-dependent: `show mac address-table` on Cisco and Arista, `show ethernet-switching table` on Juniper, `bridge fdb show` on Linux, and `ifconfig bridge0` on macOS for the built-in bridge.

■ 23.8.6 WORDS – remember these

1. Hub — repeats everything to everyone — a multiport repeater, one collision domain, obsolete since the early 2000s.
2. Switch — sends each frame to one port — a MAC bridge per IEEE 802.1D, learning and forwarding per destination address.
3. MAC address table — the switch's notebook — the filtering database mapping address and VLAN to egress port, with an ageing timer.
4. Learning — writing down who is where — populating the filtering database from observed source addresses.
5. Flooding — sending to everyone because you do not know — forwarding unknown unicast, broadcast and unregistered multicast to all ports in the VLAN.
6. Collision domain — where two talkers can clash — the set of ports sharing one contended medium; one per port on a switch.
7. Broadcast domain — how far a broadcast reaches — the set of ports a broadcast frame is flooded to, bounded by a router or a VLAN edge.
8. Store-and-forward — read it all, check it, then send — full-frame buffering with FCS validation before egress.
9. Cut-through — start sending after the address — forwarding begins after the destination MAC is read, propagating errored frames.
10. Managed switch — one you can configure — a bridge supporting 802.1Q, spanning tree, aggregation, mirroring and management protocols.

23.9 VLANs and spanning tree, briefly

■ 23.9.1 PLAIN – in simple words

1. One switch is one network. Every device on it can reach every other device, and every broadcast reaches everyone.
2. Often that is not what you want. An office may want staff computers, guest WiFi, security cameras and phones kept apart from each other.

3. You could buy four separate switches and four sets of cables. That is expensive and inflexible.
4. Instead you tell one switch to pretend to be four switches. Each pretend switch is called a **VLAN**, a virtual local area network.
5. Ports are assigned to VLANs. A port in VLAN 10 cannot reach a port in VLAN 20, even though they are in the same physical box.
6. To get between VLANs you need a router. That is the point: it gives you a place to apply rules.
7. Now the second idea. A network with two paths between the same two switches has a **loop**.
8. Loops are wonderful for reliability and catastrophic without protection, because a broadcast frame goes round the loop forever.
9. There is no field in an Ethernet frame that counts hops, so nothing ever stops it. It multiplies at every switch and the network dies in seconds.
10. **Spanning tree** is the protocol that finds loops and switches off just enough links to break them, while keeping those links ready as backups.

■ 23.9.2 PLAIN – a picture in your head

1. Picture an office floor with one big open room.
2. VLANs are movable partition walls. The floor is unchanged, but people in one section cannot walk into another without going through a door with a guard on it. The guard is the router.
3. To move a person from one section to another you do not rewire anything. You change which section their desk is assigned to. That is reassigning a port to a VLAN.
4. Now picture the announcement system. A message sent in section A is repeated by every loud-speaker in section A only.
5. Now the loop. Imagine two corridors joining the same two rooms, and a rule that any announcement heard must be repeated into every other corridor.
6. Someone makes one announcement. It goes round both corridors, arrives back, gets repeated, and now there are two. Then four. Then eight.
7. Within seconds nobody can hear anything else. That is a **broadcast storm**.
8. Spanning tree is a supervisor who walks the building once, works out the corridors, and locks exactly one door so no circular route exists.

Where this comparison breaks: a locked door stays locked. Spanning tree keeps watching, and if the open corridor collapses it unlocks the spare door within seconds. Also, the supervisor is not a person: every switch runs the algorithm independently and they reach the same answer by exchanging small messages, with no central authority. That distributed agreement is the clever part.

■ 23.9.3 PLAIN – a worked example

1. A small office, one 24-port switch, three VLANs.

VLAN	Name	Ports
10	Staff	1 to 12
20	Guest WiFi	13 to 18
30	Cameras	19 to 23
all	Uplink to router	24

2. Ports 1 to 23 are **access ports**. Each belongs to exactly one VLAN. The device plugged in knows nothing about VLANs at all.
3. Port 24 is a **trunk port**. It carries all three VLANs to the router over one cable.

4. For that to work, frames leaving port 24 must say which VLAN they belong to. So a 4-byte tag is inserted into each frame.
5. A camera on port 19 sends a frame. It is untagged, a plain 64-byte frame.
6. The switch forwards it towards port 24 and inserts a tag saying VLAN 30. The frame is now 68 bytes.
7. The router receives it, sees VLAN 30, and applies the camera policy: allowed to reach the recorder, not allowed to reach the internet.
8. A guest laptop on port 15 broadcasts. That broadcast reaches ports 13 to 18 and the trunk, and never touches ports 1 to 12. The staff never see it.
9. Without VLANs, one broadcast domain of 23 ports. With VLANs, three broadcast domains sharing one box.

■ 23.9.4 PLAIN – what is really happening inside

1. The tag is four bytes inserted immediately after the source MAC address.
2. The first two bytes are always 0x8100. That is the EtherType that says “a VLAN tag follows”.
3. The next two bytes hold three things: three bits of priority, one bit that is almost always zero, and twelve bits of VLAN identifier.
4. Twelve bits gives 4096 values, of which 0 and 4095 are reserved, so 4094 usable VLANs.
5. Because the tag is inserted, the switch must recompute the frame check sequence. Adding a tag changes the frame, so the old CRC is no longer valid.
6. An access port strips the tag on the way out, so end devices never see it.
7. Now spanning tree. Every switch sends small messages called BPDUs out of every port every two seconds.
8. Each BPDU carries the sender’s identity and its cost to reach the switch it currently believes is the root of the tree.
9. Through this gossip, all switches agree on one **root bridge**: the switch with the numerically lowest bridge identifier.
10. Every other switch works out its own cheapest path to the root, keeps that port forwarding, and blocks any other port that would create a second path.
11. A blocked port still listens to BPDUs. If they stop arriving, the topology has changed, and the blocked port is brought back into service.

■ 23.9.5 TECHNICAL – the engineer’s version

1. VLAN tagging is IEEE 802.1Q, first published in 1998 and now folded into 802.1Q-2022. The tag structure:

Part	Bits	Meaning
TPID	16	Always 0x8100
PCP	3	Priority, 802.1p
DEI	1	Drop eligible indicator
VID	12	VLAN ID, 1 to 4094

2. A tagged frame’s maximum size is 1522 octets instead of 1518. Older equipment that counted 1518 as an absolute maximum reported these as “baby giants”.
3. The **native VLAN** on a trunk is carried untagged, by convention VLAN 1. This is the basis of VLAN hopping via double tagging, so best practice is to set the native VLAN to an unused ID and tag everything.

4. 802.1ad (2005), “QinQ” or provider bridging, stacks a second tag with TPID 0x88A8 so a carrier can transport a customer’s whole VLAN space.
5. Spanning Tree Protocol was invented in 1985 at Digital Equipment Corporation by Radia Perlman, who described it in her paper “An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN” and in the poem “Algorhyme”. IEEE published it as 802.1D in 1990.
6. Classic STP port states are blocking, listening, learning, forwarding and disabled. Convergence takes 30 to 50 seconds, because listening and learning each last one forward delay of 15 seconds by default.
7. Default timers: hello 2 seconds, forward delay 15 seconds, max age 20 seconds.
8. Rapid Spanning Tree Protocol, IEEE 802.1w (2001), cut the states to discarding, learning and forwarding, and converges within three hello times or within milliseconds of a detected physical link failure. It was merged into 802.1D-2004.
9. Multiple Spanning Tree Protocol, IEEE 802.1s (2002), runs one tree per group of VLANs so different VLANs can use different physical links. It was merged into 802.1Q-2005.
10. Shortest Path Bridging, IEEE 802.1aq (2012), and TRILL, RFC 6325 (2011), replace the tree with a link-state protocol so all links can carry traffic. In practice most large data centres skipped both and moved to layer 3 fabrics with ECMP, or to VXLAN overlays per RFC 7348 (2014).
11. Why a loop is catastrophic, precisely: the Ethernet header has no TTL or hop count field. An IP packet has a TTL that a router decrements, so an IP routing loop self-terminates after at most 255 hops. A layer 2 loop has no such stop condition. Broadcast and unknown-unicast frames replicate at every switch on every pass, so traffic grows geometrically until link utilization reaches 100 percent, typically within one to two seconds.
12. There is a second failure alongside the storm: **MAC table instability**. The same source address arrives on two ports alternately, so every switch rewrites that entry thousands of times a second, and legitimate unicast traffic gets flooded as well.
13. Protective features, all vendor implementation details rather than standards: BPDU guard (shut a port that receives a BPDU when it should not), root guard, loop guard, and storm control thresholds.
14. Home routers usually run no spanning tree at all. Plugging one cheap unmanaged switch into another twice, by accident, will take down a home or small office network completely, and this is a genuinely common fault.

■ 23.9.6 WORDS – remember these

1. VLAN — one switch pretending to be several — a virtual LAN, an isolated broadcast domain identified by a 12-bit VID under IEEE 802.1Q.
2. Access port — a port for an ordinary device — an untagged port belonging to exactly one VLAN.
3. Trunk port — a port carrying many VLANs at once — a tagged port carrying 802.1Q-tagged frames for multiple VIDs.
4. Tag — the four bytes saying which VLAN — TPID 0x8100 plus PCP, DEI and VID, inserted after the source MAC.
5. Loop — two paths between the same switches — a cycle in the layer 2 topology with no hop count to stop frames.
6. Broadcast storm — the network drowning in copies of one frame — geometric replication of flooded frames around a loop.
7. Spanning tree — the protocol that breaks loops safely — IEEE 802.1D and its successors, electing a root bridge and blocking redundant ports.
8. BPDU — the small message switches gossip with — bridge protocol data unit, sent every hello

interval to build and maintain the tree.

9. Root bridge — the switch everything measures distance from — the bridge with the lowest bridge ID, elected by all participants.

23.10 ARP: from an IP address to a MAC address

■ 23.10.1 PLAIN – in simple words

1. Here is the problem this section solves, and it is the most important idea in the chapter.
2. Software works with IP addresses. The reader's laptop wants to send something to 192.168.0.1, the router.
3. But the network card cannot send to an IP address. It can only send to a MAC address, the hardware address from section 23.7.
4. Nothing connects the two. An IP address does not contain a MAC address and a MAC address does not contain an IP address.
5. So the laptop has to ask. It shouts to the whole local network: "whoever has 192.168.0.1, tell me your hardware address".
6. The router hears its own IP address in the question and answers directly: "192.168.0.1 is at this MAC address".
7. That question-and-answer is **ARP**, the address resolution protocol.
8. The laptop writes the answer down in a small table so it does not have to ask again for every packet.
9. That table is the **ARP cache**, and entries in it expire after a few minutes.
10. ARP is only for the local network. To reach something far away, the laptop ARP's for the router and sends everything there. The router deals with the rest.

■ 23.10.2 PLAIN – a picture in your head

1. You are in a large office and you have a note to hand to "the head of accounts". You do not know who that is or where they sit.
2. You stand up and call out to the whole floor: "who is the head of accounts?"
3. Everyone hears. Everyone except one person ignores you.
4. One person calls back: "that is me, third desk by the window."
5. You write that down on a sticky note on your monitor, and from then on you walk straight there.
6. A few weeks later the sticky note is out of date, because someone else took the job. So you throw the note away periodically and ask again.
7. Notice the danger. When you called out, anyone could have answered. Nobody checks credentials. The first confident reply wins.

Where this comparison breaks: in an office, a wrong answer would be noticed socially within a day. On a network, a false ARP reply is silently believed forever, or at least until the entry ages out, and the victim has no way at all to tell a true reply from a false one. ARP has no authentication of any kind. This is not a bug that was later fixed; it is the design, and the design is from 1982.

■ 23.10.3 PLAIN – a worked example

1. This is the reader's own network. The laptop is at 192.168.0.107 and the router is at 192.168.0.1. MAC addresses are illustrative, because the session record does not include them.

Step 1: the request, broadcast to everyone

Ethernet dest : FF:FF:FF:FF:FF:FF (everyone)


```

Ethernet src : 3C:22:FB:8A:41:D7    (the laptop)
EtherType    : 0x0806              (this is ARP)
Opcode       : 1                   (request)
Sender MAC   : 3C:22:FB:8A:41:D7
Sender IP    : 192.168.0.107
Target MAC   : 00:00:00:00:00:00    (unknown, that is
                                   the whole question)
Target IP    : 192.168.0.1

```

Step 2: the reply, sent to one machine only

```

Ethernet dest : 3C:22:FB:8A:41:D7    (back to the laptop)
Ethernet src  : 5C:A6:E6:11:22:33    (the router)
EtherType     : 0x0806
Opcode        : 2                   (reply)
Sender MAC    : 5C:A6:E6:11:22:33
Sender IP     : 192.168.0.1
Target MAC    : 3C:22:FB:8A:41:D7
Target IP     : 192.168.0.107

```

2. Notice the request is a broadcast and the reply is a unicast. The question must reach everyone. The answer only needs to reach the asker.
3. Notice also that the router learns the laptop's mapping for free, from the sender fields of the request. Both sides end up with a cache entry after one exchange.
4. Now the laptop can build a real frame. To send an IP packet to GitHub at 20.207.73.82, it puts:
 1. Destination MAC: 5C:A6:E6:11:22:33, the router.
 2. Source MAC: 3C:22:FB:8A:41:D7, itself.
 3. Destination IP: 20.207.73.82, GitHub.
 4. Source IP: 192.168.0.107, itself.
5. Read that again, because it is the sentence that confuses everyone. The MAC addresses are local and change at every hop. The IP addresses are end to end and do not change.
6. Here is the reader's ARP cache, read on macOS:

```

$ arp -a
? (192.168.0.1) at 5c:a6:e6:11:22:33 on en0 ifscope [ethernet]
? (192.168.0.107) at 3c:22:fb:8a:41:d7 on en0 [ethernet]
? (192.168.0.255) at ff:ff:ff:ff:ff:ff on en0 ifscope [ethernet]

```

7. The question marks are hostnames macOS could not resolve. en0 is the interface. ifscope means the entry is bound to that specific interface.

■ 23.10.4 PLAIN – what is really happening inside

1. Before sending any IP packet, the machine asks one question: is this destination on my own local network, or somewhere else?
2. It answers that by comparing the destination IP against its own IP and subnet mask. The next chapter covers exactly how that comparison works.
3. If the destination is local, it ARPs for the destination itself.
4. If the destination is not local, it does not ARP for the destination at all. It ARPs for the **default gateway**, which here is 192.168.0.1.
5. This is why every machine on the internet has a cache entry for its router and for nothing else outside its own street.
6. Packets waiting for an ARP reply sit in a small queue. If no reply comes, the request is retried a few times and then the packets are dropped and an error is reported to the application.

7. A **gratuitous ARP** is an unsolicited announcement: a machine broadcasts an ARP for its own address, which nobody asked for.
8. It has two honest uses. It tells everyone to update their cache when an address moves to new hardware, and it detects duplicate addresses at startup.
9. It also has a dishonest use. A machine can send gratuitous replies claiming to own the router's IP address, and every machine on the network will believe it.
10. That is **ARP spoofing**. The attacker now receives traffic meant for the router, can read it, change it, and pass it on. It is a textbook man-in-the-middle attack, and it requires only being on the same network.

■ 23.10.5 TECHNICAL – the engineer's version

1. ARP is defined in RFC 826, published November 1982, by David C. Plummer. It is Internet Standard STD 37 and it has never been revised.
2. It rides directly on Ethernet with EtherType 0x0806. It is not carried in IP, so it is neither TCP nor UDP and has no port number.
3. The packet is 28 bytes for the IPv4-over-Ethernet case:

Field	Bytes	Value for IPv4
HTYPE hardware type	2	1 for Ethernet
PTYPE protocol type	2	0x0800 for IPv4
HLEN hardware length	1	6
PLEN protocol length	1	4
OPER operation	2	1 request, 2 reply
SHA sender hardware	6	Sender MAC
SPA sender protocol	4	Sender IP
THA target hardware	6	Target MAC
TPA target protocol	4	Target IP

4. 28 bytes is below the 46-byte minimum payload, so 18 bytes of padding are always added. Some old implementations leaked memory contents in that padding; this was named "Etherleak" and assigned CVE-2003-0001.
5. Cache timeouts are implementation details, not standards. On macOS and other BSD-derived systems the default is 1200 seconds, tunable with `sysctl net.link.ether.inet.arp_keep` on some versions. On Linux the relevant knobs are `net.ipv4.neigh.default.base_reachable_time_ms`, default 30000, and `gc_stale_time`, default 60 seconds. Cisco IOS defaults to 14400 seconds, four hours, which is far longer than most people expect.
6. Variants worth knowing by name: **proxy ARP** (RFC 1027, 1987), where a router answers on behalf of a host on another segment; **ARP probe** and **ARP announcement** (RFC 5227, 2008), used for duplicate address detection; and **RARP** (RFC 903, 1984), the obsolete reverse lookup replaced by BOOTP and DHCP.
7. IPv6 does not use ARP at all. It uses Neighbor Discovery Protocol, RFC 4861 (2007), which runs over ICMPv6 and uses multicast rather than broadcast. The reader's session reported IPv6: (none), so no NDP was in play in this case.
8. ARP spoofing defences: dynamic ARP inspection on managed switches, which validates ARP packets against the DHCP snooping binding table; static ARP entries for critical addresses; 802.1X port authentication; and simply encrypting everything so that intercepting the traffic gains the attacker little. TLS, covered later in Part F, is the real answer.

9. Commands to observe and manipulate ARP on macOS:

```
# show the whole table
arp -a

# show one entry, numeric only
arp -n 192.168.0.1

# delete one entry, forcing a fresh ARP
sudo arp -d 192.168.0.1

# delete every entry
sudo arp -a -d

# add a permanent static entry
sudo arp -S 192.168.0.1 5c:a6:e6:11:22:33

# watch the exchange live
sudo tcpdump -i en0 -e -n arp
```

10. On Linux the modern equivalents are `ip neigh show`, `ip neigh flush all` and `ip neigh replace`. The old `arp` command still works but is deprecated in favour of `ip`.
11. Diagnostic value: if `arp -a` shows no entry for your gateway, and deleting and re-pinging does not create one, the problem is beneath IP entirely. Your machine is not reaching the router at layer 2. That is a different fault from the reader's, whose ARP was clearly fine, since traffic reached hop 1 and twelve hops beyond it.
12. The honest version of "ARP is a local protocol": ARP messages are never forwarded by routers, so an ARP table only ever contains addresses on your own link. Any tool that appears to show you a remote machine's MAC address is showing you something else, usually a router's.

■ 23.10.6 WORDS - remember these

1. ARP — asking who owns an IP address on this network — the Address Resolution Protocol, RFC 826, EtherType 0x0806.
2. ARP request — the broadcast question — opcode 1, sent to FF:FF:FF:FF:FF:FF with the target hardware address zeroed.
3. ARP reply — the direct answer — opcode 2, unicast back to the requester.
4. ARP cache — the notes you keep so you need not ask again — the neighbour table mapping IP to MAC, with a per-entry ageing timer.
5. Default gateway — the router you send everything distant to — the next hop for destinations outside your own subnet.
6. Gratuitous ARP — announcing yourself without being asked — an unsolicited ARP with sender and target protocol addresses equal.
7. ARP spoofing — lying about who owns an address — forging ARP replies to redirect traffic, enabling man-in-the-middle interception.
8. Proxy ARP — a router answering on someone else's behalf — RFC 1027 behaviour where a gateway replies for hosts it can reach.
9. Neighbor Discovery — the IPv6 replacement for ARP — RFC 4861, running over ICMPv6 with multicast solicitations.

23.11 DHCP: how the laptop got an address

■ 23.11.1 PLAIN – in simple words

1. Nobody configured the reader's laptop with an IP address. It joined the WiFi and simply had one, in the 192.168.0.x range.
2. Something handed it that address. That something is a **DHCP server**, and in a home it lives inside the router at 192.168.0.1.
3. DHCP stands for dynamic host configuration protocol. It automates the whole job of joining a network.
4. There is a chicken-and-egg problem. To ask for an address, you must send a message. To send a message, you normally need an address.
5. DHCP solves it by letting the client send from the address 0.0.0.0, meaning "no address yet", to the address 255.255.255.255, meaning "everyone here".
6. The conversation takes four messages, and they have a memorable name: DORA.
7. **Discover**: the client shouts "is there a DHCP server out there?"
8. **Offer**: a server replies "yes, and you may have this address".
9. **Request**: the client says "I accept that address" — and it says it out loud so any other servers know they were not chosen.
10. **Acknowledge**: the server confirms and records the assignment.
11. The address is not given away. It is lent, for a fixed period called a **lease**, and the client must renew before it runs out.

■ 23.11.2 PLAIN – a picture in your head

1. Picture arriving at a large hotel that has no reception desk you can see.
2. You stand in the lobby and call out: "does anyone here give out rooms?" That is Discover.
3. A clerk calls back: "yes, room 107 is free, it comes with a floor plan, the number for the concierge, and it is yours for one day." That is Offer.
4. You call back, loudly enough for the whole lobby: "I will take room 107 from that clerk." That is Request. The loudness matters, because a second clerk who also had a room ready now knows to put it back.
5. The clerk writes your name in the register and hands you the key. That is Acknowledge.
6. Halfway through the day, you go and ask to extend. Usually you get it. If the clerk has gone, you keep trying, and when the time is nearly up you start asking any clerk at all.
7. If nobody at all answers your first call, you do not stand in the lobby forever. You pick a corner of the lobby to sit in and hope somebody notices. That corner is the 169.254.x.x address, and it means you never got a room.

Where this comparison breaks: in a hotel, a second clerk handing out the same room would be caught by the register. On a network, two DHCP servers on the same segment genuinely can hand out conflicting addresses, and the usual cause is someone plugging a home router into an office network by mistake. There is no central register. There is only a convention that clients take the first offer and servers ping an address before offering it.

■ 23.11.3 PLAIN – a worked example

1. This is the reader's laptop joining their home network. The address 192.168.0.107 is illustrative; the session record confirms only that it was in the 192.168.0.x range.

```
1. DISCOVER    client -> broadcast
   Ethernet dst : FF:FF:FF:FF:FF:FF
```

```

IP src      : 0.0.0.0
IP dst      : 255.255.255.255
UDP src port : 68
UDP dst port : 67
Contains    : client MAC, requested options

2. OFFER      server -> client
IP src      : 192.168.0.1
IP dst      : 255.255.255.255 (or the offered IP)
UDP src port : 67
UDP dst port : 68
Contains    : yiaddr 192.168.0.107, mask, gateway,
              DNS servers, lease time

3. REQUEST    client -> broadcast
IP src      : 0.0.0.0
IP dst      : 255.255.255.255
UDP 68 -> 67
Contains    : "I want 192.168.0.107 from 192.168.0.1"

4. ACK        server -> client
IP src      : 192.168.0.1
UDP 67 -> 68
Contains    : final confirmed values and lease time

```

2. Note the ports. The server always listens on 67. The client always listens on 68. They are fixed, and a client that used a random source port would never hear the reply.
3. Note that Request is broadcast even though the client now knows the server's address. That is deliberate, so other servers can withdraw their offers.
4. Suppose the lease is 86400 seconds, one day. Two timers are set from it.
5. T1, the renewal timer, fires at 50 percent: 43200 seconds, twelve hours in. The client asks its own server directly, by unicast, to extend.
6. T2, the rebinding timer, fires at 87.5 percent: 75600 seconds, twenty-one hours in. If the original server never answered, the client now broadcasts to any server at all.
7. If nothing answers by 86400 seconds, the client must stop using the address entirely and start again from Discover.
8. The reader can see all of this on macOS:

```

# current lease and everything the server sent
ipconfig getpacket en0

# just the router and DNS the server supplied
ipconfig getoption en0 router
ipconfig getoption en0 domain_name_server

# force a fresh lease
sudo ipconfig set en0 DHCP

# watch the whole DORA exchange live
sudo tcpdump -i en0 -n port 67 or port 68

```

■ 23.11.4 PLAIN – what is really happening inside

1. A DHCP server does not just give an address. It gives a whole configuration, and this is the part people forget.
2. The standard things it hands out are the IP address, the subnet mask, the default gateway, the DNS servers, the domain name and the lease time.
3. It can also hand out time servers, TFTP boot servers, and dozens of other things, each identified by a number.
4. Here is a fact from the reader's session worth pausing on. Their DNS resolver was 1.1.1.1, the Cloudflare public resolver, not the router.
5. That means either the DHCP server was configured to hand out 1.1.1.1, or somebody overrode DHCP's suggestion on the laptop itself. Both are common. The session data does not say which.
6. Now the failure case, which is the diagnostically useful one.
7. If no DHCP server answers, the operating system assigns itself an address in the range 169.254.x.x, picked at random and checked for clashes with ARP.
8. That is a **link-local** address. Two machines on the same cable with 169.254 addresses can talk to each other, and to nothing else at all.
9. So seeing 169.254.x.x on your machine is not a random fault. It is a specific message, and the message is: "I never heard from a DHCP server".
10. That single observation narrows the fault enormously. The cable or radio link is probably up, since the machine is on a network. But nothing answered.
11. Common causes: the router's DHCP service is off or crashed, the pool is exhausted, the machine is on the wrong VLAN, or the DHCP relay is missing on a routed segment.
12. The reader never saw 169.254. Their laptop held a valid 192.168.0.x address throughout, and reached twelve hops out. So DHCP was not the fault here. This is what ruling something out looks like.

■ 23.11.5 TECHNICAL – the engineer's version

1. Lineage: RARP, RFC 903 (June 1984). BOOTP, RFC 951 (September 1985). DHCP, RFC 1531 (October 1993), then RFC 1541, then the current RFC 2131 (March 1997). Options are in RFC 2132 (March 1997). DHCPv6 is RFC 3315 (2003), superseded by RFC 8415 (2018).
2. DHCP is backward compatible with BOOTP and reuses its packet format, which is why the fields still have BOOTP names such as yiaddr (your IP address), siaddr (next server), giaddr (relay agent address) and chaddr (client hardware address).
3. Message types are carried in option 53. The eight defined values are DISCOVER 1, OFFER 2, REQUEST 3, DECLINE 4, ACK 5, NAK 6, RELEASE 7 and INFORM 8.
4. Options seen in nearly every real exchange:

Option	Name	Typical value
1	Subnet mask	255.255.255.0
3	Router	192.168.0.1
6	DNS servers	1.1.1.1
15	Domain name	localdomain
42	NTP servers	pool.ntp.org address
51	Lease time	86400 seconds
53	Message type	1 to 8
55	Parameter request	list of wanted options

Option	Name	Typical value

5. Timers per RFC 2131: T1 defaults to 0.5 times the lease duration, T2 to 0.875 times it. Servers may override both with options 58 and 59. At T1 the client unicasts a REQUEST in RENEWING state. At T2 it broadcasts in REBINDING state.
6. A server may reply DHCPNAK if the requested address is no longer valid, for example after a laptop moves between networks and asks for its old address. The client must then restart from DISCOVER.
7. DHCP is UDP, not TCP. Ports 67 (server, “bootps”) and 68 (client, “bootpc”) are fixed by IANA. Both are below 1024, so a DHCP client needs privilege to bind port 68.
8. Broadcasts do not cross routers, so on any network larger than one segment a **DHCP relay agent**, also called an IP helper, forwards DISCOVER messages as unicast to a central server and records its own address in `giaddr` so the server knows which pool to use. RFC 3046 (2001) adds option 82, the relay agent information option, which carries the physical port the request came from.
9. Static assignment versus reservation, and the distinction matters operationally:
 1. A **static** address is configured on the client itself. The DHCP server does not know about it and may hand the same address to someone else.
 2. A **reservation**, or static lease, is configured on the server and keyed to the client’s MAC address. The client still runs normal DORA and still gets the gateway, DNS and domain automatically.
 3. Reservations are almost always the right choice, because configuration stays in one place. Their weakness is MAC randomization, which changes the key the reservation is matched on.
10. IPv4 link-local is 169.254.0.0/16, defined in RFC 3927 (May 2005). The first and last 256 addresses, 169.254.0.0/24 and 169.254.255.0/24, are reserved. Microsoft calls the mechanism APIPA. The IPv6 equivalent is `fe80::/10` per RFC 4291 (February 2006), and it is always present, not a fallback.
11. Security notes: DHCP has no authentication. A rogue server can hand out its own address as the gateway and intercept everything. The defence on managed switches is **DHCP snooping**, which permits server replies only on ports explicitly marked trusted, and builds a binding table that dynamic ARP inspection then uses.
12. Observation on Linux: `dhclient -v`, `networkctl status`, `journalctl -u systemd-networkd`, or the lease files under `/var/lib/dhcp/`. On macOS the lease lives under `/var/db/dhcpclient/leases/`.

■ 23.11.6 WORDS – remember these

1. DHCP — the thing that gives your device an address automatically — Dynamic Host Configuration Protocol, RFC 2131, over UDP ports 67 and 68.
2. DORA — the four-step exchange — Discover, Offer, Request, Acknowledge.
3. Lease — the address is lent, not given — a time-limited allocation with renewal at T1 and rebinding at T2.
4. T1 and T2 — when to ask for more time — 50 percent and 87.5 percent of the lease duration by default.
5. Default gateway — the router address DHCP hands you — option 3, the next hop for non-local destinations.
6. Reservation — always giving the same device the same address — a server-side static lease keyed on the client MAC address.

7. Relay agent — how DHCP crosses a router — an IP helper that unicasts client broadcasts to a central server and fills in giaddr.
8. Link-local address — the address you give yourself when nothing answers — 169.254.0.0/16 per RFC 3927, reachable only on the same link.
9. DHCP snooping — the switch refusing to believe rogue servers — a security feature permitting server messages only on trusted ports.

23.12 The network interface card

■ 23.12.1 PLAIN - in simple words

1. The **network interface card**, or NIC, is the piece of hardware that turns data in memory into signals on the medium, and back.
2. In a laptop it is not a card at all. It is a chip soldered to the board, and on WiFi it is a radio chip with an antenna.
3. It has four jobs. Frame the data, put the signal on the wire, filter incoming frames by address, and check the FCS.
4. It cannot do any of this alone. It needs a **driver**, which is the piece of operating system software that knows how to talk to that exact chip.
5. The card and the operating system share a set of memory slots called **ring buffers**. The name is because the slots are used in a circle, over and over.
6. There is one ring for outgoing frames and one for incoming.
7. To send, the operating system writes a frame into a free slot and tells the card. The card copies it out and marks the slot free again.
8. To receive, the card writes into a free slot and raises an **interrupt**, a hardware signal that tells the CPU to stop what it is doing and look.
9. At high speed, interrupting for every single frame would consume the whole CPU. So the card waits a little and interrupts once for a batch instead.
10. Normally the card throws away frames not addressed to it. If you turn that filtering off, it passes up everything it hears. That is how packet capture works.

■ 23.12.2 PLAIN - a picture in your head

1. Picture a busy loading bay behind a shop, with a row of numbered bays around a circular driveway.
2. The staff inside load goods into a free bay and raise a small flag. The driver outside takes whatever bay has a flag, loads the lorry, and lowers the flag.
3. Deliveries arriving work the same way in reverse. The driver fills a free bay and rings a bell so the staff come out.
4. If lorries arrive every few seconds, ringing the bell each time means the staff never get anything else done.
5. So the driver waits until three lorries have arrived, or ten seconds have passed, then rings once. That is interrupt coalescing.
6. If all the bays are full because the staff are busy, arriving lorries are turned away. Those are dropped packets, and they are counted.

Where this comparison breaks: the flags in a real NIC are not checked by a person walking out to look. The card writes directly into main memory over the system bus, with no CPU involvement at all, using DMA. The CPU only finds out afterwards. Also, a lorry turned away can come back

tomorrow. A dropped frame is gone, and it is entirely up to a higher-layer protocol to notice and resend it.

■ 23.12.3 PLAIN – a worked example

1. Here is a real captured frame, decoded field by field. It is the first frame of an attempt to reach GitHub from the reader's laptop. MAC addresses are illustrative.

Raw bytes as `tcpdump -xx` would show them:

```
5c a6 e6 11 22 33 3c 22 fb 8a 41 d7 08 00 45 00
00 3c 1c 46 40 00 40 06 xx xx c0 a8 00 6b 14 cf
49 52 e3 f4 01 bb ...
```

Decoded:

```
5c a6 e6 11 22 33 Destination MAC, the router
3c 22 fb 8a 41 d7 Source MAC, the laptop
08 00             EtherType 0x0800, so IPv4
45               IP version 4, header length 5 words
00               Differentiated services, none set
00 3c            Total length 60 bytes
1c 46            Identification
40 00            Flags: Don't Fragment set
40               TTL 64, the usual Linux/macOS start
06               Protocol 6, which is TCP
xx xx            Header checksum
c0 a8 00 6b      Source IP 192.168.0.107
14 cf 49 52      Destination IP 20.207.73.82
e3 f4            Source port 58356
01 bb            Destination port 443
```

2. 14 cf 49 52 in decimal is 20, 207, 73, 82. That is exactly the address the reader's `curl` printed: Trying 20.207.73.82:443....
3. 01 bb in decimal is 443, the HTTPS port.
4. Notice the destination MAC is the router, not GitHub. GitHub is not on this network, so the frame is addressed locally and routed onwards.
5. Notice the TTL is 64. Every router that forwards this packet subtracts one. That mechanism is what makes `traceroute` possible, and it gets a chapter of its own later in Part F.
6. Everything from 45 onwards is the IP packet, which is the next chapter. Everything before it is this chapter.
7. In the reader's outage, frames exactly like this one went out and nothing ever came back. The frame was correct. The path was not.

■ 23.12.4 PLAIN – what is really happening inside

1. A modern NIC does far more than move bytes. It does work that the CPU used to do, and this is called **offload**.
2. **Checksum offload** means the card computes the IP and TCP checksums in hardware as the frame goes out, and verifies them on the way in.
3. This has a confusing side effect for anyone capturing traffic. A capture taken on the sending machine shows outgoing checksums as invalid, because the card has not filled them in yet. That is normal, not a fault.
4. **Segmentation offload** means the operating system hands the card one huge block, say 64 kilobytes, and the card chops it into 1500-byte frames itself, generating all the headers.

5. That turns forty-odd trips through the network stack into one, which is a large saving at gigabit speeds and above.
6. The same trick in reverse merges many arriving small frames into one big one before handing it up. A capture on such a machine will show impossible 64-kilobyte “frames”. They never existed on the wire.
7. **Promiscuous mode** turns off the address filter. The card passes up every frame it receives, regardless of destination.
8. On a switched network that gains you less than people expect, because the switch already only sends you frames for you, plus broadcasts and floods.
9. To capture everything on a switched network you need a **mirror port**, where the switch is configured to copy all traffic from certain ports to yours.
10. On WiFi, promiscuous mode is not enough either. You need **monitor mode**, which captures raw radio frames including management frames, and on many drivers it means disconnecting from the network to do it.

■ 23.12.5 TECHNICAL – the engineer’s version

1. The descriptor rings are arrays of small structures in host memory. Each descriptor holds a physical buffer address, a length and status bits. The driver owns some entries, the card owns others, and ownership is passed by writing a status bit and advancing a tail pointer.
2. Data movement is DMA, direct memory access, so the frame body never passes through the CPU. On systems with an IOMMU the addresses are translated and bounded, which is what stops a compromised NIC from reading arbitrary memory.
3. Interrupt coalescing is configured by two thresholds: a packet count and a time. Tuned with `ethtool -C eth0 rx-usecs 50 rx-frames 32` on Linux. Lower values reduce latency and raise CPU usage; higher values do the opposite.
4. Linux replaced pure interrupt-driven receive with **NAPI** in kernel 2.4.20 (2002). On the first interrupt the driver disables further receive interrupts and switches to polling, then re-enables them when the ring drains. This is what stops a machine from livelocking under a packet flood.
5. Offload features, with their Linux names:

Feature	Linux name	What it does
RX checksum	rx-checksumming	Verify in hardware
TX checksum	tx-checksumming	Compute in hardware
TCP segmentation	tso	Card splits large sends
Generic segmentation	gso	Kernel splits late
Large receive	lro	Card merges arrivals
Generic receive	gro	Kernel merges arrivals
Receive side scaling	rss	Spread flows over queues

6. Receive side scaling hashes each flow’s addresses and ports to choose one of several receive queues, each bound to a different CPU core. Without it a single core would have to handle a whole 10 or 100 Gbit/s interface.
7. Modern high-speed NICs support SR-IOV, presenting many virtual functions to virtual machines directly, and kernel-bypass frameworks such as DPDK and AF_XDP that let user-space code poll the ring itself. That is how software routers reach line rate.
8. Packet capture on Unix goes through **BPF**, the Berkeley Packet Filter, described by Steven McCanne and Van Jacobson in the 1993 USENIX paper “The BSD Packet Filter: A New Architecture for

User-level Packet Capture”. The filter expression is compiled to bytecode and run in the kernel, so unwanted packets are discarded before they are copied to user space.

9. Linux later generalized this into eBPF, which now runs far more than packet filters, and into AF_PACKET with a memory-mapped ring for capture.
10. tcpdump was written at Lawrence Berkeley Laboratory starting in 1988 by Van Jacobson, Craig Leres and Steven McCanne. libpcap, the capture library underneath it, came from the same group and is what almost every capture tool uses today.
11. Wireshark began as Ethereal, written by Gerald Combs in 1998, and was renamed Wireshark in 2006 over a trademark issue. Its command-line form is tshark.
12. Useful capture commands:

```
# macOS: capture ARP and DHCP with Ethernet headers shown
sudo tcpdump -i en0 -e -n 'arp or port 67 or port 68'

# capture to a file for Wireshark, full frames
sudo tcpdump -i en0 -s 0 -w capture.pcap

# only traffic to the address curl was trying
sudo tcpdump -i en0 -n host 20.207.73.82

# show raw bytes including the Ethernet header
sudo tcpdump -i en0 -xx -c 1
```

13. On macOS, tcpdump on a WiFi interface will not show other stations’ traffic even with -p disabled, because the radio decrypts only frames for you. Apple’s airport utility historically enabled monitor mode, but it was removed in macOS 14.4 (2024); the supported route now is the Wireless Diagnostics application. That is an implementation detail and it has changed more than once.
14. Interface statistics worth reading before anything else: netstat -i on macOS, ip -s link on Linux, and ethtool -S eth0 for the full vendor-specific counter set. Non-zero Ierrs, 0errs or rx_crc_errors point at a physical problem: a bad cable, a dirty optic, or a duplex mismatch.
15. The honest version: a capture taken on the machine under test is not the truth about the wire. Offloads reshape it, the driver may drop frames before the filter, and the timestamps come from the host clock unless the NIC does hardware timestamping. For a definitive answer you capture from a mirror port or a passive tap on another machine.

■ 23.12.6 WORDS – remember these

1. NIC — the hardware that connects you to the network — network interface controller, implementing the physical and MAC layers.
2. Driver — the software that knows this exact chip — the kernel module managing descriptor rings, interrupts and offload configuration.
3. Ring buffer — a circle of memory slots shared with the card — a descriptor ring with head and tail pointers and DMA buffer addresses.
4. DMA — the card writing to memory by itself — direct memory access, moving frame data without CPU involvement.
5. Interrupt coalescing — batching the “look at me” signals — delaying interrupts by packet count or time to trade latency for CPU cost.
6. Checksum offload — the card doing the arithmetic — hardware computation and verification of IP, TCP and UDP checksums.
7. Segmentation offload — handing the card one big block — TSO or GSO, splitting a large buffer into MTU-sized frames in hardware or late in the stack.

8. Promiscuous mode — accepting frames not addressed to you — disabling the MAC destination filter for capture.
9. Mirror port — a switch port that copies other ports' traffic — SPAN or port mirroring, required to capture on a switched network.
10. BPF — the filter that decides what to capture — Berkeley Packet Filter bytecode executed in the kernel, used by libpcap and tcpdump.

23.13 Putting the local network together

■ 23.13.1 PLAIN - in simple words

1. We now have every piece. Let us watch the reader's laptop join the network from cold, in order, naming everything.
2. It starts with radio, because the reader is on WiFi. The card scans for networks, picks one, and authenticates and associates with the access point.
3. From that moment the laptop is on a network, but it has no address and knows nothing about it.
4. So it asks. It broadcasts a DHCP Discover from 0.0.0.0 to 255.255.255.255.
5. The router answers with an Offer containing an address in the 192.168.0.x range, the mask, the gateway 192.168.0.1, and the DNS servers.
6. The laptop broadcasts a Request accepting it, and the router sends an Acknowledge. The laptop now has an address and a lease.
7. Before it uses that address, it usually checks nobody else has it, by sending an ARP probe for its own address and listening for a reply.
8. Now it wants to reach something. It compares the destination against its own subnet and finds the destination is elsewhere.
9. So it needs the router's hardware address. It broadcasts an ARP request for 192.168.0.1 and gets a reply.
10. Only now can the first real frame be built and sent. Everything up to this point was preparation, and all of it is beneath IP.

■ 23.13.2 PLAIN - a picture in your head

1. Think of arriving in a new city where you know nobody.
2. First you have to get through the door of the building. That is joining the WiFi.
3. Then you need an address of your own, so anyone can reply to you. That is DHCP.
4. Then you check nobody else already lives at that address. That is the ARP probe.
5. Then you need to know which door leads out of the building. That is the default gateway.
6. Then you need to know the physical location of that door, not just its name. That is ARP for the gateway.
7. Only then can you post your first letter.

Where this comparison breaks: a person does all of this once and remembers it for years. A laptop redoes almost all of it every time it wakes from sleep, changes network, or the lease timer expires. It also does it again silently every few minutes as ARP entries age out. The steady state of a network is not stillness; it is this whole sequence running quietly, over and over.

■ 23.13.3 PLAIN - a worked example

1. Here is the full sequence for the reader's laptop, from cold to first packet towards GitHub.

TIME	WHO	WHAT	PROTOCOL
----	-----	-----	-----

t+0	laptop	scan for networks	802.11
t+1	laptop	authenticate, associate	802.11
t+2	both	4-way key handshake	802.11i
t+3	laptop	DHCP Discover, broadcast	DHCP/UDP68
t+3	router	DHCP Offer 192.168.0.107	DHCP/UDP67
t+3	laptop	DHCP Request, broadcast	DHCP/UDP68
t+3	router	DHCP Ack, lease granted	DHCP/UDP67
t+3	laptop	ARP probe for own address	ARP
t+4	laptop	ARP who has 192.168.0.1	ARP
t+4	router	ARP reply, here is my MAC	ARP
t+5	laptop	DNS query for github.com	DNS/UDP53
t+5	resolver	answer 20.207.73.82	DNS/UDP53
t+5	laptop	TCP SYN to 20.207.73.82:443	TCP
t+20	...	nothing comes back at all	silence

- Steps t+0 to t+4 are this chapter. Everything from t+5 belongs to later chapters in Part F, and each gets its own.
- The DNS lookup went to 1.1.1.1, the Cloudflare public resolver, and it worked. The name resolved to 20.207.73.82, which is in a Microsoft-owned range; GitHub has been owned by Microsoft since 2018 and fronts traffic through Microsoft's network edge.
- So name resolution succeeded and the local network worked perfectly.
- The TCP connection attempt then received no response at all for fifteen seconds. No SYN-ACK, no reset, no ICMP unreachable.
- What that proves: layers 1 and 2 were healthy end to end on the local segment, DHCP worked, ARP worked, and DNS worked.
- What it merely suggests: the packets were being discarded silently somewhere beyond the local network. The evidence is consistent with a silent drop on that path. It is not consistent with the server being down, because the same site loaded instantly over mobile data on the same phone.
- We will not name a culprit. The observation supports a path-specific silent drop and nothing more.

■ 23.13.4 PLAIN - what is really happening inside

- Here is the same journey drawn as a picture, with the layers each hop cares about.

LAPTOP		ROUTER		THE INTERNET
192.168.0.107		192.168.0.1		
+-----+		+-----+		
app				
TCP				
IP	----->	IP	----->	hop 2 ...
Ethernet	frame	Ethernet		
radio	<----->	radio		
+-----+	802.11	+-----+		

The frame stops at the router. It is unwrapped, the IP packet inside is read, and a brand new frame is built for the next hop.

MAC addresses: change at every single hop.
IP addresses : stay the same all the way.

- That last pair of lines is the single most important idea in local networking, and the reason MAC and IP both exist.

3. The reader's traceroute confirms it. Hop 1 was 192.168.0.1, the router in the flat. Hop 2 was 172.31.0.17, inside the ISP.
4. Between those two hops the frame was completely rebuilt, with different source and destination MAC addresses, on a different physical medium.
5. The IP packet inside was unchanged apart from the TTL being decremented by one and the header checksum being recomputed.
6. Hops 2 to 6 in the reader's trace were private addresses in the RFC 1918 ranges, belonging to the ISP's own core. 172.16.0.0/12 covers 172.16.x through 172.31.x, so 172.31.0.17, 172.26.22.235 and 172.16.18.33 are all private.
7. Hop 3, 137.97.29.249, was a public address belonging to the ISP.
8. From hop 7 the path entered Microsoft's backbone at ntwk.msn.net, went through Delhi, then Mumbai, then Pune, and then produced nothing.
9. All of that is layer 3 and above, and it is the subject of the next several chapters. This chapter ends where the frame ends: at the router in the flat.

■ 23.13.5 TECHNICAL – the engineer's version

1. Sequence with the exact standards involved, for a WPA2 or WPA3 personal WiFi network:
 1. Probe request and response, or passive beacon reception. IEEE 802.11.
 2. Open system authentication, then association request and response. IEEE 802.11.
 3. 4-way handshake deriving the pairwise transient key. IEEE 802.11i, ratified 2004, marketed as WPA2. WPA3 (2018) replaces the pre-shared key exchange with SAE.
 4. DHCP DORA. RFC 2131, UDP 67 and 68.
 5. ARP announcement or probe for the assigned address. RFC 5227.
 6. ARP request for the default gateway. RFC 826.
 7. DNS query. RFC 1035, and later chapters cover it.
 8. TCP three-way handshake. RFC 9293, which in 2022 replaced RFC 793 from 1981.
2. On a wired port, replace the first three steps with autonegotiation, and optionally 802.1X authentication using EAP over LAN before the port is opened.
3. Everything above layer 2 in that list is deferred. IP addressing and subnets, NAT, DNS, routing and traceroute, TCP, and TLS each get a full chapter in Part F.
4. Diagnostic ordering follows the same sequence, and this is the practical payoff of the whole chapter. Test bottom-up and stop at the first failure.

Symptom	First thing to check
No link light	Cable, port, transceiver
Link up, no address	DHCP server, VLAN
169.254.x.x address	DHCP not answering
No ARP for gateway	Layer 2 path, VLAN
Gateway pings, nothing else	Routing or NAT
IP works, names do not	DNS
Name works, connect hangs	Path, filtering, MTU

5. The reader's fault sits on the last row. Every row above it had been passed.
6. A more precise statement of the observation: a TCP SYN was transmitted to 20.207.73.82 port 443 and no segment, no ICMP message, and no RST arrived within 15 seconds. Under RFC 9293

the client will retransmit the SYN with exponential backoff before giving up, which is why the wait was long and featureless rather than an immediate error.

7. An immediate “connection refused” would have been a TCP RST, proving something answered. An “unreachable” message would have been ICMP type 3, proving a router made a decision and told you. Silence proves only that nothing chose to speak.
8. Silence is the least informative failure mode in networking, and it is the most common one on the modern internet, because filtering devices are usually configured to drop rather than reject.

■ 23.13.6 WORDS – remember these

1. Association — joining a WiFi network — the 802.11 exchange that binds a station to an access point before any data frames.
2. Default gateway — the way out of your network — the router address used for any destination outside your own subnet.
3. Encapsulation — wrapping data in a header at each layer — a TCP segment inside an IP packet inside an Ethernet frame.
4. Hop — one router-to-router step — a single layer 3 forwarding decision, with a complete new layer 2 frame each time.
5. TTL — the counter that stops packets looping forever — time to live, decremented at every hop, packet discarded at zero.
6. Silent drop — a packet discarded with no message — a firewall DROP rather than a REJECT, giving the sender no information.
7. Proven versus suggested — what an observation actually establishes — the discipline of separating direct evidence from inference, which is the whole method of Part F.

23.98 Common wrong ideas

1. Wrong: bandwidth and speed are the same thing. Right: bandwidth is capacity per second, latency is delay before anything arrives. Doubling capacity does not reduce delay by a single microsecond.
2. Wrong: a MAC address travels with the packet across the internet. Right: it is replaced at every single hop. The MAC addresses in a frame are only ever the two ends of one physical link.
3. Wrong: a switch and a hub do the same job. Right: a hub copies every frame to every port and shares one collision domain. A switch learns and forwards to one port, giving each port its own collision domain.
4. Wrong: collisions are a normal part of modern Ethernet. Right: on a switched full-duplex link collisions are impossible. If your interface counters show collisions, you have a duplex mismatch or a real hub.
5. Wrong: a 1500-byte MTU is a physical limit of the cable. Right: it is a number chosen in the 1980 DIX specification to limit how long one station could hold a shared coax cable, kept ever since for compatibility.
6. Wrong: category 6 cable makes your internet faster. Right: cable category sets the maximum a link can carry over 100 m. If your line delivers 100 Mbit/s, Cat 5e was already twenty times more than enough.
7. Wrong: ARP is part of IP. Right: ARP rides directly on Ethernet with EtherType 0x0806. It is not carried inside IP, has no port number, and never crosses a router.
8. Wrong: seeing 169.254.x.x means the network card is broken. Right: it means no DHCP server answered. The link is probably up. That single observation rules out several faults and points at one.

9. Wrong: a traceroute ending in stars proves the destination is unreachable. Right: many routers rate-limit or block ICMP time-exceeded replies. Silence at the end of a trace is normal and proves nothing on its own.
10. Wrong: MAC address filtering secures a WiFi network. Right: addresses are broadcast in clear text in every frame header and can be changed in one command. Since MAC randomization became the phone default it does not even work as an inconvenience.
11. Wrong: fibre is faster because light is faster than electricity. Right: a signal moves through fibre at about two-thirds of the vacuum speed of light, similar to copper. Fibre wins on attenuation and bandwidth over distance, not on propagation speed.
12. Wrong: a packet capture on your own machine shows exactly what was on the wire. Right: segmentation and checksum offload reshape what you see. Frames larger than the MTU and invalid outbound checksums are normal artefacts.

23.99 Chapter summary in 20 lines

1. A network is machines sharing a medium, and every network must solve addressing, framing, error detection and medium sharing.
2. Topologies are bus, star, ring and mesh; modern local networks are stars and the internet core is a mesh.
3. A wire carries voltages, not numbers, and plain high-low signalling fails on clock recovery and DC balance.
4. Line codes fix this: Manchester doubles the bandwidth, 4B/5B and 8b/10b cost 25 percent, 64b/66b costs 3.1 percent, PAM-4 trades margin for symbols.
5. Twisting cancels interference because both wires spend equal time nearest the noise, and the receiver reads only the difference.
6. Cable categories are frequency ratings: Cat 5e at 100 MHz, Cat 6A at 500 MHz, Cat 8 at 2000 MHz, with 100 m channels except Cat 8's 30 m.
7. Fibre guides light by total internal reflection; single-mode goes tens of kilometres, multi-mode a few hundred metres, and SFP modules set the reach.
8. Bandwidth, throughput, latency and jitter are four different things, and the bandwidth-delay product explains why a fast link can still feel slow.
9. The reader's line was not slow. It was silent, which is a different fault entirely and needs a different diagnosis.
10. Ethernet began at Xerox PARC on 22 May 1973 with Metcalfe and Boggs at 2.94 Mbit/s, was standardized as IEEE 802.3 in 1985, and now runs to 800 Gbit/s.
11. CSMA/CD and the 64-byte minimum frame exist because of collision timing on a shared cable, and neither matters on a modern full-duplex link.
12. The frame is preamble 7, SFD 1, destination 6, source 6, EtherType 2, payload 46 to 1500, FCS 4, for 64 to 1518 bytes.
13. The 1500-byte MTU came from the 1980 DIX specification and has never changed; fragmentation costs headers, memory and all-or-nothing loss.
14. A MAC address is 48 bits: a 24-bit vendor OUI plus a device part, with bits for multicast and for locally administered addresses.
15. Phones and laptops now randomize their MAC addresses by default, which breaks MAC filtering and MAC-keyed DHCP reservations.
16. A switch learns which address is on which port by reading source addresses, floods when it does not know, and ages entries after about 300 seconds.

17. VLANs split one switch into several broadcast domains using a 4-byte 802.1Q tag, and spanning tree stops loops that would otherwise storm in seconds.
18. ARP maps an IP address to a MAC address by broadcast question and unicast answer, has no authentication at all, and never crosses a router.
19. DHCP hands out address, mask, gateway, DNS and more in four messages over UDP ports 67 and 68, with renewal at 50 percent and rebinding at 87.5 percent.
20. Everything in this chapter happens before a single IP packet leaves the building, and the next six chapters take that packet the rest of the way.

IP Addresses, Private and Public, NAT and Carrier-Grade NAT

24.0 What this chapter gives you

1. You will be able to say exactly what an IP address labels, and why it is not a label for a device and not a label for a person.
2. You will be able to convert any IPv4 address to binary by hand, and back again, without a calculator.
3. You will be able to read a CIDR prefix such as /24 or /12 and work out the network address, the broadcast address, the first and last usable host, and the exact host count.
4. You will be able to look at any address and say instantly whether it is public, private, loopback, link-local, multicast, or something special, and name the RFC that says so.
5. You will be able to explain why every home near you can use 192.168.0.1 at the same time without anything colliding.
6. You will be able to describe NAT precisely: what the translation table holds, what source NAT and destination NAT are, and what your home router really does, which is PAT.
7. You will be able to explain what a port is, why NAT would be impossible without ports, and what the real limit on simultaneous connections is.
8. You will be able to say what NAT breaks, why peer-to-peer software needs STUN, TURN and ICE, and why NAT is honestly a workaround rather than a design.
9. You will be able to read the reader's own traceroute, hop by hop, and say what the private addresses at hops 2 to 6 do and do not prove about carrier-grade NAT.
10. You will be able to run the exact macOS commands that reveal your address, your gateway, your mask and your public address, and decide in under a minute whether you are behind carrier-grade NAT.

24.1 What an IP address is

■ 24.1.1 PLAIN – in simple words

1. An IP address is a label. It says where to deliver something on a network.
2. The thing it labels is a **network interface**: one connection point, one way in and out of a machine.
3. It does not label the machine. A machine can have many connection points.
4. Your laptop has a Wi-Fi radio. That is one interface. It has an Ethernet socket. That is another. Each gets its own address.
5. If you turn on a VPN, the operating system makes a pretend interface in software. That gets an address too.

6. The reader's own macOS machine showed several interfaces named `utun`. Those are software tunnels. Each one can carry its own address.
7. So the question "what is my IP address" has no single answer. The honest answer is "which interface, and seen from where".
8. An IP address also does not label a person. It labels a doorway. Many people can walk through the same doorway.
9. Addresses are borrowed, not owned. Most home machines are given an address for a while and handed a different one later.

■ 24.1.2 PLAIN – a picture in your head

1. Think of a large office building with several entrances.
2. The building has a main street entrance, a loading bay at the back, and a staff door at the side.
3. Each entrance has its own number painted above it. Deliveries come to the number that matches the entrance.
4. The number is not the name of the company inside. The company could move out tomorrow and a different one moves in, keeping the same painted number.
5. The number is not the name of any person inside either. Hundreds of people use the same doorway every day.
6. If the building opens a new temporary door for a festival, that door gets its own number for the week. That is what a VPN interface is like.

Where this comparison breaks: painted door numbers are stable for decades and tied to a physical place you can walk to. IP addresses are handed out for hours or days, can be moved between cities in minutes, and one address can be served by machines on three continents at once. A door number also cannot be shared by a thousand separate buildings at the same time, which is exactly what private addresses do.

■ 24.1.3 PLAIN – a worked example

1. An IPv4 address is 32 switches in a row. Each switch is on or off, 1 or 0.
2. Writing 32 ones and zeros is horrible for humans, so we chop them into four groups of 8 and write each group as an ordinary number.
3. A group of 8 bits can hold 0 to 255. That is why every part of an address stays between 0 and 255.
4. The four numbers are written with dots between them. That is called **dotted quad** notation.
5. Let us convert the reader's own router address, 192.168.0.1, by hand.
6. Each of the 8 bits has a value. From left to right they are 128, 64, 32, 16, 8, 4, 2, 1. You take the biggest value that fits, subtract, and repeat.

Part	Working	Binary
192	128 + 64	11000000
168	128 + 32 + 8	10101000
0	nothing	00000000
1	1	00000001

7. So 192.168.0.1 in full is:

| 11000000 10101000 00000000 00000001

8. Now the address the reader's machine got for `github.com`, 20.207.73.82.

Part	Working	Binary
20	16 + 4	00010100
207	128+64+8+4+2+1	11001111
73	64 + 8 + 1	01001001
82	64 + 16 + 2	01010010

9. So 20.207.73.82 in full is:

| 00010100 11001111 01001001 01010010

10. Going backwards is just addition. Take 11001111, add up the columns that hold a 1: $128 + 64 + 8 + 4 + 2 + 1 = 207$.
11. There are 32 bits, so there are 2 multiplied by itself 32 times possible addresses. That is 4,294,967,296, usually said as about 4.29 billion.

■ 24.1.4 PLAIN – what is really happening inside

1. When your machine sends data, it wraps it in a small header, like an address label stuck on a parcel.
2. Inside that header are two fixed slots: 32 bits for where it came from, and 32 bits for where it is going.
3. The dots are not in the header. The dots exist only on paper and on screens for humans. On the wire it is 32 raw bits and nothing else.
4. Every router that touches the parcel reads the destination slot, looks it up in a table, and decides which cable to push it down next.
5. The router does not open the parcel. It does not care what is inside. It reads the label and forwards.
6. The source slot matters for the reply. Without it the far end would have no idea where to send anything back.
7. This is why a wrong source address is a real problem. If a machine lies about where it came from, replies go to the wrong place. That is called spoofing.
8. Your machine keeps a small list of its own interfaces and the address on each. When you open a connection, it picks the interface that can reach the destination and stamps that interface's address as the source.

■ 24.1.5 TECHNICAL – the engineer's version

1. IPv4 is specified in RFC 791, published September 1981, by Jon Postel at the Information Sciences Institute. It is Internet Standard STD 5.
2. The address is a 32-bit unsigned integer carried in network byte order, which means big-endian, most significant byte first.
3. In a standard 20-byte IPv4 header, the source address occupies bytes 12 to 15 and the destination address occupies bytes 16 to 19, counting from zero.
4. Dotted quad notation is a presentation convention, not part of the protocol. RFC 791 itself writes addresses that way, but the wire format has no separators.
5. The same address has several equally valid written forms.

Form	192.168.0.1	20.207.73.82
Dotted quad	192.168.0.1	20.207.73.82
Decimal integer	3232235521	349129042

Form	192.168.0.1	20.207.73.82
Hexadecimal	c0a80001	14cf4952
Old class	C	A

- The total space is 2 to the power 32, exactly 4,294,967,296 addresses. Not all of them are usable, as section 24.4 shows.
- An interface may hold more than one address. On Linux this is normal and unlimited in practice. On macOS you add extra addresses with `ifconfig en0 alias`.
- Loopback is an interface too, normally `lo0` on macOS and `lo` on Linux, carrying 127.0.0.1.
- Tunnel interfaces created by VPN software on macOS appear as `utun0`, `utun1` and so on. The reader's session showed several. Each can carry its own address and its own routes, which is exactly why a VPN can change which address the world sees without changing your Wi-Fi address at all.
- Commands that observe this: `ifconfig -a`, `ipconfig getifaddr en0`, `networksetup -listallhardwareports` on macOS; `ip -4 addr show` on Linux.

```
ipconfig getifaddr en0      # just the IPv4 address of Wi-Fi
ifconfig en0 | grep 'inet ' # address, netmask, broadcast
ifconfig -a | grep utun     # the VPN tunnel interfaces
```

■ 24.1.6 WORDS – remember these

- IP address — a delivery label for one connection point — a 32-bit unsigned integer in the IPv4 header, bytes 12 to 15 and 16 to 19.
- Interface — one way in or out of a machine — a network endpoint, physical or virtual, that can hold addresses and routes.
- Dotted quad — writing an address as four numbers with dots — the presentation format for a 32-bit IPv4 address, one decimal octet per byte.
- Octet — one of the four numbers, 0 to 255 — an 8-bit field; the term is used instead of “byte” because early machines had bytes of other sizes.
- Big-endian — biggest part written first — network byte order, the byte ordering all IP header fields use.
- Spoofing — lying about where a packet came from — forging the source address field, mitigated by ingress filtering as described in BCP 38.

24.2 Networks and hosts: the split, the mask and CIDR

■ 24.2.1 PLAIN – in simple words

- An IP address is really two facts glued together.
- The left part says which network you are on. The right part says which machine you are within that network.
- The glue point is not fixed. It can sit anywhere along the 32 bits.
- Something has to say where the split is. That something is the **subnet mask**.
- A mask is another 32-bit pattern. Every 1 in it means “this bit belongs to the network part”. Every 0 means “this bit belongs to the machine part”.
- The ones always come first and never have gaps. So a mask is fully described by counting them.
- Counting them is the whole idea of CIDR notation. You write a slash and the number of network bits. /24 means the first 24 bits are network.

8. So 192.168.0.1/24 means: the network is the first 24 bits, and 8 bits are left over to number machines.
9. Two addresses are on the same network when their network parts match. That single fact decides whether your machine talks to something directly or hands it to the router.

■ 24.2.2 PLAIN – a picture in your head

1. Think of a phone number written as an area code plus a local number.
2. The area code says which town. The local number says which phone in that town.
3. If you are calling someone in your own town, you dial the local part only. You do not need the area code.
4. If they are in a different town, you dial the whole thing, and the call goes through the long-distance exchange first.
5. Your machine does the same test. Same network part means “shout directly down the wire”. Different network part means “give it to the router”.
6. The router is the long-distance exchange. It is the only way out.

Where this comparison breaks: phone area codes are a fixed number of digits and map to real geography. A CIDR prefix can be any number of bits from 0 to 32, and it maps to whoever was allocated it, which can be a company with offices in ten countries. Phone numbers also have no equivalent of the broadcast address, and no equivalent of a mask you can change on your own equipment.

■ 24.2.3 PLAIN – a worked example

1. Take the reader’s own home network, written as 192.168.0.0/24.
2. /24 means the first 24 bits are the network. In binary:

```
address 192.168.0.1  11000000 10101000 00000000 00000001
mask      /24        11111111 11111111 11111111 00000000
|----- network -----|- host -|
```

3. The **network address** is what you get when every host bit is 0. Here the last 8 bits go to zero, giving 192.168.0.0.
4. The **broadcast address** is what you get when every host bit is 1. The last 8 bits become 11111111, which is 255, giving 192.168.0.255.
5. The first usable host is the network address plus one: 192.168.0.1. That is the reader’s router.
6. The last usable host is the broadcast address minus one: 192.168.0.254.
7. The total addresses are 2 to the power 8, which is 256. Two are reserved for network and broadcast, so 254 machines can be numbered.
8. Now take 137.97.0.0/16, the classical block containing the reader’s hop 3 at 137.97.29.249.
9. /16 means the first 16 bits are the network, so the whole last two octets are host bits.

Item	192.168.0.0/24	137.97.0.0/16
Mask	255.255.255.0	255.255.0.0
Network address	192.168.0.0	137.97.0.0
Broadcast	192.168.0.255	137.97.255.255
First usable	192.168.0.1	137.97.0.1
Last usable	192.168.0.254	137.97.255.254
Total addresses	256	65,536
Usable hosts	254	65,534

10. Check that 137.97.29.249 really falls inside 137.97.0.0/16. Compare the first 16 bits: 137.97 matches 137.97. It does.
11. Honest note: 137.97.0.0/16 is the historical block boundary. We are using it as a clean worked example. The actual prefix the reader's ISP announces to the internet could be smaller, such as a /19 or /22, carved out of it. The arithmetic is the same either way.

■ 24.2.4 PLAIN - what is really happening inside

1. When your machine wants to send to some address, it does one calculation first, and it does it for every single packet.
2. It takes the destination address and lines it up with its own mask.
3. It performs a bitwise AND. That means: for each bit position, the answer is 1 only if both are 1.
4. ANDing an address with its mask wipes out the host bits and leaves the network number.
5. It does the same to its own address, and compares the two results.
6. If they match, the destination is a neighbour. The machine finds the neighbour's hardware address and sends the frame straight to it.
7. If they do not match, the destination is elsewhere. The machine sends the frame to the router instead, using the router's hardware address but keeping the far-away destination IP inside.
8. Worked through: the reader's laptop wants 20.207.73.82.
9. 20.207.73.82 AND 255.255.255.0 gives 20.207.73.0.
10. The laptop's own 192.168.0.x AND 255.255.255.0 gives 192.168.0.0.
11. They differ, so the packet goes to 192.168.0.1, the router. This is the exact reason hop 1 of the reader's trace is the router.
12. The router then repeats the same test against its own, much larger, table.

■ 24.2.5 TECHNICAL - the engineer's version

1. CIDR, Classless Inter-Domain Routing, is defined in RFC 4632, August 2006, which is BCP 122. It obsoletes RFC 1519 and RFC 1518, both published in September 1993.
2. A prefix is written address/length. The length is the count of leading 1 bits in the mask. Masks must be contiguous; discontinuous masks were legal in some ancient equipment and are not permitted now.
3. The number of addresses in a prefix of length n is 2 to the power (32 - n).
4. Usable host count is that minus 2 for any prefix shorter than /31, because the all-zeros and all-ones host patterns are the network identifier and the directed broadcast.
5. RFC 3021, December 2000, permits /31 on point-to-point links, where both addresses are usable and there is no broadcast. Service provider backbones use /31 heavily to save address space.
6. A /32 is a single host route, used for loopbacks on routers and for advertising one address.
7. Full reference table, /8 to /32:

Prefix	Mask	Addresses	Usable hosts
/8	255.0.0.0	16,777,216	16,777,214
/9	255.128.0.0	8,388,608	8,388,606
/10	255.192.0.0	4,194,304	4,194,302
/11	255.224.0.0	2,097,152	2,097,150
/12	255.240.0.0	1,048,576	1,048,574
/13	255.248.0.0	524,288	524,286
/14	255.252.0.0	262,144	262,142
/15	255.254.0.0	131,072	131,070

Prefix	Mask	Addresses	Usable hosts
/16	255.255.0.0	65,536	65,534
/17	255.255.128.0	32,768	32,766
/18	255.255.192.0	16,384	16,382
/19	255.255.224.0	8,192	8,190
/20	255.255.240.0	4,096	4,094
/21	255.255.248.0	2,048	2,046
/22	255.255.252.0	1,024	1,022
/23	255.255.254.0	512	510
/24	255.255.255.0	256	254
/25	255.255.255.128	128	126
/26	255.255.255.192	64	62
/27	255.255.255.224	32	30
/28	255.255.255.240	16	14
/29	255.255.255.248	8	6
/30	255.255.255.252	4	2
/31	255.255.255.254	2	2 point-to-point
/32	255.255.255.255	1	1 host route

8. Routers select a next hop by **longest prefix match**: among all routes that contain the destination, the one with the most network bits wins. A /24 beats a /16 which beats the default route 0.0.0.0/0.
9. VLSM, Variable Length Subnet Masking, means different prefix lengths inside one organization. It became possible once routing protocols carried the mask with the prefix, which OSPF and BGP-4 do and RIP version 1 did not.
10. Aggregation is the point of CIDR: an ISP holding 137.97.0.0/16 announces one route to the world instead of 256 separate /24 routes. Without this the global routing table would have collapsed under its own size in the 1990s.
11. Observation commands: `ipcalc 192.168.0.0/24`, `sipcalc`, `netstat -rn` on macOS to see the routing table, `route -n get 20.207.73.82` to ask macOS which route and gateway it would pick for one destination.

```
route -n get 20.207.73.82
# route to: 20.207.73.82
# destination: default
# gateway: 192.168.0.1
# interface: en0
```

■ 24.2.6 WORDS – remember these

1. Subnet mask — the pattern saying which part is the network — a 32-bit value of contiguous 1 bits followed by 0 bits, ANDed with an address.
2. CIDR notation — the slash number — prefix length in bits, defined in RFC 4632, replacing the class system.
3. Network address — the address with all host bits 0 — the prefix identifier, not assignable to a host.
4. Broadcast address — the address with all host bits 1 — the directed broadcast for that subnet, per RFC 919.
5. Bitwise AND — keep only bits that are 1 in both — the masking operation used for every local-versus-remote forwarding decision.
6. Longest prefix match — the most specific route wins — the forwarding rule that picks the entry with the greatest prefix length.

7. VLSM — different sized subnets in one network — variable length subnet masking, requires classless routing protocols.
8. Aggregation — announcing one big block instead of many small ones — route summarization, the mechanism that keeps the global BGP table finite.

24.3 Public and private addresses

■ 24.3.1 PLAIN – in simple words

1. Some addresses are meant for the open internet. Anyone anywhere can reach them. Those are **public** addresses.
2. Some addresses are set aside for use inside one building or one home. Those are **private** addresses.
3. A private address only means something inside its own network. Outside, it means nothing.
4. Three blocks of addresses were reserved for private use in 1996, in a document called RFC 1918.
5. The three blocks are: everything starting 10, a range starting 172.16 up to 172.31, and everything starting 192.168.
6. The reader's router is 192.168.0.1. That is in the third block. So is almost every home router in the world.
7. Your neighbour's router is very likely also 192.168.0.1 or 192.168.1.1. That causes no problem at all.
8. It causes no problem because private addresses never travel on the public internet. Routers out there refuse to carry them.
9. So the two 192.168.0.1 addresses never meet. They live in separate sealed worlds.
10. Something must translate between the sealed world and the open one. That something is NAT, and it is section 24.6.

■ 24.3.2 PLAIN – a picture in your head

1. Think of the extension numbers in an office phone system.
2. Inside the office you dial 101 and reach reception. Extension 101 exists in that office and nowhere else.
3. Every office in the city has an extension 101. There is no clash, because you cannot dial an extension from outside.
4. From outside you dial the single main number of the building. Only that number is listed publicly.
5. The main number is the public address. The extensions are the private addresses.
6. The switchboard is the router doing NAT. It is the only thing that knows both worlds.

Where this comparison breaks: a human switchboard operator can put an incoming call through to any extension on request. A home router cannot do that by default. Nobody outside can ask for an extension, because the router has no idea which internal machine an unexpected incoming packet is meant for. That single difference is the source of most of the pain in section 24.8.

■ 24.3.3 PLAIN – a worked example

1. Here are the three private blocks with their exact boundaries.

Block	Range	Addresses
10.0.0.0/8	10.0.0.0 to 10.255.255.255	16,777,216
172.16.0.0/12	172.16.0.0 to 172.31.255.255	1,048,576
192.168.0.0/16	192.168.0.0 to 192.168.255.255	65,536

2. The middle one confuses people, so let us prove its range.
3. 172.16.0.0/12 fixes the first 12 bits. The first octet, 172, uses 8 of them. That leaves 4 more, taken from the second octet.
4. In binary, 172 is 10101100. The second octet 16 is 00010000. The first four bits of the second octet are 0001, and those are frozen.
5. The remaining four bits of the second octet are free. They range from 0000 to 1111. So the second octet runs from 00010000 to 00011111, which is 16 to 31.
6. Therefore the block covers 172.16.x.x all the way to 172.31.x.x. It does not cover 172.32.x.x, and it does not cover 172.15.x.x.
7. Now apply that to the reader's own traceroute. Hops 2 to 6 were:

2	172.31.0.17
3	137.97.29.249
4	172.26.22.235
5	172.16.18.33 / 172.26.14.75 / 172.26.22.234
6	172.26.14.75 / 172.16.18.33
8. 172.31.0.17 has second octet 31. That is inside 16 to 31. Private.
9. 172.26.22.235, 172.26.14.75 and 172.26.22.234 all have second octet 26. Inside 16 to 31. Private.
10. 172.16.18.33 has second octet 16, the very bottom edge of the block. Private.
11. 137.97.29.249 is not in any private block. 137 is not 10, not 172, not 192. Public.
12. So of the reader's first six hops, five are private and one is public. That single fact is the whole story of section 24.9.

■ 24.3.4 PLAIN – what is really happening inside

1. There is nothing physically different about a private address. The bits look the same as any other bits.
2. The difference is entirely an agreement. Everyone agreed not to give these blocks to anybody as public addresses, and agreed that routers on the public internet would not carry them.
3. Two mechanisms enforce it. First, the address registries never allocate these blocks to anyone. Second, network operators filter them.
4. Filtering means a router is configured with a rule: if a route for 192.168.0.0/16 arrives from a neighbour, discard it. If a packet with a private source arrives from outside, drop it.
5. Those unwanted routes and packets are called **bogons**, short for bogus.
6. So a packet with source 192.168.0.24 sent onto the open internet would either be dropped, or would arrive somewhere and generate a reply that could never find its way home.
7. That is why the translation in NAT is not optional. A private address must be swapped for a public one before it leaves.
8. Inside a network, private addresses behave exactly like public ones. Routing, masks, broadcasts, all identical. Only the outside world treats them differently.
9. Large organizations use 10.0.0.0/8 precisely because it is huge: 16.7 million addresses, enough for a national telecom network's internal plumbing.

■ 24.3.5 TECHNICAL – the engineer's version

1. RFC 1918, "Address Allocation for Private Internets", February 1996, is BCP 5. Authors include Yakov Rekhter and Robert Moskowitz. It updated the earlier RFC 1597 from March 1994.
2. The three blocks and their historical classful description:

Prefix	RFC 1918 name	Old class equivalent
10.0.0.0/8	24-bit block	one class A network
172.16.0.0/12	20-bit block	16 class B networks
192.168.0.0/16	16-bit block	256 class C networks

3. RFC 1918 hosts are described in the document as being in one of three categories, from category 1 needing no external access at all up to category 3 needing full external access through translation.
4. The complete authoritative list of special IPv4 blocks lives in the IANA IPv4 Special-Purpose Address Registry, whose framework is RFC 6890, April 2013, later corrected by RFC 8190, June 2017.
5. Filtering practice: operators drop RFC 1918 prefixes at the edge. This is a strong convention and appears in BCP 84 and in the Team Cymru bogon lists, but it is not enforced by the protocol itself. A misconfigured network can and does leak private prefixes.
6. Private addresses appearing inside a traceroute are entirely normal. Many carriers number their internal point-to-point links from 10.0.0.0/8 or 172.16.0.0/12 to conserve public space. The reader's hops 2, 4, 5 and 6 are exactly this.
7. Important precision: an RFC 1918 address seen in a traceroute is the address of the router interface that generated the ICMP time-exceeded reply. It is not the address your packet was carrying. Do not confuse the two. Section 24.9 depends on this distinction.
8. A private address can still be globally reachable through a tunnel, a VPN or an MPLS VPN, because those carry it inside another packet. The rule is about the public routing table, not about physics.
9. Overlap is a real operational problem. When two companies merge and both use 10.0.0.0/8, engineers have to renumber one side or deploy NAT between them, sometimes called twice-NAT or NAT overlapping networks.
10. Observation: `netstat -rn` shows which prefixes your machine treats as local. `whois 137.97.29.249` at a regional registry will name the holder of a public block; `whois 172.31.0.17` will tell you it is RFC 1918 and nothing more, because nobody holds it.

■ 24.3.6 WORDS – remember these

1. Public address — an address the whole internet can reach — a globally unique address allocated through IANA and a regional registry.
2. Private address — an address meaningful only inside one network — an address from 10.0.0.0/8, 172.16.0.0/12 or 192.168.0.0/16 per RFC 1918, BCP 5.
3. RFC 1918 — the 1996 document reserving private ranges — Best Current Practice 5, “Address Allocation for Private Internets”.
4. Bogon — an address or route that should never appear — a martian prefix; traffic or routing announcements from reserved or unallocated space.
5. Ingress filtering — dropping packets with impossible sources at the edge — the practice described in BCP 38 and RFC 2827.
6. Address overlap — two networks using the same private numbers — the merger problem solved by renumbering or twice-NAT.

24.4 The other special ranges

■ 24.4.1 PLAIN – in simple words

1. Private addresses are not the only ones with a special job. Several other blocks are carved out and cannot be used as ordinary addresses.

2. **Loopback:** everything starting with 127. This means “me, this machine”. Packets sent there never touch a cable. They turn around inside the software.
3. 127.0.0.1 is the one everybody types. The name for it is `localhost`.
4. **Link-local:** everything starting 169.254. A machine gives itself one of these when it asks for an address and nobody answers.
5. If you ever see 169.254 on your laptop, it means the address request failed. It is a symptom, not a setting.
6. **Shared address space:** everything from 100.64 to 100.127. This block was set aside in 2012 for internet providers to number their customers when they have run out of public addresses.
7. **Multicast:** everything from 224 to 239. These are not one machine. They are a group. One packet, many listeners.
8. **All zeros**, 0.0.0.0, means “no address yet” or “every address here”, depending on where it appears.
9. **All ones**, 255.255.255.255, means “everyone on this wire, right now”.
10. There are also three blocks reserved purely for writing examples in books and manuals, so that a printed example never accidentally names a real machine.

■ 24.4.2 PLAIN – a picture in your head

1. Think of the reserved words in a language.
2. In English, “here” and “everyone” and “nobody” are not names. You cannot call a child “Everyone” and expect the register to accept it.
3. 127.0.0.1 is the word “here”. It always points back at the speaker, whoever is speaking.
4. 255.255.255.255 is the word “everyone in this room”. It never leaves the room, because shouting does not travel through walls.
5. 0.0.0.0 is the phrase “no name yet”, used by someone who has just walked in and has not been introduced.
6. 169.254.x.x is the name a person invents for themselves when nobody at the door would give them a badge.
7. 192.0.2.x is like the name “John Doe” in a legal form. It is deliberately nobody.

Where this comparison breaks: reserved words in a language are enforced by custom, and you can break the rule and be merely odd. These reserved address blocks are enforced by software. Your operating system will refuse to route 127.x.x.x off the machine even if you order it to, and other people’s routers will simply discard the traffic. The rule is mechanical, not social.

■ 24.4.3 PLAIN – a worked example

1. Type this on any machine and it will answer without touching a network:

```
ping -c 2 127.0.0.1
# 64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.045 ms
```

2. A time of about 0.05 milliseconds is not a fast network. It is no network. The packet went down the stack and straight back up.
3. Unplug your Ethernet cable and turn off Wi-Fi, and 127.0.0.1 still works. That is the test that proves it never leaves the machine.
4. Now the failure case. Take a laptop, plug it into a switch with no DHCP server, and wait about twenty seconds.

```
ipconfig getifaddr en0
# 169.254.213.7
```

5. That address was not given to the laptop. The laptop invented it, picked at random from 169.254.0.0/16, after checking nobody else on the wire had it.
6. Two laptops that both do this can talk to each other, and to nothing else. There is no gateway, so nothing routes.
7. Now the carrier case. If your router's outside address is 100.64.9.31, you are not on the internet directly. Your provider has put you behind their own translation. That is carrier-grade NAT, and section 24.9 covers it.
8. The reader's own hops do not show 100.64.x.x. They show RFC 1918 addresses instead. That is a different but equally common carrier choice, and it is discussed honestly in section 24.9.

■ 24.4.4 PLAIN – what is really happening inside

1. For loopback, the operating system's network stack checks the destination before anything reaches a driver. If the destination is on the loopback interface, the packet is handed straight back to the receive path.
2. That is why loopback latency measures the speed of your kernel, not of any wire, and why loopback has an enormous maximum packet size, typically 16384 bytes on macOS instead of the usual 1500.
3. For link-local, the machine picks a random address in 169.254.1.0 to 169.254.254.255, then sends an ARP probe asking "does anyone own this".
4. If somebody answers, it picks another and tries again. If nobody answers after a few tries, it keeps the address.
5. That process is why the address appears about fifteen to twenty seconds after the DHCP attempt fails, not instantly.
6. For multicast, routers and switches maintain group membership. A machine announces "I want group 239.1.1.1" and the network arranges for copies to reach it. One sender, many receivers, one copy per link rather than per receiver.
7. For 255.255.255.255, routers never forward it. It reaches every machine on the local segment and stops. That limit is deliberate; otherwise one packet could reach the whole internet.
8. This is exactly how DHCP starts. A machine with no address at all sends from 0.0.0.0 to 255.255.255.255, because it cannot address the server it has not met yet.

■ 24.4.5 TECHNICAL – the engineer's version

1. The full set of special-purpose IPv4 blocks, each with the document that defines it:

Prefix	Purpose	Defining RFC
0.0.0.0/8	this network	RFC 791
0.0.0.0/32	unspecified source	RFC 1122
10.0.0.0/8	private	RFC 1918
100.64.0.0/10	shared, for CGNAT	RFC 6598
127.0.0.0/8	loopback	RFC 1122
169.254.0.0/16	link-local	RFC 3927
172.16.0.0/12	private	RFC 1918
192.0.0.0/24	IETF protocol use	RFC 6890
192.0.2.0/24	documentation, TEST-NET-1	RFC 5737
192.168.0.0/16	private	RFC 1918
198.18.0.0/15	benchmark testing	RFC 2544
198.51.100.0/24	documentation, TEST-NET-2	RFC 5737
203.0.113.0/24	documentation, TEST-NET-3	RFC 5737

Prefix	Purpose	Defining RFC
224.0.0.0/4	multicast	RFC 1112
240.0.0.0/4	reserved, future use	RFC 1112
255.255.255.255/32	limited broadcast	RFC 919, RFC 8190

- RFC 1122, “Requirements for Internet Hosts - Communication Layers”, October 1989, edited by Robert Braden, is what reserves all of 127.0.0.0/8 for loopback, not just 127.0.0.1. That is 16,777,216 addresses spent on one idea.
- RFC 3927, “Dynamic Configuration of IPv4 Link-Local Addresses”, May 2005, defines 169.254.0.0/16. Apple’s implementation is branded Bonjour, and Microsoft’s is called APIPA, Automatic Private IP Addressing. Both implement the same RFC.
- RFC 6598, “IANA-Reserved IPv4 Prefix for Shared Address Space”, April 2012, is BCP 153. It reserves 100.64.0.0/10, which is 4,194,304 addresses, from 100.64.0.0 to 100.127.255.255. It exists because carriers were using RFC 1918 space for CGNAT and colliding with customers’ own networks.
- RFC 1112, “Host Extensions for IP Multicasting”, August 1989, by Steve Deering, defines 224.0.0.0/4. Assignment inside it is governed by RFC 5771, BCP 51. Notable addresses: 224.0.0.1 all hosts on this subnet, 224.0.0.2 all routers, 224.0.0.5 and 224.0.0.6 for OSPF, 224.0.0.251 for mDNS.
- RFC 5737, January 2010, reserves the three documentation blocks. This chapter deliberately does not use them, because the reader asked for their own real addresses. Where an address here is invented rather than measured, it is labelled as such.
- RFC 2544, March 1999, reserves 198.18.0.0/15 for device benchmarking; RFC 6815, November 2012, warns that this traffic must never leave the lab.
- 240.0.0.0/4 is 268 million addresses sitting unused since 1989. There have been repeated IETF proposals to release it. It has not happened, because every operating system, router and firewall on earth would need updating to accept it, and the gain is roughly one year of global growth.
- 0.0.0.0/0 as a route means “everything”, the default route. 0.0.0.0 as a bind address in a server means “listen on every local address”. Same digits, entirely different meaning depending on the field it sits in.
- Observation: `netstat -rn` on macOS shows the 127.0.0.0/8 route pointing at `lo0` and the default route pointing at 192.168.0.1 for the reader.

■ 24.4.6 WORDS - remember these

- Loopback — the address that means this machine — 127.0.0.0/8 per RFC 1122, handled entirely inside the kernel, never placed on a link.
- Link-local — the address a machine gives itself when nothing answers — 169.254.0.0/16 per RFC 3927, claimed by ARP probe, no gateway.
- Shared address space — the block providers use to number customers behind their own NAT — 100.64.0.0/10 per RFC 6598, BCP 153.
- Multicast — one packet delivered to a group — 224.0.0.0/4 per RFC 1112, with group membership managed by IGMP.
- Limited broadcast — everyone on this wire and no further — 255.255.255.255/32, never forwarded by a router.
- Unspecified address — I do not have an address yet — 0.0.0.0 as a source, as used by a DHCP client before it is configured.

7. TEST-NET — addresses reserved for writing examples — 192.0.2.0/24, 198.51.100.0/24 and 203.0.113.0/24 per RFC 5737.

24.5 Address exhaustion: why 4.29 billion was not enough

■ 24.5.1 PLAIN – in simple words

1. There are 4,294,967,296 possible IPv4 addresses. There are more than 8 billion people, and far more than 8 billion devices.
2. So even in the best case the numbers do not work. But the shortage arrived much earlier than the raw count suggests.
3. It arrived early because of how the addresses were handed out, not because they were all in use.
4. In the beginning, addresses were given out in three fixed sizes only: about 16 million, about 65 thousand, or 256.
5. There was nothing in between. If your university needed 400 addresses, 256 was too few, so you got 65,536 and wasted 65,136 of them.
6. Big early organizations were handed 16 million addresses each, often for a few thousand machines.
7. By 1992 people could see the end coming. Not just the end of addresses, but the end of the routers, which were running out of memory to hold all the separate routes.
8. In 1993 the fixed sizes were abolished and replaced by the slash notation of section 24.2. That let blocks be any size, which stopped most of the waste.
9. Private addresses plus NAT arrived soon after, which let one public address serve a whole household or office.
10. Those two changes bought about twenty extra years. The free pool of unused addresses still ran dry, starting in 2011.

■ 24.5.2 PLAIN – a picture in your head

1. Imagine a city where land is sold only in three plot sizes: a whole district, a whole street, or a single house.
2. A family needing two houses must buy a whole street, and leave the rest empty, because there is no plot of two.
3. A school needing thirty houses must also buy a street. So does a factory needing sixty.
4. Early buyers, arriving when land felt endless, took whole districts for a handful of buildings.
5. The city fills up not because it is full of people, but because it is full of fences around empty ground.
6. In 1993 the city changed the rules: you can now buy any number of plots you actually need. That helped enormously, but the districts already sold stayed sold.

Where this comparison breaks: land can be compulsorily bought back and redeveloped. IPv4 blocks mostly cannot. Some early holders have sold blocks on an open market, and some space has been returned, but there is no authority that can take an allocation back by force. The fences from 1985 are still standing.

■ 24.5.3 PLAIN – a worked example

1. Here is the old class system, the one abolished in 1993.

Class	First bits	Prefix	Hosts each
A	0	/8	16,777,214
B	10	/16	65,534

Class	First bits	Prefix	Hosts each
C	110	/24	254
D	1110	multicast	not applicable
E	1111	reserved	not applicable

- The class was decided by the leading bits of the address itself, so you could tell the network size by looking at the first number.
- First number 1 to 126 meant class A. 128 to 191 meant class B. 192 to 223 meant class C.
- Check the reader's own addresses against that. 192.168.0.1 begins with 192, so it sat in the old class C range. 137.97.29.249 begins with 137, so class B. 20.207.73.82 begins with 20, so class A.
- That last one is real history, not trivia. The 20.0.0.0/8 block, all 16.7 million addresses, was allocated in the 1980s to a single company. Microsoft later held it, and today it fronts services including the GitHub endpoint the reader was trying to reach.
- Now the waste. Suppose a company in 1990 needs 400 addresses.
- A class C gives 254. Not enough.
- A class B gives 65,534. So they get a class B, and 65,134 addresses sit idle forever.
- With CIDR the same company gets a /23, which is 512 addresses, 510 usable. Waste drops from 65,134 to 110.
- Multiply that saving across every allocation made after 1993 and you have the reason the internet did not stop growing in 1996.

■ 24.5.4 PLAIN - what is really happening inside

- Two separate crises were happening at once, and it helps to keep them apart.
- The first is **address exhaustion**: running out of numbers to give people.
- The second is **routing table growth**: the routers in the middle of the internet were running out of memory to store one entry per network.
- Class B allocation was making both worse. Every class B handed out was one more route every core router had to remember.
- CIDR fixed both. Smaller allocations meant less waste, and blocks handed to one provider could be advertised as one summary route instead of hundreds.
- NAT then attacked the first problem from the other end: instead of finding more addresses, it made one address serve many machines.
- The IANA free pool is the top of a three-level tree. IANA holds unallocated space, gives large blocks to five regional registries, and those registries give blocks to providers and companies.
- When people say the internet ran out in 2011, they mean the top level ran dry. The registries kept issuing from their own stock for years afterwards, under increasingly strict rules.
- Today, in most regions, new IPv4 space is obtained by buying it from an existing holder on a transfer market, not by asking a registry.

■ 24.5.5 TECHNICAL - the engineer's version

- Total IPv4 space is 2 to the power 32, exactly 4,294,967,296. Subtracting the special blocks of section 24.4, roughly 3.7 billion are usable as ordinary public unicast addresses.
- The exhaustion timeline, with verified dates:

Date	Event
31 January 2011	IANA gave last 2 free /8s

Date	Event
3 February 2011	NRO declared pool depleted
15 April 2011	APNIC hit final /8 policy
September 2012	RIPE NCC hit final /8
10 June 2014	LACNIC exhaustion
24 September 2015	ARIN exhaustion
2017	AFRINIC entered phase 1
25 November 2019	RIPE NCC fully out

3. The precision matters. On 31 January 2011 IANA allocated 39.0.0.0/8 and 106.0.0.0/8 to APNIC, which triggered the “final five” policy under which the five remaining /8s were distributed one to each registry. The Number Resource Organization made the formal announcement on 3 February 2011. So both dates are correct, for different events.
4. APNIC, serving Asia-Pacific including India, reached its final /8 first, on 15 April 2011, because demand there was highest. That is directly relevant to the reader: an Indian ISP has been operating under severe IPv4 scarcity for over a decade, which is the background to section 24.9.
5. CIDR was specified in RFC 1518 and RFC 1519, both September 1993, and deployed through 1994. RFC 4632, August 2006, BCP 122, is the current specification.
6. Classful addressing was formally deprecated by RFC 4632. Some textbooks and certification syllabuses still teach classes; treat them as history that explains why blocks look the way they do, not as current practice.
7. The measures that bought time, in rough order of impact: CIDR from 1993, NAT with RFC 1918 from 1994 to 1996, DHCP address recycling from RFC 2131 in 1997, and carrier-grade NAT from around 2010.
8. Global BGP routing table size is the counter-measure to watch. It passed roughly 100,000 routes around 2000 and is now well past 900,000 IPv4 routes. Aggregation slowed the growth; it did not stop it.
9. Transfer markets now set a real price for IPv4. Reported per-address prices rose from a few dollars in 2011 into the mid-tens of dollars by the mid-2020s, with real volatility. Treat any single quoted figure as approximate and dated.
10. Amazon Web Services began charging for every public IPv4 address on 1 February 2024, at 0.005 US dollars per address per hour. That is the clearest signal that IPv4 became a priced commodity rather than a free resource.

■ 24.5.6 WORDS – remember these

1. Address exhaustion — running out of numbers to hand out — depletion of unallocated IPv4 unicast space at IANA and the regional registries.
2. Classful addressing — the old fixed sizes A, B and C — the pre-1993 scheme where the leading bits determined the prefix length; deprecated by RFC 4632.
3. CIDR — letting blocks be any size — Classless Inter-Domain Routing, RFC 1518 and RFC 1519 in 1993, now RFC 4632.
4. IANA — the body holding the master pool — Internet Assigned Numbers Authority, operated under Public Technical Identifiers within ICANN.
5. RIR — a regional address registry — one of AFRINIC, APNIC, ARIN, LACNIC and RIPE NCC.
6. Free pool — unallocated addresses still available — the stock a registry can issue without a transfer.
7. Transfer market — buying addresses from an existing holder — registry- approved reassignment of allocated space, now the main source of IPv4.

24.6 NAT, explained completely

■ 24.6.1 PLAIN – in simple words

1. You have one public address, from your provider. You have a laptop, a phone, a television and a printer. All four want the internet at once.
2. NAT is the trick that lets all four share the one address.
3. NAT stands for Network Address Translation. The router rewrites addresses as packets pass through it.
4. Going out, the router takes your private source address and replaces it with its own public one. The far end never sees your real address.
5. Coming back, the router does the reverse. It puts your private address back before delivering the reply to you.
6. To do the reverse it must remember which reply belongs to which machine. So it keeps a table.
7. The table has one row per conversation. It is created when you start a conversation and deleted when the conversation ends or goes quiet.
8. The key trick is that the router also rewrites the port number, not just the address. That is what allows many machines to share one address at once.
9. Ports are section 24.7. For now, treat a port as a numbered slot on the router, one per conversation.
10. Nothing on the internet knows any of this happened. To the far end, it looks like one machine with one address doing a lot of things.

■ 24.6.2 PLAIN – a picture in your head

1. Think of a hotel with one street address and one post room.
2. A guest in room 214 writes a letter and asks for a reply.
3. The post room will not put “Room 214” on the envelope. It writes the hotel’s street address, and adds a reference number: “reply to reference 51022”.
4. It notes in a ledger: reference 51022 belongs to room 214, and the letter went to this particular company.
5. The reply arrives addressed to the hotel, quoting reference 51022.
6. The post room looks up 51022 in the ledger, sees room 214, writes that on the envelope, and puts it in the guest’s pigeonhole.
7. A thousand guests can write letters at once, because each gets a different reference number.
8. If a letter arrives quoting a reference that is not in the ledger, the post room has no idea what to do with it. It goes in the bin.

Where this comparison breaks: a hotel post room can be told in advance “any letter for the conference goes to room 214”, which is what port forwarding is. But without that instruction, unexpected post is genuinely undeliverable, not merely inconvenient. Also, the hotel ledger is permanent; a NAT table entry is deleted after a timeout, typically a few minutes for a quiet UDP conversation, which is why idle connections die and why applications send keep-alives.

■ 24.6.3 PLAIN – a worked example

1. Here is the reader’s own situation, with the one value we have to invent clearly marked.
2. Known from the reader’s session: the router is 192.168.0.1, and github.com resolved to 20.207.73.82, and the connection was to port 443, which is HTTPS.
3. Not recorded in the session: the laptop’s own address and the router’s public address. We will use 192.168.0.24 for the laptop, which is a plausible value in the reader’s 192.168.0.0/24 network, and we mark the public address as an example.

4. The laptop opens a connection. The operating system picks a random unused local port, say 52344.
5. The packet leaving the laptop looks like this:

```
source      192.168.0.24:52344
destination 20.207.73.82:443
protocol    TCP
```

6. It reaches 192.168.0.1. The router rewrites the source and writes a row in its table.
7. The packet leaving the router looks like this:

```
source      203.0.113.45:61001  (example public address)
destination 20.207.73.82:443
protocol    TCP
```

8. And the table row it created:

Internal	External	Destination
192.168.0.24:52344	203.0.113.45:61001	20.207.73.82:443

9. Now the full table with three devices in the flat, all talking at once. This is what a real home router holds.

Internal addr:port	External port	Dest addr:port	Proto
192.168.0.24:52344	61001	20.207.73.82:443	TCP
192.168.0.24:52345	61002	20.207.73.82:443	TCP
192.168.0.24:52346	61003	140.82.121.4:22	TCP
192.168.0.31:49780	61004	1.1.1.1:53	UDP
192.168.0.31:51201	61005	20.207.73.82:443	TCP
192.168.0.57:60110	61006	17.253.144.10:123	UDP

10. Read row 1 and row 5 together. Two different machines, both talking to 20.207.73.82 on port 443, both leaving as the same public address. They are told apart only by the external port, 61001 versus 61005.
11. A reply arrives at the router addressed to 203.0.113.45 port 61005. The router looks up 61005, finds 192.168.0.31:51201, rewrites the destination, and forwards it inside.
12. Now the crucial negative case. A packet arrives from the internet addressed to 203.0.113.45 port 8080, and no row mentions 8080. The router has no idea which of the three machines it is for. It drops it.
13. That single drop is the whole of section 24.8.

■ 24.6.4 PLAIN – what is really happening inside

1. Rewriting the address is not just a copy. The router must also fix the checksums.
2. A checksum is a small number computed from the packet contents, used to catch corruption. Change the address and the checksum no longer matches.
3. So the router recomputes the IP header checksum, and also the TCP or UDP checksum, because those are computed over the addresses as well.
4. Good implementations do this incrementally: they adjust the old checksum by the difference rather than recomputing from scratch, which is far faster.
5. The table entry is created by the first packet of a conversation and removed on a timer, or when the connection closes cleanly.

6. For TCP, the router can watch the connection state. It sees the opening handshake and the closing exchange, so it knows when a row is finished.
7. For UDP, there is no handshake and no close. The router can only use a timer, typically 30 to 300 seconds of silence.
8. This is why a video call freezes if you pause it for a few minutes, and why chat applications send a tiny packet every 20 to 30 seconds for no reason other than keeping the NAT row alive.
9. The table lives in memory and is finite. Cheap home routers hold a few thousand rows. Fill it and new connections simply fail, often with no useful error.
10. The router also has to be careful about port collisions. If two internal machines happen to pick the same source port to the same destination, the router must allocate different external ports, which it does automatically.

■ 24.6.5 TECHNICAL – the engineer’s version

1. NAT was first described in RFC 1631, “The IP Network Address Translator”, May 1994, by Kjeld Egevang and Paul Francis. It was explicitly proposed as a short-term measure.
2. Terminology was standardized in RFC 2663, August 1999. The current description of the common case is RFC 3022, “Traditional IP Network Address Translator”, January 2001, which obsoletes RFC 1631.
3. RFC 3022 separates two things. **Basic NAT** translates addresses only. **NAPT**, Network Address Port Translation, translates address and transport identifier together. Home routers do NAPT.
4. The industry name for NAPT is **PAT**, Port Address Translation, in Cisco documentation, or **NAT overload**. Linux calls it **masquerade**. All four words describe the same mechanism.
5. The four kinds you must be able to name:

Kind	What it rewrites	Typical use
Source NAT	source addr on egress	home router outbound
Destination NAT	dest addr on ingress	port forwarding
Static NAT	one to one, both ways	a public server
PAT / NAPT	addr plus port, many to one	every home router

6. The state a NAT keeps per flow is keyed on the 5-tuple: protocol, source address, source port, destination address, destination port. That is what makes rows unique even when several fields repeat.
7. NAT behaviour is classified by RFC 4787, BCP 127, January 2007, for UDP, and RFC 5382 for TCP. The mapping behaviours are endpoint-independent, address-dependent, and address-and-port-dependent. Endpoint-independent mapping is required by BCP 127 because it is the only behaviour that makes peer-to-peer hole punching reliable.
8. The older names cone NAT, restricted cone, port-restricted cone and symmetric NAT come from RFC 3489, the original STUN document from 2003. That classification was found too coarse and was deliberately replaced by the RFC 4787 vocabulary. You will still see the old names in game-console diagnostics, where “NAT type 3” means a restrictive NAT.
9. Default timeouts, as a guide rather than a standard: established TCP flows are often held 2 to 24 hours, TCP in the opening handshake around 4 minutes, UDP 30 to 180 seconds, ICMP around 30 seconds. RFC 5382 requires at least 2 hours 4 minutes for established TCP; RFC 4787 requires at least 2 minutes for UDP.
10. Observation on Linux: `conntrack -L` lists live NAT and connection tracking rows, and `cat /proc/sys/net/netfilter/nf_conntrack_max` gives the table size. On macOS the relevant tool is

`pfctl -s state`. On a home router the table is usually visible in the web interface under a name like “NAT sessions” or “connection list”.

```
# Linux router: see the live translation table
sudo conntrack -L | head -3
# tcp 6 431999 ESTABLISHED src=192.168.0.24 dst=20.207.73.82
#   sport=52344 dport=443 src=20.207.73.82 dst=203.0.113.45
#   sport=443 dport=61001 [ASSURED]
```

■ 24.6.6 WORDS – remember these

1. NAT — rewriting addresses as packets cross a boundary — Network Address Translation, RFC 2663 and RFC 3022.
2. NATP or PAT — rewriting address and port so many can share one — Network Address Port Translation, the mechanism in every home router.
3. Source NAT — changing where a packet says it came from — SNAT, applied on egress, the out-bound direction.
4. Destination NAT — changing where a packet is going — DNAT, applied on ingress; port forwarding is a DNAT rule.
5. Static NAT — a fixed one-to-one mapping in both directions — a bidirectional NAT binding, used to publish a server on a dedicated public address.
6. NAT table — the router’s memory of who is talking to whom — the translation state table, keyed on the 5-tuple.
7. 5-tuple — the five facts that identify one conversation — protocol, source address, source port, destination address, destination port.
8. Masquerade — the Linux word for many-to-one source NAT — an iptables or nftables target that uses the egress interface’s current address.

24.7 Ports: the trick that makes NAT work

■ 24.7.1 PLAIN – in simple words

1. An address gets a packet to the right machine. A **port** gets it to the right conversation on that machine.
2. A port is just a number. It is 16 bits, so it runs from 0 to 65535.
3. It is not a physical thing. There is no socket on the back of the computer labelled 443. It is a number written in the packet header.
4. Your machine may be running a web browser, a mail program, a music stream and a code editor, all talking at once. Every one of those conversations has its own port number.
5. When data arrives, the operating system reads the port and hands the data to the program that claimed that number.
6. A server claims a fixed, well-known port so clients can find it. Web servers with encryption claim 443. That is why the reader’s `curl` said port 443.
7. A client does not need a fixed number. It grabs any free one, uses it for that one conversation, and releases it afterwards.
8. Those temporary client numbers are called **ephemeral** ports, meaning short-lived.
9. NAT depends completely on this. Because the router can change the port as well as the address, it can give every conversation a unique number even when they all share one address.
10. Without ports, one public address could carry exactly one conversation at a time, and NAT would be useless.

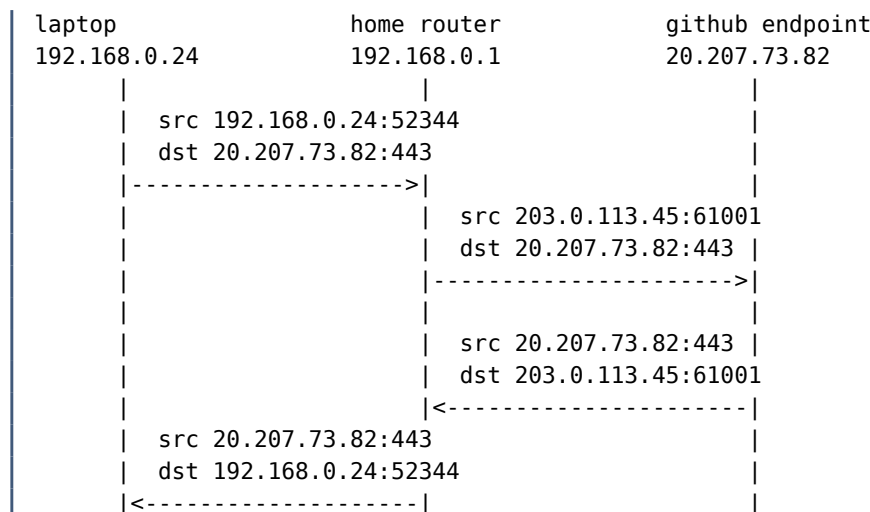
■ 24.7.2 PLAIN – a picture in your head

1. Think of a large company with one street address and many departments.
2. The street address gets the post to the building. “Attention: Accounts” gets it to the right desk inside.
3. Some departments have famous, published names. Everyone knows to write “Attention: Sales” without asking. Those are the well-known ports.
4. When Sales writes out to a supplier, it says “reply to case 52344”. That case number is invented for this one exchange and forgotten afterwards. That is an ephemeral port.
5. Two letters can go to the same supplier on the same day from the same building, and come back correctly, because the case numbers differ.

Where this comparison breaks: a company can invent unlimited case numbers. A machine has exactly 65,536 port numbers per protocol per address, and no more. That hard ceiling is a real engineering limit that busy servers and carrier NAT boxes genuinely run into, and it is the subject of the technical block below.

■ 24.7.3 PLAIN – a worked example

1. Here is the full picture of the reader’s connection attempt, showing address and port at every stage.



2. Note what stayed the same all the way through: the destination port 443. The server’s port is never rewritten by your router.
3. Note what changed: the source address and the source port, both rewritten outbound and both restored inbound.
4. Here are the port ranges and what each is for.

Range	Name	Who uses it
0 to 1023	System, well-known	servers, needs privilege
1024 to 49151	User, registered	named services
49152 to 65535	Dynamic, ephemeral	clients, never assigned

5. Some well-known ports worth memorizing, several of which appear in the reader’s own session:

Port	Service	Note
22	SSH	git over SSH
53	DNS	the reader used 1.1.1.1
80	HTTP	unencrypted web
123	NTP	clock sync
443	HTTPS	git over HTTPS, curl
993	IMAPS	encrypted mail fetch
3306	MySQL	database
5432	PostgreSQL	database

6. This table explains something from the reader's own session directly. Git over SSH uses port 22. Git over HTTPS uses port 443. They are different conversations to possibly different machines, so one can fail while the other works.

■ 24.7.4 PLAIN – what is really happening inside

1. The port numbers are not in the IP header. They are in the next header inside, the TCP or UDP header.
2. In both TCP and UDP, the source port is the first two bytes and the destination port is the next two bytes. They are the very first thing in that header, which makes them cheap for hardware to read.
3. When your program asks to open a connection, it usually does not choose a port. It asks the operating system for any free one.
4. The operating system keeps a record of which ports are in use, picks a free one from its ephemeral range, and marks it taken.
5. Modern systems pick randomly within the range rather than counting upwards. That is a deliberate security change, because predictable ports made certain attacks easier.
6. The combination that must be unique is not the port alone. It is the whole 5-tuple: protocol, both addresses, both ports.
7. That is why a busy web server can have 200,000 connections open while only listening on one port: every client is a different address or a different client port.
8. A server binding to a port below 1024 traditionally needs administrator rights on Unix systems. That is a convention from the 1980s designed to stop ordinary users impersonating system services.
9. When the router rewrites a port, it must pick one that is not already used for another conversation to the same destination. Its allocator does that check on every new flow.

■ 24.7.5 TECHNICAL – the engineer's version

1. Port numbers are 16-bit unsigned integers, 0 to 65535, defined separately for each transport protocol. TCP port 443 and UDP port 443 are different registry entries and different listening sockets.
2. The three ranges are defined in RFC 6335, August 2011, which is BCP 165. The official names are System Ports, User Ports and Dynamic Ports; the informal names are Well Known, Registered and Private or Ephemeral.
3. Operating systems do not agree on the ephemeral range, and this is an implementation detail, not a standard:

System	Default ephemeral range
macOS and FreeBSD	49152 to 65535
Linux	32768 to 60999

System	Default ephemeral range
Windows Vista onward	49152 to 65535
Windows XP and earlier	1025 to 5000

4. On Linux the range is readable and writable at `/proc/sys/net/ipv4/ip_local_port_range`. On macOS it is `sysctl net.inet.ip.portrange.first` and `.last`.
5. Port 0 is reserved and must not be used on the wire. Binding to port 0 in a socket API is a request for the kernel to allocate an ephemeral port, which is a different meaning of the same number.
6. The theoretical connection limit. For a single client address talking to a single server address and port, the only free field is the client port, so the ceiling is 65,536 minus reserved values, roughly 64,500 simultaneous connections. If the operating system uses the standard ephemeral range only, the practical ceiling is about 16,384 on macOS and about 28,000 on Linux.
7. For NAT, the same arithmetic applies to the public address. One public address can support roughly 64,500 simultaneous flows to any one destination endpoint, but many times that across different destinations, because the destination fields also vary.
8. In carrier practice, providers plan on a sharing ratio rather than the theoretical limit. Published deployments have used anywhere from 4 to over 100 subscribers per public address, with per-subscriber port quotas of a few hundred to a few thousand. RFC 6888 requires the ability to set such a quota precisely so one subscriber cannot exhaust the pool.
9. Port exhaustion is a real production failure mode. Its symptoms are new connections failing while existing ones work, and on Linux a rising count of sockets in `TIME_WAIT`, which holds a port for 2 times the maximum segment lifetime, usually 60 seconds total.
10. Observation commands:

```
lsof -i -nP | head           # macOS: which program owns a port
netstat -an | grep 443       # all conversations on port 443
sysctl net.inet.ip.portrange.first net.inet.ip.portrange.last
ss -tan state time-wait | wc -l # Linux: count TIME_WAIT sockets
```

■ 24.7.6 WORDS – remember these

1. Port — a number identifying one conversation on a machine — a 16-bit field in the TCP or UDP header, first four bytes of that header.
2. Well-known port — a published number a server listens on — System Ports 0 to 1023, assigned by IANA under RFC 6335, BCP 165.
3. Ephemeral port — a temporary number a client borrows — Dynamic Ports 49152 to 65535 by IANA, with operating system defaults that differ.
4. Socket — one end of a conversation — the pairing of an address and a port, plus a protocol, exposed as a file descriptor by the operating system.
5. Port exhaustion — running out of conversation numbers — depletion of the available ephemeral or NAT external port pool, causing new connections to fail while old ones survive.
6. `TIME_WAIT` — a closed connection still holding its number — the TCP state that retains the 4-tuple for twice the maximum segment lifetime.

24.8 What NAT breaks

■ 24.8.1 PLAIN – in simple words

1. NAT works beautifully in one direction and badly in the other.
2. Outward is fine. You start the conversation, the router makes a table row, replies find their way home.
3. Inward is the problem. Nobody outside can start a conversation with you.
4. When an unexpected packet arrives, the router looks in its table, finds nothing, and has no way to guess which of your devices it was for. So it drops it.
5. This means you cannot run a server at home that other people reach, unless you configure the router specially.
6. That special configuration is called **port forwarding**: a permanent rule saying “anything arriving on port 8080 goes to 192.168.0.24”.
7. It also means two people both behind NAT cannot simply call each other. Neither can start the conversation, because both are unreachable from outside.
8. Video calls, online games and file sharing all hit this. They solve it with helper servers and a trick called hole punching.
9. Some older protocols carry addresses inside their own messages. NAT rewrites the header but not the message, so those protocols break in confusing ways.
10. The internet was originally designed so that any machine could talk to any other machine directly. NAT ended that. It is a workaround that became permanent.

■ 24.8.2 PLAIN – a picture in your head

1. Go back to the hotel post room from section 24.6.
2. A guest writes out and gets a reply. That works, because the ledger has the reference number.
3. Now imagine somebody outside wants to write to the guest first, without being asked. They know the hotel address. They do not know the guest is there, and they certainly do not know any reference number.
4. Their letter arrives with nothing useful on it. The post room genuinely cannot deliver it. There are four hundred rooms.
5. Worse: two guests in two different hotels want to write to each other. Each can write out. Neither can be written to. Their letters cross in the post and both land in bins.
6. The only fix is a third party. Both guests write to a shared friend who already has a public, reachable address. The friend tells each one the other’s hotel address and reference number, at the same moment, so both send first and both ledgers get their rows.

Where this comparison breaks: this is where the comparison stops being a comparison and becomes literally true. The shared friend is a STUN server, the simultaneous sending is hole punching, and if it fails, the friend physically relays every letter, which is a TURN server. The technical block names them.

■ 24.8.3 PLAIN – a worked example

1. Say the reader wants to run a small web server on the laptop at 192.168.0.24, listening on port 8080, and let a friend see it.
2. The friend types the public address into a browser. The packet arrives at the router addressed to port 8080.
3. The router checks its table. There is no row for port 8080. It drops the packet. The friend sees a connection timeout with no error message. Silence.
4. The fix is a port forwarding rule on the router:

Public port	Internal address	Internal port
8080	192.168.0.24	8080

5. That rule is a destination NAT entry, permanently installed rather than created by traffic. Now the packet is rewritten and delivered.
6. But note what you have just done: you have permanently allocated one public port to one machine. You cannot forward port 8080 to two laptops.
7. Now the peer-to-peer case. Two friends both behind home NAT want a video call.
8. Each contacts a public helper server and asks “what address and port do you see me as”. The helper answers with the outside view, which is the router’s public address and the chosen external port.
9. Each sends that answer to the other, through a signalling channel that both can already reach.
10. Then both send a packet to the other at the same moment. Each outgoing packet creates a table row in its own router. The incoming packet then matches a row that already exists, so it is delivered.
11. That is hole punching. Both sides must send first, which is why it needs careful timing and a coordinator.
12. If the routers are too restrictive for that to work, both sides give up and send everything through a relay server instead. That works always, but costs the relay operator bandwidth, and adds delay.

■ 24.8.4 PLAIN – what is really happening inside

1. The deep problem is that NAT breaks an assumption the internet was built on.
2. The original assumption is that an address means the same thing everywhere. The address you have is the address others use to reach you.
3. Under NAT that is false. You have one address; the world sees another. There is no way for your machine to know its own public address without asking somebody outside.
4. Any protocol that puts an address inside its own message, rather than only in the header, is therefore broken by NAT.
5. Old-style FTP does exactly that. In active mode the client sends its own address and port as text inside a command, and asks the server to connect back. Under NAT the address it sends is private and useless.
6. Telephony signalling has the same shape. It negotiates media addresses inside the message body, and those addresses are private.
7. The workaround is an **ALG**, an Application Layer Gateway: the router inspects the payload, finds the address written in text, and rewrites that too.
8. That is a genuinely ugly solution. The router now has to understand every application protocol, and it breaks when the protocol changes or is encrypted.
9. Encryption killed the ALG approach for good. If the payload is encrypted, no middle box can rewrite it. This is one reason modern protocols never put addresses in the payload.
10. There is a second cost that is easy to miss. NAT makes the network stateful. A router that only forwards can be rebooted or replaced and traffic continues. A router holding NAT state loses every conversation when it restarts.

■ 24.8.5 TECHNICAL – the engineer’s version

1. The design principle NAT violates is stated in “End-to-End Arguments in System Design” by Jerome Saltzer, David Reed and David Clark, presented in 1981 and published in ACM Transactions on Computer Systems in November 1984. The argument is that functions should live at the endpoints, and the network should stay simple.

2. NAT puts per-conversation state in the middle of the network. That is precisely what the end-to-end argument says not to do.
3. The NAT traversal stack, with current documents:

Protocol	Job	Current RFC
STUN	discover your outside view	RFC 8489, Feb 2020
TURN	relay when nothing else works	RFC 8656, Feb 2020
ICE	try candidates, pick one	RFC 8445, July 2018

4. RFC 8489 obsoletes RFC 5389 from October 2008, which in turn replaced RFC 3489 from March 2003. RFC 8656 obsoletes RFC 5766 from February 2010. RFC 8445 obsoletes RFC 5245 from 2010.
5. ICE gathers three kinds of candidate address: host candidates from local interfaces, server-reflexive candidates learned from STUN, and relayed candidates allocated on a TURN server. It then runs connectivity checks in priority order and uses the best pair that works.
6. In practice a WebRTC video call in a browser is running exactly this machinery. Roughly speaking, most calls succeed with a direct or server-reflexive path, and a minority fall back to TURN relay. The exact fallback rate depends heavily on the network population being measured, so treat any single published percentage with caution.
7. Protocols historically needing an ALG: FTP in active mode, where RFC 959's PORT command carries an address as ASCII text; SIP, where addresses appear in the SDP body; H.323; and some peer-to-peer file sharing protocols.
8. IPsec is a specific casualty. Authentication Header protects the IP header including addresses, so any NAT rewrite invalidates it. The workaround is NAT Traversal, RFC 3947 and RFC 3948, January 2005, which wraps ESP inside UDP on port 4500.
9. UPnP IGD and NAT-PMP, and their successor Port Control Protocol in RFC 6887, April 2013, let an internal device ask the router to open a mapping. They work, and they are also a real security concern, since any program on the network can open a hole without asking a human.
10. The honest verdict, stated plainly: NAT is a workaround. RFC 1631 said so in 1994, calling it a short-term solution while a longer-term one was developed. The longer-term solution was IPv6. Thirty years later the short-term measure is still carrying most of the world's traffic. Experts do disagree about whether that is a tragedy or a pragmatic success; the engineering costs listed above are not disputed by either side.

■ 24.8.6 WORDS – remember these

1. Port forwarding — a permanent rule sending one outside port to one inside machine — a static destination NAT entry.
2. Hole punching — both sides sending first so both NAT tables get a row — the simultaneous-open technique used by ICE.
3. STUN — asking a public server what address it sees you as — Session Traversal Utilities for NAT, RFC 8489.
4. TURN — routing everything through a public relay when nothing else works — Traversal Using Relays around NAT, RFC 8656.
5. ICE — trying every possible path and using the best one — Interactive Connectivity Establishment, RFC 8445.
6. ALG — a router that reads and rewrites inside the message — Application Layer Gateway, needed by FTP and SIP, defeated by encryption.

7. End-to-end principle — keep the clever parts at the edges — the design argument of Saltzer, Reed and Clark, 1981 and 1984.
8. Stateful — the box in the middle remembers your conversation — connection state held outside the endpoints, lost on restart.

24.9 Carrier-grade NAT

■ 24.9.1 PLAIN - in simple words

1. You already know that your home router shares one public address between all your devices.
2. Carrier-grade NAT is the same idea, one level up. Your internet provider shares one public address between many customers.
3. They do it for exactly the same reason you do. They have run out of public addresses.
4. So instead of giving your router a public address, they give it a private one, and translate again at a big machine in their own network.
5. Your traffic is now translated twice: once by your router, once by theirs. The usual name for that is double NAT.
6. From your side, nothing looks different for ordinary browsing. Pages load. Video plays.
7. From your side, several things become impossible or unreliable. You cannot accept incoming connections at all, not even with port forwarding, because the outer translation is not yours to configure.
8. You also share a public address with strangers. If one of them misbehaves and a website blocks that address, you are blocked too.
9. Diagnosing faults becomes harder, because there is a machine in the path you cannot see and cannot ask questions of.
10. The reader's own traceroute shows private addresses for five of the first six hops. That is the fingerprint of a provider running a large private core, and it is consistent with carrier-grade NAT.

■ 24.9.2 PLAIN - a picture in your head

1. Go back to the hotel one final time, and add a floor.
2. Your flat is a hotel with extensions. Fine.
3. Now the whole street is served by one switchboard, and your hotel's "main number" turns out not to be a real public number at all. It is an extension on the street switchboard.
4. Outsiders dialling the street switchboard reach an operator who has no idea which hotel, let alone which room, they want.
5. You can still call out. You always could. Calling in is now doubly impossible.
6. And if a neighbour on your street makes nuisance calls, the number that gets barred is the street's number, which is also yours.

Where this comparison breaks: a street switchboard would be an obvious, visible thing. Carrier-grade NAT is invisible to customers. Providers rarely announce it, there is no light on your router for it, and the only way to find out is the test in section 24.13.

■ 24.9.3 PLAIN - a worked example

1. Here is the reader's own first six hops again, exactly as recorded.

1	192.168.0.1
2	172.31.0.17
3	137.97.29.249
4	172.26.22.235

```

5 172.16.18.33 / 172.26.14.75 / 172.26.22.234
6 172.26.14.75 / 172.16.18.33

```

2. Hop 1 is the reader's own router, inside the flat, 192.168.0.1, private.
3. Hop 2 is 172.31.0.17. Second octet 31, inside the 16 to 31 range of 172.16.0.0/12. Private. This is the provider's first router, not the reader's.
4. Hop 3 is 137.97.29.249. Not in any private block. Public.
5. Hops 4, 5 and 6 are all in 172.16.0.0/12. Private again.
6. So the pattern is: private, private, public, private, private, private.
7. That is not a mistake and not a loop. It is what a provider network looks like when most internal links are numbered from private space, but some equipment has a public address on an interface.
8. Now the honest part, and it matters more than anything else in this section.
9. A traceroute shows the address of the interface that generated each reply. It does not show what address your packet was carrying at that point.
10. So the private addresses at hops 2, 4, 5 and 6 prove one thing: the provider numbers those internal links from RFC 1918 space.
11. They do not, on their own, prove that your traffic was translated to a shared public address. That is strongly suggested, not proven.
12. What would prove it is a two-line test, given in full in section 24.13: compare the address on your router's outside interface with the address a public website reports. If they differ, there is a translation between you and the internet that is not yours.

■ 24.9.4 PLAIN – what is really happening inside

1. A carrier-grade NAT is a large, expensive box, or a software function on a large router, sitting between the provider's access network and its public edge.
2. It holds the same kind of table as your home router, but with millions of rows instead of thousands.
3. Each customer is allocated a slice of the port space on a shared public address. A common design gives each subscriber a block of a few thousand ports.
4. Giving each subscriber a contiguous block, rather than picking ports one at a time, is a deliberate choice. It makes logging vastly cheaper: one log line per subscriber per session instead of one per connection.
5. Logging matters because law enforcement requests name an address, a port and a time, and the provider must be able to say which customer that was.
6. What your packet actually experiences, going out:

```

laptop 192.168.0.24
-> home router, source becomes the WAN address
-> provider access network (private numbering)
-> carrier NAT, source becomes a shared public address
-> provider public edge
-> the rest of the internet

```

7. Coming back, the reverse happens, twice, and both tables must still hold the row. If either times out, the connection dies.
8. Because the outer table is not yours, you cannot create a permanent inbound rule in it. Port forwarding on your own router still works for traffic that reaches your router, but nothing from the internet reaches your router unasked in the first place.
9. Where hop 3 fits: a public address on a provider router interface tells you that at least that interface is globally addressable. It marks a place where the provider's equipment touches public space. It

does not by itself mark the exact point where your source address was rewritten, and it would be overclaiming to say that it does.

■ 24.9.5 TECHNICAL - the engineer's version

1. Carrier-grade NAT is also called CGN, large-scale NAT or LSN. The requirements document is RFC 6888, "Common Requirements for Carrier-Grade NATs", April 2013, which is BCP 127.
2. RFC 6598, April 2012, BCP 153, reserves 100.64.0.0/10 as Shared Address Space for exactly this purpose: numbering the link between the CGN and customer premises equipment. Its existence is a direct admission that carriers were already using RFC 1918 space and colliding with customers' home networks.
3. RFC 6264, June 2011, describes the incremental CGN approach for IPv6 transition. RFC 6598's own justification is the collision problem.
4. Key RFC 6888 requirements worth knowing: paired address pooling, meaning all sessions from one internal address must use the same external address; a configurable per-subscriber port limit; and a minimum 120-second delay before an external port is reused.
5. Deployment forms include NAT444, where there are three address realms and two translations, and DS-Lite, RFC 6333, August 2011, which tunnels IPv4 over IPv6 to a carrier translator called an AFTR. MAP-E and MAP-T, RFC 7597 and RFC 7599, both July 2015, are stateless alternatives.
6. What CGNAT costs the customer:

Effect	Consequence
No unsolicited inbound	cannot host anything
Port forwarding useless	your rule stops at you
Shared reputation	blocked for a stranger
CAPTCHA and rate limits	per-IP limits hit sooner
Harder diagnosis	an invisible stateful hop
Game and call quality	strict NAT, more TURN relaying
Geolocation errors	mapped to the wrong city

7. The shared reputation problem is the one users notice most. Per-address rate limiting, abuse blocklists and anti-fraud scoring all assume one address is roughly one customer. Under CGNAT that assumption is simply false, and it can put hundreds of households behind one score.
8. Interpreting the reader's trace precisely. Proven by the data: hops 2, 4, 5 and 6 are RFC 1918 addresses, so those provider interfaces are privately numbered; hop 3 at 137.97.29.249 is a public address on a provider interface. Suggested but not proven: that the reader's own traffic is subject to carrier-grade translation. Not shown at all: which specific device performs any translation, or where exactly it sits.
9. Hops 5 and 6 listing the same two addresses in different order, 172.16.18.33 and 172.26.14.75, is the signature of equal-cost multi-path forwarding, where consecutive probes take different parallel links. It is normal. Paris traceroute, published in 2006 by Brice Augustin and colleagues, exists specifically to keep the flow identifier constant so that this artefact disappears.
10. Definitive detection, not inference: read the WAN address from the router's own status page or with a UPnP query, then compare it with what a public reflector reports. Disagreement means an outer NAT. Both being equal and public means no CGNAT.

```
# what the internet sees
curl -s https://api.ipify.org; echo
# an alternative that uses DNS instead of HTTP
```

```
dig +short myip.opendns.com @resolver1.opendns.com
# then compare with the WAN address shown by the router itself
```

■ 24.9.6 WORDS – remember these

1. Carrier-grade NAT — your provider sharing one public address between many customers — CGN or LSN, specified for behaviour by RFC 6888, BCP 127.
2. Double NAT — two translations between you and the internet — NAT444, three address realms with a translation at each boundary.
3. Shared address space — the block used between the carrier NAT and your router — 100.64.0.0/10 per RFC 6598, BCP 153.
4. Port block allocation — giving each customer a fixed slice of port numbers — deterministic port assignment, which reduces logging volume enormously.
5. Shared reputation — being punished for a stranger on the same address — the collapse of the one-address-one-subscriber assumption used by abuse scoring.
6. ECMP — several equally good parallel paths — equal-cost multi-path forwarding, the reason one traceroute hop can show several addresses.
7. DS-Lite — carrying IPv4 over an IPv6-only access network — Dual-Stack Lite, RFC 6333, with translation at a carrier AFTR.

24.10 Reading the reader's own first six hops as a story

■ 24.10.1 PLAIN – in simple words

1. A traceroute is a list of the machines that handled your packet on the way out, in order.
2. It works by sending packets with a deliberately short lifespan. The first has a lifespan of one hop, so the first router kills it and sends back a complaint. The complaint reveals that router's address.
3. Then it sends one with a lifespan of two, and the second router complains. And so on.
4. So each line of a traceroute is one router saying "I am here, and your packet died on my desk".
5. The reader's first six hops tell a clear story: out of the flat, into the provider, through the provider's private core, with one public address appearing in the middle.
6. From hop 7 onwards the names change completely and the path enters Microsoft's own network, which is where GitHub lives.
7. This section reads only the first six, because those six are entirely about addressing, which is this chapter's subject.

■ 24.10.2 PLAIN – a picture in your head

1. Imagine posting a letter with an instruction on the envelope: "after passing through one sorting office, stop and send this envelope back to me with the office's stamp on it".
2. You get back an envelope stamped by your local office. Now you know the first step of the route.
3. Post another saying "stop after two offices". Back it comes, stamped by the regional depot. Now you know the second step.
4. Repeat twenty times and you have mapped the whole postal route without ever delivering a real letter.
5. Some offices refuse to send anything back. Those steps come back blank. It does not mean the office is broken; it means it does not answer surveys.

Where this comparison breaks: postal routes are stable, so twenty separate letters follow the same path. Network paths are not. Each probe can take a different parallel link, which is exactly why hops

5 and 6 of the reader's trace show several addresses each. A traceroute is a rough sketch made from twenty different journeys, not a photograph of one.

■ 24.10.3 PLAIN – a worked example

- Here is the reader's own trace, hop by hop, classified.

Hop	Address	Type
1	192.168.0.1	private, 192.168.0.0/16
2	172.31.0.17	private, 172.16.0.0/12
3	137.97.29.249	public
4	172.26.22.235	private, 172.16.0.0/12
5	172.16.18.33 and two more	private, 172.16.0.0/12
6	172.26.14.75 and one more	private, 172.16.0.0/12

- And what each one means.

Hop	What it means
1	the reader's own router
2	the provider's first router
3	a provider public interface
4	provider core, private
5	parallel links, load balanced
6	the same parallel links again

- Hop 5 listed 172.16.18.33, 172.26.14.75 and 172.26.22.234. Hop 6 listed 172.26.14.75 and 172.16.18.33.
- The same addresses appear at two consecutive hops. That looks alarming and is not. It is what happens when three probes for hop 5 and two probes for hop 6 each pick a different parallel link.
- Notice also that two different private blocks are in use: 172.16.x, 172.26.x and 172.31.x. All three are inside 172.16.0.0/12. A large provider commonly divides that /12 by region or by role.
- The shape of the path in one picture:

flat	provider access	provider core	public
+-----+	+-----+	+-----+	+-----+
laptop	192.168.0.1	172.31.0.17	hop 7
192.	hop 1	172.26.x	msn.net
168.0.x	private	172.16.18.33	public
+-----+	+-----+	all private	+-----+
		137.97.29.249	
		one public	
		+-----+	

■ 24.10.4 PLAIN – what is really happening inside

- The lifespan number on the packet is called TTL, Time To Live. Every router subtracts one before forwarding.
- When a router subtracts one and gets zero, it must discard the packet and send back an ICMP time-exceeded message.
- That reply comes from whichever interface address the router chooses, normally the one closest to you. That is why the addresses you see are the routers' inbound interfaces.

4. It also means a private address in a trace is a router's own interface, not your packet's source. This is the point that section 24.9 rested on.
5. Not every router replies. Some are configured not to. Some rate-limit these replies to a handful per second, which is why long traces show occasional gaps in the middle.
6. Silence at the end of a trace, which the reader saw at hops 13 to 20 as three asterisks each, is normal and proves nothing on its own.
7. The reason the reader's trace still mattered is what it showed at the start, not what it failed to show at the end: a provider with a large private core, one public interface early on, and parallel links.

■ 24.10.5 TECHNICAL – the engineer's version

1. Traceroute was written by Van Jacobson in 1987 at Lawrence Berkeley National Laboratory. The behaviour it depends on, the ICMP time-exceeded reply, is in RFC 792, September 1981.
2. Classic Unix traceroute sends UDP datagrams to high, unlikely destination ports, incrementing the TTL. macOS traceroute does this by default and needs no privilege for that mode.
3. Windows tracert sends ICMP echo requests instead. This matters, because networks often treat ICMP and UDP differently, so the two tools can produce different paths on the same network.
4. `traceroute -I` uses ICMP on macOS and requires root. `traceroute -T` on Linux uses TCP SYN, which is often the only method that survives firewalls that discard everything else.
5. By default traceroute sends three probes per TTL value, which is why three addresses can appear on one line and why unanswered hops print three asterisks.
6. Multiple addresses on one hop indicate ECMP load balancing, typically hashed per flow on the 5-tuple. Classic traceroute varies the destination port between probes, which changes the hash, which is why the probes diverge. Paris traceroute, presented in 2006 by Brice Augustin and colleagues, holds the flow identifier constant to avoid this.
7. Reverse DNS is why hops 7 onward in the reader's trace showed names such as `ae66-0.del01-96cbe-1b.ntwk.msn.net` while hops 1 to 6 showed none. Private addresses have no public reverse record, and many carriers do not publish names for internal interfaces.
8. The naming on those later hops is a Microsoft convention, not a standard. The city codes are airport-style: `del01` Delhi, `bom01` Mumbai, `pnq20` and `pnq21` Pune. The role codes `ibr`, `rwa` and `owr` mean internal backbone, regional wide-area and outer wide-area router respectively, again by that operator's convention. Interface prefixes `ae`, `be` and `po` all mean several physical links bonded into one logical link; different vendors use different words for the same idea.
9. Useful invocations:


```
traceroute -n 20.207.73.82      # numeric only, no reverse DNS
traceroute -I -n 20.207.73.82  # ICMP probes, needs sudo on macOS
traceroute -q 1 -n 20.207.73.82 # one probe per hop, less noise
mtr -n 20.207.73.82           # continuous, shows loss per hop
```
10. The strongest reading discipline: separate what a trace proves from what it suggests. It proves which interfaces answered. It suggests, but never proves, the forward path, and it says nothing at all about the reverse path, which can be completely different.

■ 24.10.6 WORDS – remember these

1. Traceroute — a tool that lists the routers on the way — a TTL-incrementing probe tool relying on ICMP time-exceeded replies, RFC 792.
2. TTL — the countdown that stops packets looping forever — Time To Live, an 8-bit IPv4 header field decremented at each hop.
3. ICMP time-exceeded — the complaint a router sends when a packet expires — ICMP type 11, the message that makes traceroute possible.
4. Hop — one router along the path — one decrement of TTL.
5. Asterisk in a trace — no reply came back — an unanswered probe, commonly caused by ICMP rate limiting or policy, not by a fault.
6. Reverse DNS — turning an address back into a name — a PTR record lookup in the in-addr.arpa zone.

24.11 Static and dynamic addresses, and how yours is chosen

■ 24.11.1 PLAIN – in simple words

1. Nobody typed 192.168.0.x into the reader's laptop. It was handed over automatically.
2. The thing that hands it over is called DHCP. Your laptop shouts "does anyone have an address for me", and the router answers.
3. The router does not give the address away permanently. It lends it, for a period called a **lease**.
4. Before the lease runs out, your machine asks to keep it, and normally gets the same one back. That is why your laptop's address usually looks stable.
5. The same thing happens one level up. Your router asks your provider for a public address, and gets one on loan.
6. That loan can end. When it does, you may get a different public address. This is why home connections are said to have a **dynamic** address.
7. A **static** address is one that never changes, arranged deliberately and usually paid for.
8. If you want to run something at home that people can find by name, and your address keeps changing, you need a service that updates the name every time the address moves. That is dynamic DNS.
9. Businesses pay for static addresses because servers, mail and remote access all break when the address moves.

■ 24.11.2 PLAIN – a picture in your head

1. Think of a library that lends reading-room desks rather than selling them.
2. You arrive, ask for a desk, and are given number 24 for four hours.
3. Nobody else gets desk 24 while your loan runs.
4. Halfway through, at two hours, you are asked whether you want to keep it. You say yes and the four hours restart. You keep the same desk.
5. If you leave without saying anything, the desk goes back into the pool once the four hours are up, and someone else may get it.
6. A subscriber who pays extra gets a reserved desk with their name on it, every day, guaranteed. That is a static address.

Where this comparison breaks: the library knows you by face. DHCP identifies you by a hardware identifier your machine sends, and modern phones and laptops deliberately randomize that identifier for privacy. So your own device may be handed a different address on purpose, on a network it has used for years, and that is a feature rather than a fault.

■ 24.11.3 PLAIN – a worked example

1. The four steps of DHCP, in order, with the names engineers use.

Step	Who sends	Meaning
DISCOVER	the laptop	anyone out there
OFFER	the router	you can have this one
REQUEST	the laptop	yes, I take it
ACK	the router	confirmed, here are details

2. The first message goes from source 0.0.0.0 to destination 255.255.255.255, because the laptop has no address and does not know the server's address.
3. The ACK carries more than an address. It typically carries the mask, the gateway, the DNS servers and the lease time.
4. For the reader that would be: address 192.168.0.x, mask 255.255.255.0, gateway 192.168.0.1, and a DNS server.
5. Note a detail from the reader's own session: their DNS resolver was 1.1.1.1, Cloudflare's public resolver, not the router. So either the router was configured to hand out 1.1.1.1, or the laptop was set manually and ignored what DHCP offered. DHCP suggests; the operating system may override.
6. On macOS you can see the whole lease, raw:

```
ipconfig getpacket en0
# yiaddr = 192.168.0.24
# subnet_mask (ip): 255.255.255.0
# router (ip_mult): {192.168.0.1}
# lease_time (uint32): 0x15180      (86400 seconds = 24 hours)
```

7. yiaddr is the field name in the protocol and means "your address". It has been called that since 1993.
8. For the public side, many home connections in India and elsewhere use PPPoE: the router logs in with a username and password, and the address is negotiated as part of that session rather than by DHCP.
9. With PPPoE, the address usually changes when the session drops and reconnects, which is why rebooting a router often changes your public address, and sometimes does not.

■ 24.11.4 PLAIN – what is really happening inside

1. A DHCP lease has three timers, not one.
2. The full lease time is how long the address is valid.
3. At half the lease time, the client tries to renew directly with the server that gave it the address. This is called T1.
4. At seven-eighths of the lease time, if renewal failed, the client broadcasts to any server that will listen. This is called T2.
5. If nothing answers by the end, the client must stop using the address. In practice most operating systems then fall back to a link-local 169.254.x.x address, which is the symptom described in section 24.4.
6. Because renewal starts at half the lease, a machine that stays switched on keeps the same address indefinitely. Addresses only really move when a machine is away long enough for the lease to expire and be reissued.

7. On the public side with PPPoE, the address is negotiated by a sub-protocol during session setup, not by DHCP at all. The provider's access concentrator picks an address from its pool and offers it.
8. Providers vary in how sticky that is. Some effectively give the same address for months. Some rotate it every 24 hours deliberately.
9. Under carrier-grade NAT the question changes shape entirely. Your router's outside address may be stable while the public address the world sees changes whenever the carrier NAT reassigns you.

■ 24.11.5 TECHNICAL - the engineer's version

1. DHCP is RFC 2131, March 1997, with options in RFC 2132. It extends BOOTP from RFC 951, September 1985, and reuses its packet format, which is why the field is still called yiaddr.
2. The client sends from UDP port 68 to UDP port 67. The exchange is DHCPDISCOVER, DHCPOFFER, DHCPREQUEST, DHCPACK, commonly abbreviated DORA.
3. Renewal timers default to T1 at 0.5 of the lease and T2 at 0.875 of the lease, per RFC 2131. Both can be set explicitly with options 58 and 59.
4. Typical lease durations by deployment:

Environment	Typical lease
Home router	24 hours
Corporate wired	8 days
Public Wi-Fi, cafe	1 to 2 hours
Conference network	10 to 30 minutes

5. Client identity is normally the MAC address in the chaddr field, optionally overridden by option 61. Since Android 10 in 2019 and iOS 14 in 2020, and in recent macOS releases on the desktop, devices rotate a randomized MAC per network by default. Address reservations keyed on MAC therefore need the randomization turned off for that network.
6. A DHCP reservation, sometimes called static DHCP, pins one address to one client identifier. It is not the same as a static address configured on the host: the client still leases, it simply always receives the same value.
7. PPPoE is RFC 2516, February 1999. It carries PPP over Ethernet; the address is assigned during the IPCP negotiation phase of PPP, from RFC 1332. The provider side is a BRAS or BNG, a broadband remote access server or broadband network gateway.
8. Dynamic DNS updates are standardized in RFC 2136, April 1997. Consumer services predate and largely ignore it, using proprietary HTTP endpoints instead; most home routers have built-in support for two or three of them. Practical caveat: dynamic DNS is useless behind carrier-grade NAT, because the name will resolve to an address that will not accept your inbound traffic.
9. Static addresses cost money for a reason that has nothing to do with technology. They consume a scarce public address permanently, whereas dynamic pools let a provider serve more subscribers than it has addresses, because not all are online at once. That over-subscription is the whole economic point.
10. Observation commands:

```
ipconfig getpacket en0      # the full DHCP lease on macOS
ipconfig getoption en0 lease_time # just the lease duration
sudo ipconfig set en0 DHCP    # force a fresh DHCP exchange
netstat -rn | grep default    # the gateway the lease installed
```

■ 24.11.6 WORDS – remember these

1. DHCP — the system that hands out addresses automatically — Dynamic Host Configuration Protocol, RFC 2131, UDP ports 67 and 68.
2. Lease — a borrowed address with an expiry — a time-bounded binding, renewed at T1 and re-bound at T2.
3. DORA — the four-step handshake — DISCOVER, OFFER, REQUEST, ACK.
4. Static address — one that never changes — a manually configured or permanently allocated address, usually a paid service.
5. Dynamic address — one that can change — an address from a provider pool, held only for the duration of a lease or session.
6. Reservation — always getting the same address from DHCP — a static mapping from client identifier to address inside the DHCP server.
7. PPPoE — logging in to your broadband with a username and password — PPP over Ethernet, RFC 2516, with addressing negotiated by IPCP.
8. Dynamic DNS — a name that follows your changing address — automated updating of an A record, standardized as RFC 2136, defeated by carrier-grade NAT.

24.12 IPv6 in brief

■ 24.12.1 PLAIN – in simple words

1. IPv6 is the replacement for the addressing scheme this whole chapter has described. It uses 128 bits instead of 32.
2. Four times the bits does not mean four times the addresses. Each extra bit doubles the count. So 128 bits gives about 340 undecillion addresses, a 3 followed by 38 digits.
3. That is enough to give every grain of sand on earth its own network, with room left over. The size is deliberate and slightly absurd, so that nobody ever has to do this again.
4. Because the numbers are so long, they are written in hexadecimal, which uses the digits 0 to 9 and the letters a to f, in eight groups separated by colons.
5. There are two shortening rules, and only two, so the addresses stay readable.
6. IPv6 exists because IPv4 ran out, exactly as section 24.5 described.
7. With enough addresses for everyone, NAT is no longer needed. Every device can have its own real address again.
8. The reader's own machine reported IPv6 as (none). That means their connection offered no IPv6 at all, so every packet in this chapter's story travelled over IPv4, through the provider's private core and its translation.
9. Chapter 34 covers IPv6 properly. This section is only enough to place it in the story of addressing.

■ 24.12.2 PLAIN – a picture in your head

1. Think of telephone numbers being replaced by a scheme with room for every person, object and appliance on the planet.
2. The new numbers are longer, so nobody memorizes them. You rely on your contacts list. In networking that contacts list is DNS.
3. The old and the new cannot dial each other. A new-scheme phone cannot ring an old-scheme phone at all.
4. So during the change, every phone must be able to use both schemes. That is dual stack.
5. Nobody wants to be first, because being first costs money and buys nothing until others follow. Everyone wants to be second.

Where this comparison breaks: phone numbers could be translated between schemes by an exchange, and countries did exactly that when they lengthened numbers. IPv4 and IPv6 are genuinely different protocols with different header layouts, and translation between them is lossy and awkward. That incompatibility, not laziness, is the main reason the change has taken three decades.

■ 24.12.3 PLAIN – a worked example

1. Full form, all 32 hex digits written out:

```
| 2001:0db8:0000:0000:0000:ff00:0042:8329
```

2. Rule one: drop leading zeros inside each group. Not all zeros, only leading ones.

```
| 2001:db8:0:0:0:ff00:42:8329
```

3. Rule two: replace the longest run of all-zero groups with a double colon, once and only once in an address.

```
| 2001:db8::ff00:42:8329
```

4. Once only, because two double colons would be ambiguous. A reader could not tell how many zero groups belonged to each.

5. A second example, a link-local address of the kind macOS shows on every interface:

```
| fe80:0000:0000:0000:0204:61ff:fe9d:f156
| fe80::204:61ff:fe9d:f156
```

6. Note that 0204 became 204, losing only the leading zero, and that 61ff was untouched because its zeros are not leading.

7. Two special short ones:

Meaning	Full	Short
Loopback	0000: ... :0001	::1
Unspecified	0000: ... :0000	::

8. The main address types, by their leading bits:

Type	Prefix	Use
Global unicast	2000::/3	routable on the internet
Link-local	fe80::/10	this link only, always present
Unique local	fc00::/7	private, like RFC 1918
Multicast	ff00::/8	groups
Documentation	2001:db8::/32	books and manuals

9. Every IPv6 interface has a link-local address whether or not it has anything else. That is required, not optional, and it is what neighbour discovery runs over.

■ 24.12.4 PLAIN – what is really happening inside

1. IPv6 does not use DHCP by default. A machine can configure itself.
2. It works like this. The machine sends a router solicitation on the link. A router replies with a router advertisement carrying a prefix, usually 64 bits long.

3. The machine takes that 64-bit prefix and invents its own 64-bit second half. Joining them gives a complete address.
4. That is called SLAAC, stateless address autoconfiguration. Stateless because no server keeps a record of who has what.
5. Originally the second half was derived from the hardware address, which meant the same machine had a recognizable address on every network it visited. That is a privacy disaster: you could be tracked across cafes and cities.
6. So privacy addresses were added. The machine generates random second halves, uses one for outgoing connections, and replaces it every day or so.
7. This is why a machine with IPv6 usually shows several addresses at once on one interface: a link-local one, a stable one, and one or more temporary ones.
8. During the transition most machines run both protocols side by side. When you open a connection to a name that has both kinds of address, the machine races them and uses whichever answers first.
9. The reader's machine had no IPv6 at all, so none of this applied. Every attempt to reach 20.207.73.82 went over IPv4, and had no alternative path to fall back on.

■ 24.12.5 TECHNICAL – the engineer's version

1. IPv6 is specified in RFC 8200, July 2017, Internet Standard STD 86, which obsoletes RFC 2460 from December 1998. The first specification was RFC 1883, December 1995.
2. The addressing architecture is RFC 4291, February 2006. The canonical text representation is RFC 5952, August 2010.
3. RFC 5952 states the rules as requirements, not preferences: leading zeros must be suppressed; :: must shorten the longest zero run; :: must not be used for a single zero group; on a tie the leftmost run wins; hexadecimal letters must be lowercase.
4. Address space: 2 to the power 128, exactly 340,282,366,920,938,463,374,607,431,768,211,456. Global unicast is currently allocated only out of 2000::/3, which is one eighth of the total, leaving the rest untouched for the future.
5. Unique local addresses are RFC 4193, October 2005. The block is fc00::/7, but in practice only fd00::/8 is used, with 40 random bits chosen by the site so that two organizations merging almost certainly do not collide. That random collision-avoidance is the direct lesson learned from RFC 1918 overlap.
6. SLAAC is RFC 4862, September 2007, using ICMPv6 Neighbor Discovery from RFC 4861. Temporary privacy addresses are RFC 8981, February 2021, which obsoletes RFC 4941 from September 2007.
7. Dual stack is RFC 4213, October 2005. Happy Eyeballs version 2, RFC 8305, December 2017, is the algorithm that races an IPv6 and an IPv4 connection attempt with a short head start for IPv6, and is why a broken IPv6 path no longer stalls a browser for thirty seconds.
8. Header differences that matter in practice: the IPv6 header is a fixed 40 bytes with no options inline and no header checksum; routers never fragment, so path MTU discovery is mandatory; the minimum link MTU is 1280 bytes.
9. Why thirty years:

Reason	Effect
No wire compatibility	v6-only cannot reach v4-only
No first-mover benefit	cost now, benefit later
NAT relieved pressure	the crisis felt survivable

Reason	Effect
Equipment and staff cost	every device and skill set
Dual stack doubles work	two of everything to operate

- Measured adoption is published live by Google as the share of its users reaching it over IPv6. It crossed 10 percent in 2016 and has grown steadily since; because that figure moves every month, look it up rather than trusting a printed number, including this one.
- Observation on macOS: `ifconfig en0 inet6`, `netstat -rn -f inet6`, and `ping6 -c1 ipv6.google.com`. If the last of these fails and `ifconfig` shows only an `fe80::` address, you have link-local IPv6 only and no global connectivity, which is exactly the reader's situation reported as IPv6: (none).

■ 24.12.6 WORDS – remember these

- IPv6 — the 128-bit replacement for IPv4 — Internet Protocol version 6, RFC 8200, STD 86.
- Hextet — one group of four hex digits — a 16-bit field of an IPv6 address, eight per address.
- Zero compression — the double colon shortcut — `::` replacing the longest run of all-zero 16-bit fields, once per address, per RFC 5952.
- Global unicast — an address the internet can route to — currently allocated from `2000::/3`.
- Link-local — the address every IPv6 interface always has — `fe80::/10`, used by neighbour discovery, never routed off the link.
- Unique local — IPv6's version of a private address — `fc00::/7`, used as `fd00::/8` with 40 random bits, RFC 4193.
- SLAAC — a machine configuring its own address from a router hint — stateless address autoconfiguration, RFC 4862.
- Privacy address — a temporary random address that changes daily — temporary address extensions, RFC 8981.
- Dual stack — running both protocols at once — RFC 4213, with connection racing by Happy Eyeballs, RFC 8305.

24.13 Practical: finding your own addresses on macOS

■ 24.13.1 PLAIN – in simple words

- Four facts describe your position on the network: your own address, your mask, your gateway, and your public address.
- Your own address is what your machine is called on the local network.
- Your mask says how big that local network is.
- Your gateway is the router you hand everything else to. For the reader it is 192.168.0.1.
- Your public address is what the rest of the internet sees. It is usually different from your own address, because of NAT.
- Getting all four takes about thirty seconds and four commands.
- Comparing two of them tells you instantly whether your provider is running carrier-grade NAT.

■ 24.13.2 PLAIN – a picture in your head

- Think of checking your own postal details before complaining that a parcel never arrived.
- Your flat number is your local address. The building's street number is your public address. The porter is the gateway. The postcode area is the mask.

3. If someone tells you the parcel was delivered to a street number you have never heard of, then something in the middle is redirecting your post, and you need to know that before you argue with anyone.

Where this comparison breaks: you can walk outside and read your building's street number. You cannot see your public address from inside your own machine at all. You must ask a server on the internet to tell you what it sees. That is not a convenience, it is a fundamental consequence of NAT.

■ 24.13.3 PLAIN – a worked example

1. The four commands, in order.

```
# 1. my own address on Wi-Fi
ipconfig getifaddr en0
# 192.168.0.24

# 2. my mask, and the address again, with more detail
ifconfig en0 | grep 'inet '
# inet 192.168.0.24 netmask 0xffffffff broadcast 192.168.0.255

# 3. my gateway
netstat -rn -f inet | grep default
# default 192.168.0.1 UGScg en0

# 4. what the internet sees
curl -s https://api.ipify.org; echo
```

2. Note that macOS prints the mask in hexadecimal. 0xffffffff00 is 255.255.255.0, which is a /24, which matches section 24.2 exactly.
3. Converting is easy: ff is 255, 00 is 0. So 0xffffffff00 reads as 255, 255, 255, 0.
4. One command gives three of the four at once:

```
networksetup -getinfo "Wi-Fi"
# IP address: 192.168.0.24
# Subnet mask: 255.255.255.0
# Router: 192.168.0.1
```

5. Now the carrier-grade NAT test. Take the public address from step 4. Then open your router's status page and read the address on its outside, or WAN, interface.
6. Decide as follows:

```
Is the router's WAN address in 100.64.0.0/10,
10.0.0.0/8, 172.16.0.0/12 or 192.168.0.0/16 ?
|
+-- yes -> you are behind carrier-grade NAT.
|
+-- no --> is it the same as what curl reported ?
|
+-- yes -> a normal public address, no CGNAT
|
+-- no --> something translates in between:
CGNAT, or a proxy, or a VPN
```

7. Do the VPN check first, because it produces the same symptom. The reader's machine showed several utun interfaces; if one of them is active and carrying the default route, the public address you see is the VPN's, not your provider's.


```
ifconfig | grep -A2 '^utun'      # are any tunnels up
netstat -rn -f inet | grep default # which interface is default
```

■ 24.13.4 PLAIN – what is really happening inside

1. `ipconfig getifaddr` reads the address the operating system has stored for that interface. No packets are sent. It is instant and works offline.
2. `netstat -rn` prints the routing table straight from the kernel. The line marked `default` is the route used for anything not matching a more specific entry.
3. `curl https://api.ipify.org` is completely different in kind. It makes a real connection to a real server on the internet, and that server reports the source address it saw.
4. So that command measures what the world sees after every translation on the path, including your router, any carrier NAT, and any VPN.
5. The DNS-based alternative asks a resolver the same question over DNS rather than HTTP. It is worth knowing because it works when HTTP is blocked but DNS is not, and because it takes a different path.
6. If those two methods disagree, that itself is information: something is treating your web traffic and your DNS traffic differently.
7. None of these commands can see the carrier NAT box itself. You infer it from the difference between two numbers, which is the only tool you have from inside.

■ 24.13.5 TECHNICAL – the engineer's version

1. Command reference, macOS, with what each actually queries:

Command	Source of truth
<code>ipconfig getifaddr en0</code>	configd stored state
<code>ifconfig en0</code>	kernel interface list
<code>netstat -rn -f inet</code>	kernel routing table
<code>route -n get default</code>	kernel route lookup
<code>ipconfig getpacket en0</code>	the stored DHCP lease
<code>networksetup -getinfo</code>	the system preference
<code>curl to a reflector</code>	a live remote observation

2. The interface name is not always `en0`. On many Macs `en0` is Wi-Fi, but on models with Ethernet it may be wired, with Wi-Fi on `en1`. Resolve it properly rather than guessing:

```
networksetup -listallhardwareports
route -n get default | awk '/interface/{print $2}'
```

3. Reflector services worth knowing, all returning plain text: `api.ipify.org`, `ifconfig.me`, `icanhazip.com`. Prefer HTTPS. The DNS method, `dig +short myip.opendns.com @resolver1.opendns.com`, avoids HTTP entirely.
4. To force IPv4 or IPv6 explicitly, use `curl -4` and `curl -6`. On a dual stack host these can legitimately return different addresses, and a machine reporting IPv6: (none) as the reader's did will simply fail on `curl -6`.
5. For a fuller picture, `scutil --nwi` prints macOS network information including which interfaces are considered usable for IPv4 and IPv6, and `arp -an` shows the neighbours your machine has actually spoken to on the local segment, which will include 192.168.0.1.
6. If the WAN address is in 100.64.0.0/10, the diagnosis is certain and needs no further evidence: RFC 6598 space exists only for the link between a carrier NAT and customer equipment.

7. If the WAN address is RFC 1918 space, as the reader's hop 2 at 172.31.0.17 suggests may be the case here, the diagnosis is very likely but not certain, because a modem in bridge or router mode can also produce that pattern locally.
8. Practical consequence to check next: if you are behind carrier-grade NAT, stop trying to fix inbound problems with port forwarding, and ask the provider for a public address, or use an outbound-initiated tunnel service instead.

■ 24.13.6 WORDS - remember these

1. Gateway — the router you hand outbound traffic to — the next hop of the default route, 0.0.0.0/0.
2. Default route — where anything unmatched goes — the least specific routing table entry, always losing to any longer prefix match.
3. WAN address — the address on your router's outside — the address the provider assigned to the customer equipment, by DHCP or PPPoE.
4. Reflector — a server that tells you how you look from outside — an HTTP or DNS service echoing the observed source address.
5. Hexadecimal netmask — the 0xffffffff00 form macOS prints — the same 32-bit mask written in base 16, two digits per octet.
6. scutil — the macOS tool for network state — the System Configuration framework utility, `scutil --nwi` for reachability information.

24.98 Common wrong ideas

1. Wrong: an IP address identifies a person. Right: it identifies a network interface at a moment in time. Under NAT it identifies a whole household; under carrier-grade NAT it can identify hundreds of households at once.
2. Wrong: an IP address identifies a device. Right: it identifies one interface. One machine can hold many addresses at once across Wi-Fi, Ethernet, loopback and every VPN tunnel, and the reader's macOS session showed several `utun` interfaces doing exactly that.
3. Wrong: private addresses are more secure. Right: they are not routable, which is different from being protected. Anything on your own network reaches them freely, and most real attacks arrive through connections you started, which NAT permits by design.
4. Wrong: NAT is a firewall. Right: NAT drops unsolicited inbound traffic as a side effect of not knowing where to send it. A firewall makes deliberate policy decisions, inspects traffic and can be audited. Relying on NAT for security means relying on an accident.
5. Wrong: changing your IP address hides you. Right: it changes one weak identifier. Cookies, browser fingerprints, logins and DNS queries all survive the change, and under carrier-grade NAT your address was never uniquely yours to begin with.
6. Wrong: 172.16.0.0/12 means everything starting 172.16. Right: it covers 172.16.0.0 through 172.31.255.255, which is why the reader's hops at 172.31.0.17, 172.26.22.235 and 172.16.18.33 are all private.
7. Wrong: private addresses in a traceroute prove you are behind carrier-grade NAT. Right: they prove those router interfaces are privately numbered. The proof of carrier-grade NAT is comparing your router's WAN address with what a public reflector reports.
8. Wrong: my router's address 192.168.0.1 clashes with my neighbour's. Right: it cannot, because private addresses are never carried on the public internet. The two networks never meet.
9. Wrong: a hop showing three addresses is an error or a loop. Right: it is equal-cost multi-path load balancing across parallel links, which is exactly what the reader's hops 5 and 6 show.

10. Wrong: IPv6 is just IPv4 with longer addresses, so switching is easy. Right: it is a separate protocol with a different header, and an IPv6-only host cannot reach an IPv4-only host at all. That incompatibility is why the transition has taken three decades.

24.99 Chapter summary in 20 lines

1. An IPv4 address is 32 bits labelling one network interface, not a device and not a person.
2. Written as four decimal octets, giving 2 to the power 32, exactly 4,294,967,296 possible addresses.
3. 192.168.0.1 is 11000000 10101000 00000000 00000001 in binary, and 3232235521 as a single integer.
4. Every address splits into a network part and a host part; the subnet mask says where, and CIDR notation writes the mask as a count of bits.
5. For 192.168.0.0/24 the network is 192.168.0.0, the broadcast 192.168.0.255, the usable range 192.168.0.1 to 192.168.0.254, and the host count 254.
6. RFC 1918, February 1996, BCP 5, reserves 10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16 for private use.
7. 172.16.0.0/12 runs from 172.16.0.0 to 172.31.255.255, so the reader's hops at 172.31.0.17, 172.26.x and 172.16.18.33 are all private addresses.
8. Private addresses never travel on the public internet, which is why every home can use 192.168.0.1 with no conflict.
9. Other reserved blocks include 127.0.0.0/8 loopback, 169.254.0.0/16 link-local, 100.64.0.0/10 shared space for carrier NAT, and 224.0.0.0/4 multicast.
10. The IANA free pool was emptied on 31 January 2011 and declared depleted on 3 February 2011; APNIC, serving India, hit its final block on 15 April 2011.
11. Classful addressing wasted enormous space by offering only three block sizes; CIDR replaced it in 1993 and is now RFC 4632.
12. NAT rewrites the source address on the way out and restores it on the way back, using a table keyed on the 5-tuple.
13. What home routers actually run is PAT, also called NAPT or NAT overload, which rewrites the port as well as the address.
14. A port is a 16-bit number identifying one conversation, ranged 0 to 1023 well-known, 1024 to 49151 registered, 49152 to 65535 ephemeral per RFC 6335.
15. Ports are what let one public address carry tens of thousands of simultaneous conversations; without them NAT would be pointless.
16. NAT breaks inbound connections, needs port forwarding for servers, and forces peer-to-peer software to use STUN, TURN and ICE with hole punching.
17. NAT was proposed in RFC 1631 in 1994 as a short-term measure; it is honestly a workaround, not a design, and it violates the end-to-end principle.
18. Carrier-grade NAT applies the same trick at the provider, costing you inbound connections, a shared reputation and much harder diagnosis.
19. The reader's first six hops read private, private, public, private, private, private, which proves the provider numbers its core from RFC 1918 space and strongly suggests, without proving, carrier-grade translation.
20. IPv6 with 128 bits removes the shortage entirely, but the reader's machine reported IPv6 as none, so every packet in this story travelled the crowded IPv4 path; Chapter 34 takes IPv6 further.

DNS – From a Name to an Address

25.0 What this chapter gives you

1. You will be able to explain why the internet needs a naming system at all, and what people did in the years before it existed.
2. You will be able to draw the shape of the whole name space, and read `github.com` as a path starting from a single dot.
3. You will be able to name the four kinds of thing that take part in a lookup, and say exactly which one does the running around.
4. You will be able to walk through a real lookup of `github.com` from the reader's own machine, step by step, with realistic times for each step.
5. You will be able to explain the thirteen root servers honestly, including why there are only thirteen names and why there are over 1,900 machines.
6. You will be able to read every common record type and know when each is used, including the rule that catches people out at the top of a domain.
7. You will be able to explain caching and TTL well enough to plan a migration, and to say why “waiting for DNS to propagate” is the wrong picture.
8. You will be able to explain why the same name gives different answers in different countries, and separate the DNS reasons from the routing reasons.
9. You will be able to say exactly why the reader's `github.com` resolved to `20.207.73.82`, and prove the ownership of that address yourself.
10. You will be able to diagnose a DNS fault with `dig`, and tell in one command whether a problem is DNS at all or something else entirely.

25.1 The problem DNS solves

■ 25.1.1 PLAIN – in simple words

1. Computers on a network find each other by number, not by name.
2. The number for the reader's copy of `github.com` was `20.207.73.82`.
3. People are bad at remembering numbers and good at remembering names.
4. So we need something that turns a name into a number, on demand, in a few thousandths of a second, billions of times a day, all over the world.
5. That something is the **Domain Name System**, almost always called DNS.
6. DNS is a lookup service. You give it a name. It gives you back a number.
7. It is not a directory you browse. There is no list of every name. You can only ask about a name you already know.

8. DNS does one small job, and does it for the entire internet. Almost nothing else on the internet works if DNS stops working.
9. That is why a DNS fault feels like “the internet is down” even when every cable and every router is perfectly fine.

■ 25.1.2 PLAIN - a picture in your head

1. Think of a very large office building with no signs on any door.
2. Every room has a number. Room 20207 on floor 73. Nothing else.
3. You want to visit a person, not a number. You know the person’s name.
4. At the entrance there is a desk. You say a name. The desk says a room number. You walk to that room.
5. The desk does not walk you there. It only tells you where to go. Walking is still your job.
6. Now imagine the building is the size of a planet, with billions of rooms, and rooms move house every few minutes.
7. One desk could never hold all of that. So the building has one small desk per department, and the front desk knows which department desk to ask.
8. That chain of desks, each knowing only its own part, is the shape of DNS.

Where this comparison breaks: a receptionist knows the answer or does not. A DNS server usually knows only who to ask next, and that is the whole trick. The front desk in DNS almost never holds the final answer. Also, in a real building you ask one desk once. In DNS the answer is remembered by everyone along the way for a set number of seconds, which changes the behaviour completely.

■ 25.1.3 PLAIN - a worked example

1. Before DNS existed, there was one file. It was called `HOSTS.TXT`.
2. It was a plain text file. Each line held one address and one name.
3. Here is roughly what a line looked like in the ARPANET era.

```
HOST : 10.0.0.51 : SRI-NIC,NIC : DEC-2060 : TOPS20 : TCP/TELNET
HOST : 10.1.0.11 : UCLA-CCN : IBM-3033 : OS/MVS : TCP
HOST : 26.0.0.73 : SRI-KL,SRI : F0ONLY-F2 : TENEX : TCP/FTP
```

4. One organization maintained that file for everyone: the Network Information Center at Stanford Research Institute, in California.
5. If your machine changed address, you emailed or phoned the Network Information Center. A human edited the file.
6. Every other machine on the network downloaded the whole file, by FTP, every so often, and used its own copy.
7. Work out the cost. If there are 300 machines and each downloads the file twice a week, that is 600 transfers a week from one server.
8. Now double the number of machines. The file gets twice as big **and** twice as many machines download it. The load grows with the square.
9. Worse: your copy is stale from the moment you download it. A machine that moved on Tuesday is unreachable for you until you download again.
10. And there could be only one machine called SRI-KL in the world, because the file had one flat list of names with no structure.

■ 25.1.4 PLAIN - what is really happening inside

1. The `HOSTS.TXT` design failed for four separate reasons, and it is worth keeping them apart because DNS fixes each one differently.
2. **Load.** One server served every machine. DNS fixes this by spreading the data across millions of servers, each holding a small piece.
3. **Freshness.** A downloaded copy is a snapshot. DNS fixes this by asking at the moment you need the answer, and by attaching an expiry time.
4. **Naming clashes.** One flat list means one name each. DNS fixes this by giving names a tree shape, so `www` can exist under a million parents.
5. **Human bottleneck.** One team edited every change. DNS fixes this by handing each organization control of its own branch. This is called **delegation** and it is the single most important idea in the system.
6. Delegation means the top of the tree does not know your addresses. It only knows which servers to ask about your part of the tree.
7. That keeps the top of the tree small, stable, and almost never changing, while the leaves change every minute.

■ 25.1.5 TECHNICAL - the engineer's version

1. `HOSTS.TXT` was maintained by the Network Information Center at Stanford Research Institute, known as SRI-NIC. Elizabeth Feinler ran that group and built and maintained the first ARPANET host directory.
2. The file format and the FTP distribution procedure were written up in RFC 952 and RFC 953, both published in October 1985, which formalized a practice already years old.
3. Paul Mockapetris, then at the Information Sciences Institute of the University of Southern California, designed DNS in 1983 at the request of Jon Postel.
4. The original specification was published in November 1983 as two documents: RFC 882, "Domain Names - Concepts and Facilities", and RFC 883, "Domain Names - Implementation and Specification".
5. Those were obsoleted in November 1987 by RFC 1034 and RFC 1035, with the same two titles. Those two documents are still the base specification today.
6. Nearly forty years of later RFCs amend them. Nothing has replaced them.
7. The first widely used implementation was BIND, the Berkeley Internet Name Domain, written at the University of California, Berkeley in 1984 and later maintained by the Internet Systems Consortium.

Item	Date	Document
HOSTS.TXT format	Oct 1985	RFC 952
DNS first spec	Nov 1983	RFC 882, RFC 883
DNS current base	Nov 1987	RFC 1034, RFC 1035
Negative caching	Mar 1998	RFC 2308
EDNS0 (current)	Apr 2013	RFC 6891

8. The honest version: RFC 1034 and RFC 1035 are not a clean read. Real behaviour today is spread over roughly one hundred RFCs, and the community maintains an informal list called "DNS Terminology", RFC 9499, published in March 2024, purely to keep the vocabulary straight.

■ 25.1.6 WORDS - remember these

1. DNS — the thing that turns names into numbers — Domain Name System, the distributed hierarchical database defined by RFC 1034 and RFC 1035.
2. Resolution — looking a name up — the process of mapping a domain name to resource records of a requested type and class.
3. Delegation — handing control of a branch to someone else — NS records in a parent zone pointing at the authoritative servers of a child zone.
4. HOSTS.TXT — the single hand-edited file everyone copied before DNS — the flat ARPANET host table distributed by FTP from SRI-NIC.
5. Zone — one chunk of the tree that one operator controls — an administrative unit of the name space, bounded above and below by delegation points.

25.2 The shape of the name space

■ 25.2.1 PLAIN - in simple words

1. Domain names look flat when you type them. They are not.
2. `github.com` is a path, and you read it right to left.
3. At the far right, invisible, is the **root**. It is written as a single dot.
4. Left of the root comes `com`. That is a **top level domain**.
5. Left of that comes `github`. That is a **second level domain**.
6. If there were more parts, like `api.github.com`, each new part to the left is one step further down the tree.
7. Each part between the dots is called a **label**.
8. The full name, including the invisible dot at the end, is the **fully qualified domain name**, usually shortened to FQDN.
9. Written properly it is `github.com.` with a trailing dot. Nobody types that dot. Browsers and programs add it silently.
10. The tree grows downwards from the root, which is why people call it an inverted tree. The root is at the top of the picture and the end of the name.

■ 25.2.2 PLAIN - a picture in your head

1. Think of a postal address, written the way most countries write it.
2. 14 Baker Lane, Kothrud, Pune, Maharashtra, India.
3. You read it inwards from the left, but the sorting machine reads it outwards from the right. Country first, then state, then city, then street.
4. Nobody sorting post in Delhi needs to know every street in Pune. They need to know only “send this bag to Pune”.
5. The Pune office needs to know only “send this to Kothrud”.
6. Domain names are the same address, written in the machine’s order. The most general part is on the right, the most specific on the left.
7. `com` is the country. `github` is the city. `api` is the street.

Where this comparison breaks: a postal hierarchy is geographic and mostly fixed. The DNS tree is administrative, not geographic, and a branch can be handed to anyone anywhere. A `.in` name can be served from Texas, and a `.com` name can be served entirely from Mumbai. Also, post offices always exist at every level, whereas in DNS a level can be a pure label with no server at all.

■ 25.2.3 PLAIN – a worked example

1. Take the reader's own name, `github.com`, and split it into a path.

```
Written:           github.com
Fully qualified:   github.com.
Read right to left:

.                 <- the root, one empty label
com.              <- top level domain
github.com.       <- second level domain, delegated to GitHub

If we add a subdomain:

api.github.com.   <- third level, GitHub's own choice
raw.github.com.
codeload.github.com.
```

2. Note what is *not* in that picture. `www` is not special. It is just a label somebody chose, the same as `api` or `raw`.
3. `www.github.com` is not “the website version” of `github.com`. It is a different name that happens to point at the same place.
4. We can check that with one command, and here is the real answer.

```
$ dig +noall +answer www.github.com A
www.github.com.  3257 IN CNAME github.com.
github.com.      60   IN A      140.82.114.4
```

5. `www.github.com` is an alias pointing at `github.com`, which then has the address. Two names, one destination.

■ 25.2.4 PLAIN – what is really happening inside

1. Each label is stored separately, not as one long string.
2. In a DNS message on the wire, `github.com` is encoded as a length byte, then the letters, then a length byte, then the letters, then a zero.

```
06 67 69 74 68 75 62 03 63 6f 6d 00
 6  g  i  t  h  u  b  3  c  o  m  end
```

3. The final `00` is the root label. It has length zero. That is the trailing dot, and it is genuinely present in every message on the wire.
4. So the trailing dot is not a typing convention. It is a real byte.
5. A **domain** is a name and everything under it. A **zone** is the part of a domain that one server actually holds.
6. Those two are different, and mixing them up causes real confusion. `com` is a domain containing millions of names. The `com` zone contains almost none of their addresses, only pointers to who does.
7. The point where a parent stops and a child starts is called a **zone cut**. At every zone cut there is a delegation.

■ 25.2.5 TECHNICAL – the engineer's version

1. RFC 1035 section 2.3.4 sets the hard limits, and they have never changed.

Limit	Value	Note
One label	63 octets	Length byte uses 6 bits
Full name on wire	255 octets	Includes length bytes
Labels per name	127 max	A consequence of the above
TTL field	32 bits	Signed use, max 2147483647

- Comparison of labels is case insensitive but case is preserved on the wire. RFC 4343, published January 2006, spells this out precisely.
- Many resolvers exploit that by randomizing the case of the query name and checking that the case comes back unchanged. This is called 0x20 encoding. It is a convention and a defence, not a standard.
- Top level domains come in categories: generic (gTLD) such as `com`, country code (ccTLD) such as `in`, sponsored such as `edu`, and one infrastructure TLD, `arpa`, used for reverse lookups.
- The authoritative list is the IANA root zone database. As of version 2026081200, last updated 12 August 2026, it lists roughly 1,500 top level domains, including internationalized names that begin with `xn--`.
- Certain names are reserved by standard and must never be delegated. RFC 2606, June 1999, reserves `example.com`, `example.net`, `example.org` and the TLDs `test`, `invalid`, `example` and `localhost`.
- RFC 6761, February 2013, defines special use domain names that resolvers are expected to handle locally rather than by querying the tree.
- The honest version: `.local` is not part of the global DNS at all. It is handled by Multicast DNS, RFC 6762, February 2013, which is a different protocol on UDP port 5353 that happens to reuse the DNS message format.

■ 25.2.6 WORDS - remember these

- Label — one piece between the dots — an octet string of 1 to 63 octets, compared case insensitively.
- FQDN — the complete name including the final dot — a domain name terminated at the root, unambiguous with no search list applied.
- TLD — the last part, like `com` — a zone delegated directly from the root zone by IANA.
- Zone — the part of the tree one operator serves — the set of records under a name, stopping at each delegation below it.
- Zone cut — the boundary between parent and child — the point where NS records in the parent delegate authority downward.

25.3 The four kinds of participant

■ 25.3.1 PLAIN - in simple words

- Four different things take part in a lookup. They are easy to confuse, and almost every DNS misunderstanding comes from confusing two of them.
- The **stub resolver** lives inside your own computer. It is small. It knows almost nothing. It asks one question and waits for one answer.
- The **recursive resolver** is a server somewhere else. It is the one that does the work. For the reader that server is at `1.1.1.1`.
- The **root servers** sit at the very top. They know one thing: which servers handle each top level domain.

5. The **authoritative servers** hold real answers for one zone each. The servers for `com` are authoritative for `com`. GitHub's servers are authoritative for `github.com`.
6. Say this out loud, because it is the sentence people get wrong: your machine does not walk the tree. The recursive resolver walks the tree.
7. Your machine asks one question, once, and waits. Everything else happens somewhere else, at somebody else's expense.

■ 25.3.2 PLAIN - a picture in your head

1. Imagine you want a rare book and you are not going to leave your chair.
2. You phone your local librarian and say: get me this book.
3. That librarian does not have the book. But the librarian has a phone, a membership of the national library network, and patience.
4. The librarian phones the national catalogue. The catalogue says: that subject belongs to the medical library.
5. The librarian phones the medical library. That library says: that title is in the Pune branch.
6. The librarian phones the Pune branch and gets the answer.
7. Then the librarian phones you back with one sentence, and writes the answer on a card so the next person who asks gets it instantly.
8. You made one phone call. The librarian made three. That is exactly the split between a stub resolver and a recursive resolver.

Where this comparison breaks: a librarian could read the book to you. A DNS resolver never fetches the page, only the address. It also never invents an answer, and never asks a server that is not named in a referral it just received. Also the librarian is trusted by you personally, whereas your resolver is often a company you have never had any dealings with.

■ 25.3.3 PLAIN - a worked example

1. Here is who holds what, for the reader's actual lookup.

Who	Knows	Example
Stub on the Mac	Nothing, plus cache	asks 1.1.1.1
Recursive 1.1.1.1	Who to ask, plus cache	walks the tree
Root server	Where <code>com</code> lives	13 names, A to M
<code>com</code> servers	Where <code>github.com</code> lives	a to m gtld-servers
GitHub's servers	The actual address	<code>dns1.p08.nsone.net</code>

2. Notice that only the bottom row knows the answer.
3. Notice also that the top three rows never change from one lookup to the next. Only the bottom row is specific to `github.com`.
4. That is why a resolver that has been running for an hour answers most queries without leaving the building. It already knows the top three rows.

■ 25.3.4 PLAIN - what is really happening inside

1. The stub resolver on macOS is not a separate program you can see easily. It is library code plus a system service called `mDNSResponder`.
2. When an application calls `getaddrinfo`, that call goes into the stub.
3. The stub checks a small local cache. If the answer is there and not expired, it returns immediately, with no network traffic at all.

4. If not, the stub builds one DNS query message and sends it to the configured resolver, 1.1.1.1, on UDP port 53.
5. It sets a flag in that message called **Recursion Desired**. That flag means “please do all the work and come back with the final answer”.
6. The recursive resolver honours that flag. Root, TLD and authoritative servers do not. They answer only from their own zone, or hand back a referral naming the next servers down.
7. So the difference between a recursive resolver and an authoritative server is not the software. It is the role and the configuration.
8. Many programs can do both. Running both roles on the same address is a known bad idea, because it mixes cached data with authoritative data.

■ 25.3.5 TECHNICAL - the engineer's version

1. RFC 8499, obsoleted by RFC 9499 in March 2024, gives the precise terms: stub resolver, recursive resolver, iterative resolver, full resolver, authoritative server and forwarder.
2. A **forwarder** is a fifth role that people forget. A home router usually runs a small forwarding resolver: it accepts your query, does no tree walking, and passes the query to an upstream recursive resolver.
3. That is what the reader's router at 192.168.0.1 would normally do. The reader has bypassed it by configuring 1.1.1.1 directly.
4. The **RD** bit (Recursion Desired) is bit 8 of the DNS header flags. The **RA** bit (Recursion Available) is bit 15, set by a server that is willing to recurse for you.
5. An authoritative server sets **AA** (Authoritative Answer). A cached answer from a recursive resolver does not have AA set. This is how you tell them apart in a packet capture.
6. Query the flags directly with dig. In the reader's own output for github.com we see flags: qr rd ra, meaning it is a response, recursion was desired, recursion was available, and crucially there is no aa.
7. QNAME minimization, RFC 9156, November 2021, changed what a recursive resolver sends upward. Instead of sending the full name to the root, a modern resolver sends only com. to the root. Cloudflare and Google both do this by default now. This is a privacy improvement, deployed since 2019.

■ 25.3.6 WORDS - remember these

1. Stub resolver — the tiny piece in your own machine — the client-side code that sends one recursive query and does not follow referrals.
2. Recursive resolver — the server that does the running around — a full service resolver that follows referrals and caches results.
3. Authoritative server — a server that genuinely holds a zone — one that answers from zone data and sets the AA bit.
4. Referral — “not me, ask them” — a response with no answer records and NS records for a child zone in the authority section.
5. Forwarder — a middleman that just passes queries along — a resolver configured to send all queries to another recursive resolver.
6. RD bit — “please do the work for me” — the Recursion Desired header flag, honoured only by recursive resolvers.

25.4 One full lookup, step by step

■ 25.4.1 PLAIN - in simple words

1. We will now follow one real lookup all the way. The name is `github.com`. The machine is the reader's Mac, in India, on home broadband.
2. The resolver is `1.1.1.1`. The answer that came back was `20.207.73.82`.
3. Step 1. A program asks the operating system for the address of `github.com`.
4. Step 2. The stub resolver looks in its own small cache. Nothing there.
5. Step 3. The stub sends one question to `1.1.1.1` and waits.
6. Step 4. `1.1.1.1` looks in its own much bigger cache. Suppose it is cold and knows nothing, so it starts at the top.
7. Step 5. It asks a root server: who handles `com`?
8. Step 6. The root server does not give an address. It gives a list of the `com` servers. That is a **referral**.
9. Step 7. It asks a `com` server about `github.com`.
10. Step 8. The `com` server also gives a referral: GitHub's own name servers.
11. Step 9. It asks one of GitHub's name servers about `github.com`.
12. Step 10. That server holds the zone, so it gives a real answer.
13. Step 11. `1.1.1.1` stores the answer in its cache with a countdown.
14. Step 12. `1.1.1.1` sends the answer back to the Mac.
15. Step 13. The stub stores it too, and hands the address to the program.

■ 25.4.2 PLAIN - a picture in your head

1. Think of finding a person in a very large company on your first day.
2. You ask reception. Reception says: that is the engineering building, ask their front desk.
3. You go to engineering reception. They say: that is the platform team, ask their team lead.
4. You go to the team lead. The team lead says: desk 42, second floor.
5. Three questions, three answers, and only the last one was useful in itself.
6. The first two were not failures. They were progress. Each one narrowed the search by a huge factor.
7. Reception did not know about desk 42, and never will, and does not need to. That is the whole design.

Where this comparison breaks: you walked between buildings. In DNS the resolver sends fresh packets to fresh servers each time, and each hop is a completely independent question with its own timeout and its own retry. Also, reception in a real company will guess if unsure. A DNS server never guesses. If it does not have a delegation, it says so, and the resolver stops.

■ 25.4.3 PLAIN - a worked example

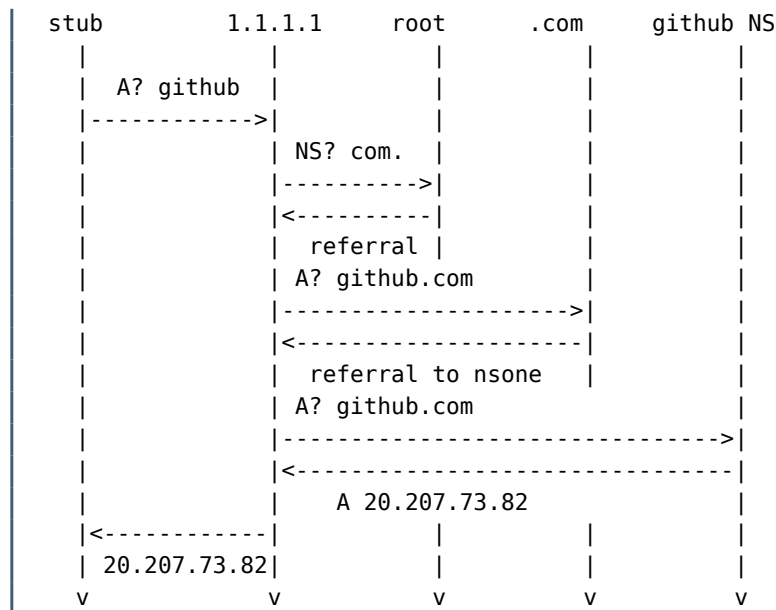
1. Here is the same lookup as a table, with the question asked at each step, what came back, and a realistic time on Indian home broadband.

Step	Question to	Answer type	Time
1	stub cache	miss	0.1 ms
2	1.1.1.1	(working)	8 ms rtt
3	root server	referral to com	18 ms
4	com server	referral to nsone	25 ms
5	nsone server	20.207.73.82	30 ms
6	back to Mac	final answer	8 ms

2. Cold total, nothing cached anywhere: roughly 90 milliseconds.
3. Warm total, when 1.1.1.1 already has the answer: roughly 8 milliseconds, because only step 2 and step 6 happen.
4. Warm total when the Mac itself has it cached: under 1 millisecond, with no packets sent at all.
5. In real life the cold case is rare. The root and com referrals live in a busy resolver's cache for hours, so usually only the last hop is needed.

■ 25.4.4 PLAIN - what is really happening inside

1. Here is the same journey drawn out. Read downwards. Each arrow is one UDP packet in one direction.



2. Count the packets. Eight in total for a completely cold lookup.
3. Only two of them touch the reader's own broadband line. The other six are sent by Cloudflare, from Cloudflare's own network, at Cloudflare's cost.
4. That is the point of the design. The expensive work happens once, near the resolver, and is then reused for every other customer of that resolver.
5. Each referral arrives with **glue**: the addresses of the servers named in the referral, in the additional section. Without glue the resolver would have to look up the name servers' own names, which could loop forever.
6. If a server does not reply within about 800 milliseconds, the resolver tries another server from the same list. There are always several.

■ 25.4.5 TECHNICAL - the engineer's version

1. You can watch the whole chain with `dig +trace`, which asks each level in turn from your own machine instead of asking a recursive resolver.
2. The output starts with the root zone NS set, which `dig` gets from its built-in hints file, then follows each referral.

```

$ dig +trace github.com

.          87203  IN  NS  a.root-servers.net.
.          87203  IN  NS  b.root-servers.net.
.          87203  IN  NS  c.root-servers.net.
...

```

```
;; Received 525 bytes from 198.41.0.4#53(a.root-servers.net)

com.      172800 IN  NS   a.gtld-servers.net.
com.      172800 IN  NS   b.gtld-servers.net.
...
;; Received 1174 bytes from 192.5.6.30#53(a.gtld-servers.net)

github.com. 172800 IN NS dns1.p08.nsone.net.
github.com. 172800 IN NS ns-421.awsdns-52.com.
...
;; Received 322 bytes from 192.31.80.30#53(d.gtld-servers.net)

github.com. 60  IN  A   20.207.73.82
;; Received 55 bytes from 198.51.44.8#53(dns1.p08.nsone.net)
```

3. The exact addresses in the final line depend on where you are. In India the reader got 20.207.73.82. From a machine in the United States the same command returns addresses in 140.82.112.0/20, which is GitHub's own space. Section 25.10 explains that difference in full.
4. The NS set for github.com is genuinely split across two providers. This is real output, taken with dig, and it is worth studying.

```
$ dig +noall +answer github.com NS
github.com. 1348 IN  NS   dns1.p08.nsone.net.
github.com. 1348 IN  NS   dns2.p08.nsone.net.
github.com. 1348 IN  NS   dns3.p08.nsone.net.
github.com. 1348 IN  NS   dns4.p08.nsone.net.
github.com. 1348 IN  NS   ns-421.awsdns-52.com.
github.com. 1348 IN  NS   ns-520.awsdns-01.net.
github.com. 1348 IN  NS   ns-1283.awsdns-32.org.
github.com. 1348 IN  NS   ns-1707.awsdns-21.co.uk.
```

5. Four servers at NS1, a managed DNS provider, and four at Amazon Route 53. Two independent providers, so an outage at either one does not take github.com off the internet.
6. Notice the Amazon servers are spread across com, net, org and co.uk. That is deliberate: if one of those TLDs has trouble, the others still resolve. This is a convention among large operators, not a standard.
7. Timing figures worth knowing, measured with dig and reported in the Query time line at the bottom of every response:

Case	Typical query time
Local stub cache hit	under 1 ms
Warm 1.1.1.1 in India	4 to 15 ms
Cold, full tree walk	60 to 150 ms
Server not responding	800 ms then retry
Total giving up	5 s default in dig

■ 25.4.6 WORDS - remember these

1. Referral — “ask these servers instead” — a delegation response containing NS records for a child zone and no answer records.
2. Glue record — the address of a name server, given along with its name — an A or AAAA record in the additional section, needed to break a dependency loop.

3. Iterative query — asking one level at a time — a query with the RD bit clear, answered from local data or with a referral.
4. Priming — how a resolver finds the root — a startup query for the root NS set, seeded from a built-in hints file.
5. Query time — how long the answer took — the round trip measured by dig, printed in milliseconds at the foot of the response.

25.5 The root servers

■ 25.5.1 PLAIN - in simple words

1. At the top of the tree sit the root servers. Every lookup that is not already cached starts there.
2. There are thirteen of them, named with the letters A to M, at addresses ending in `root-servers.net`.
3. That is the sentence everyone repeats. It is true about the names, and very misleading about the machines.
4. There are not thirteen machines. There are over 1,900 physical servers spread across the world, all answering on those same thirteen addresses.
5. They are run by twelve different organizations, in different countries, under different legal systems, on purpose.
6. No single organization can switch off the root. That was designed in.
7. What the root knows is small. It knows which servers handle each top level domain, and nothing else. It has never heard of `github.com`.
8. The root zone file is only a few megabytes. You could email it.

■ 25.5.2 PLAIN - a picture in your head

1. Think of a telephone number that reaches whichever branch of a chain is nearest to you.
2. You dial one number. Somebody answers. Which building they are sitting in depends entirely on where you are standing.
3. In Pune it rings in Pune. In Frankfurt it rings in Frankfurt. Same number, different building, and you cannot tell which.
4. That is what the thirteen root addresses are. Each one is a single number answered by hundreds of buildings.
5. The trick that makes this work is called **anycast**, and we come back to it properly in section 25.9.

Where this comparison breaks: a phone chain routes calls by a central system that decides. Anycast has no central decision at all. Each network on the path picks the shortest route it knows to that address, so the choice emerges from the routing system rather than being made by anyone. There is also no failover list: if the nearest instance disappears, routing simply converges on the next nearest, usually within seconds, with no cooperation from anybody.

■ 25.5.3 PLAIN - a worked example

1. Here are the real thirteen, with their addresses and who runs them. This list has been almost unchanged for decades.

Name	IPv4 address	Operator
a.root-servers.net	198.41.0.4	Verisign
b.root-servers.net	170.247.170.2	USC ISI

Name	IPv4 address	Operator
c.root-servers.net	192.33.4.12	Cogent
d.root-servers.net	199.7.91.13	Univ of Maryland
e.root-servers.net	192.203.230.10	NASA Ames
f.root-servers.net	192.5.5.241	ISC
g.root-servers.net	192.112.36.4	US DoD NIC
h.root-servers.net	198.97.190.53	US Army Research Lab
i.root-servers.net	192.36.148.17	Netnod
j.root-servers.net	192.58.128.30	Verisign
k.root-servers.net	193.0.14.129	RIPE NCC
l.root-servers.net	199.7.83.42	ICANN
m.root-servers.net	202.12.27.33	WIDE Project

2. Verisign runs two, A and J. That is why thirteen names are run by twelve organizations.
3. One address on that list changed recently, which is unusual enough to be worth remembering: B moved from 199.9.14.201 to 170.247.170.2 on 27 November 2023.
4. Six of the thirteen are run by United States government or university bodies. Netnod is Swedish, RIPE NCC is Dutch, WIDE is Japanese. That mix is frequently argued about in internet governance, and it is fair to say the balance is historical rather than designed.

■ 25.5.4 PLAIN - what is really happening inside

1. Each of the thirteen addresses is announced into the global routing system from many separate locations at once.
2. When your resolver sends a packet to 198.41.0.4, the routing system delivers it to whichever announcement is closest in routing terms.
3. "Closest" here does not mean closest in kilometres. It means fewest network hops by the rules the routing protocol uses.
4. The resolver has no idea which instance answered. It sees one address.
5. You can find out, though. There is a special query, `hostname.bind` in the CH class, that many instances answer with their own site identifier.

```
$ dig @198.41.0.4 hostname.bind CH TXT +short
"nnn1.a.root-servers.org"
```

6. Ask that from Mumbai and from London and you will get different strings back from the same address. That is anycast made visible.
7. The root servers are astonishingly boring by design. They serve one small zone, read only, and change roughly twice a day when the root zone is republished.

■ 25.5.5 TECHNICAL - the engineer's version

1. As of 13 August 2026 the root server system reported **2,003 operational instances** across the thirteen letters. The figure is published live by the root server operators and grows most months.
2. The distribution is very uneven. Verified counts from the same source on that date included F at 366 instances, E at 328, D at 231, I at 90, and at the other end B and G at 6 each.
3. Why thirteen, and not fourteen or a hundred? The limit came from the original 512-octet maximum for a DNS message over UDP, set in RFC 1035.
4. A priming response must carry the NS records for the root plus the A records giving their addresses. With DNS name compression, thirteen name servers plus thirteen glue addresses fit inside 512 octets. Fourteen did not.

5. That is why every root server is a single letter followed by `.root-servers.net`: the shared suffix compresses to two bytes per record.
6. The constraint is long gone. EDNS0, RFC 6891, lets a response be much larger, and the current priming response with IPv6 glue is well over 800 octets. The number thirteen survives purely because changing it has no benefit.
7. The root zone has been signed with DNSSEC since 15 July 2010. The key signing key was rolled for the first time on 11 October 2018, an operation planned for years and completed without visible breakage.
8. The root zone itself is generated by IANA, which is a function operated by Public Technical Identifiers under ICANN, and published by Verisign as the root zone maintainer.

Root fact	Value	As of
Letters	13, A to M	since 1980s
Operators	12 organizations	current
Instances	2,003	13 Aug 2026
Root zone signed	yes	15 Jul 2010
TLDs delegated	about 1,500	Aug 2026

9. The honest version: people say “the root servers are the most attacked machines on the internet”. Attacks do happen, and there were notable events in October 2002 and February 2007, but the system has never gone down as a whole, and anycast means an attack is absorbed locally by the instances near the attacker rather than concentrated on one box.

■ 25.5.6 WORDS – remember these

1. Root server — the server at the top of the name tree — an authoritative server for the root zone, reachable at one of thirteen well known addresses.
2. Anycast — one address answered from many places — the same prefix announced from multiple locations, with routing selecting the nearest.
3. Hints file — the built-in list of root addresses — the bootstrap file used for priming before any DNS query can be made.
4. Priming query — a resolver’s first question at startup — a query for the root NS set, used to replace the hints with live data.
5. Root zone maintainer — the party that publishes the root file — currently Verisign, working from the zone content authorized by IANA.

25.6 Record types

■ 25.6.1 PLAIN – in simple words

1. DNS does not only store addresses. It stores several kinds of fact about a name, and each kind has a short code called a **record type**.
2. When you ask a question you name both the name and the type. “What is the A record for `github.com`” is a different question from “what is the MX record for `github.com`”, and both are normal.
3. An **A** record holds an IPv4 address. That is the common one.
4. An **AAAA** record holds an IPv6 address. Said out loud as “quad A”.
5. A **CNAME** record says “this name is another name for that name”.
6. An **MX** record says where to deliver email for this domain.

7. A **TXT** record holds free text, used mostly to prove ownership.
8. An **NS** record names a server that is authoritative for a zone.
9. An **SOA** record holds the administrative settings for a zone.
10. A **PTR** record maps an address back to a name.
11. An **SRV** record says which host and port run a named service.
12. A **CAA** record says which certificate authorities may issue certificates for this domain.

■ 25.6.2 PLAIN - a picture in your head

1. Think of a person's entry in an old paper address book.
2. There is a street address. That is the A record.
3. There is a second address for their new flat abroad. That is AAAA.
4. There is a line saying "see under her maiden name". That is CNAME.
5. There is a separate line saying "send post care of the office". That is MX, because letters go somewhere different from where the person lives.
6. There is a scribbled note in the margin proving the page is genuine. TXT.
7. There is a line saying which secretary handles her diary. NS.
8. There is a stamp on the page saying who maintains this book and how often it is reprinted. SOA.
9. There is a reverse index at the back listing addresses and who lives there. PTR.

Where this comparison breaks: an address book is one document owned by one person. DNS records for one name can be split across different owners, and the reverse index at the back is maintained by an entirely different organization that may never have heard of the person. That mismatch is real and it is why reverse lookups so often disagree with forward ones.

■ 25.6.3 PLAIN - a worked example

1. Here are real records for `github.com`, queried with `dig`. Every line below is genuine output, only trimmed for width.

```
$ dig +noall +answer github.com SOA
github.com. 3600 IN SOA dns1.p08.nsone.net.
      hostmaster.nsone.net. 1656468023 43200 7200 1209600 3600

$ dig +noall +answer github.com MX
github.com. 130 IN MX 0
      github-com.mail.protection.outlook.com.

$ dig +noall +answer github.com CAA
github.com. 3600 IN CAA 0 issue "letsencrypt.org"
github.com. 3600 IN CAA 0 issue "digicert.com"
github.com. 3600 IN CAA 0 issue "sectigo.com"
github.com. 3600 IN CAA 0 issuewild "digicert.com"

$ dig +noall +answer www.github.com A
www.github.com. 3257 IN CNAME github.com.
github.com.      60   IN A      140.82.114.4
```

2. Read the MX line. Mail for `github.com` goes to Microsoft's Outlook mail protection service, not to GitHub's own machines. The domain is GitHub's; the mail path is Microsoft's. That is a small clue about who owns what.
3. Read the CAA lines. Only four named certificate authorities may issue for `github.com`. Any other authority that follows the rules must refuse.

4. Read the SOA line. `dns1.p08.nsone.net` is the primary server, `hostmaster.nsone.net` is the contact address with the first dot standing in for an at sign, and the numbers are serial, refresh, retry, expire and minimum, in that order.

■ 25.6.4 PLAIN - what is really happening inside

1. Records with the same name, type and class are handled together as a set, called an **RRset**. You cannot have half a set. All of it is returned or none of it.
2. A CNAME is special and slightly dangerous. It says “stop, restart the whole lookup using this other name”.
3. Because a CNAME redirects everything, the standard forbids any other record existing alongside a CNAME at the same name.
4. That rule creates the problem everyone hits eventually. At the very top of a domain, called the **apex** or the **naked domain**, an SOA record and NS records must exist. They are compulsory.
5. So a CNAME can never be placed at `github.com` itself. Only at names below it, like `www.github.com`.
6. If you have ever tried to point `example.com` at a hosting provider’s hostname and been told it is not allowed, that is exactly this rule.
7. The workarounds are covered in the technical block below, and it matters that some of them are not standards.

■ 25.6.5 TECHNICAL - the engineer’s version

1. The main types, with their numeric codes and defining documents.

Type	Code	Purpose
A	1	IPv4 address, RFC 1035
NS	2	Delegation, RFC 1035
CNAME	5	Alias, RFC 1035
SOA	6	Zone parameters, RFC 1035
PTR	12	Reverse mapping, RFC 1035
MX	15	Mail exchanger, RFC 1035
TXT	16	Free text, RFC 1035
AAAA	28	IPv6 address, RFC 3596
SRV	33	Service location, RFC 2782
CAA	257	CA authorization, RFC 8659

2. The CNAME restriction is RFC 1034 section 3.6.2: if a CNAME is present at a node, no other data should be present. DNSSEC records are the only exception the later standards carved out.
3. Workarounds for the apex problem, and their honest status:
 1. ALIAS or ANAME records: an **implementation detail** of particular providers such as NS1, DNSimple and Route 53. Not a standard. The provider resolves the target and serves an A record itself.
 2. CNAME flattening: Cloudflare’s name for the same trick, introduced in 2014. Also not a standard.
 3. The HTTPS and SVCB record types, RFC 9460, published November 2023. These are a real standard and are supported by Chrome, Safari and Firefox, but they solve service binding rather than being a general apex CNAME.

4. `github.com` has no AAAA record. This was checked in August 2026 and the answer was NODATA, meaning the name exists but has no record of that type. The response carries an SOA record in the authority section instead.

```
$ dig +noall +answer +authority github.com AAAA
github.com. 1081 IN SOA dns1.p08.nsone.net.
hostmaster.nsone.net. 1656468023 43200 7200 1209600 3600
```

5. That single fact explains a line in the reader's own diagnostics: their system reported IPv6: (none). Even with working IPv6 at home, a connection to `github.com` over IPv6 is impossible, because no IPv6 address is published for it.
6. NODATA and NXDOMAIN are different and get confused. NXDOMAIN means the name does not exist at all. NODATA means it exists but not with that type. Both return an SOA in the authority section, which is what sets the negative cache time.
7. TXT records are the internet's junk drawer. `github.com` publishes more than twenty of them, most of them proofs of ownership for third party services.

```
"v=spf1 ip4:192.30.252.0/22 include:spf.protection.outlook.com ..."
"MS=ms44452932"
"google-site-verification=UTM-3akMgubp6tQtgEuAkYNYL..."
"apple-domain-verification=RyQhdzTl6Z6x8ZP4"
"docusign=087098e3-3d46-47b7-9b4e-8a23028154cd"
```

8. Reverse DNS lives under two special zones: `in-addr.arpa` for IPv4 and `ip6.arpa` for IPv6. The address is written backwards, label by label. `20.207.73.82` becomes `82.73.207.20.in-addr.arpa`.
9. Reverse DNS is delegated by whoever holds the address block, not by whoever holds the domain name. Those are different parties, so the two directions are free to disagree, and usually do.
10. Here are three real reverse lookups that make the point better than any explanation.

```
$ dig +short -x 140.82.112.3
lb-140-82-112-3-iad.github.com.

$ dig +short -x 8.8.8.8
dns.google.

$ dig +short -x 20.207.73.82
(no answer at all)
```

11. The first has a PTR that names GitHub and even encodes the site code `iad`, which is the airport code for Washington Dulles. The second is tidy. The third, the reader's own answer, has no PTR record whatsoever.
12. That is normal and is not a fault. Large cloud edges frequently publish no reverse records for service addresses. Mail servers are the one place where a missing or mismatched PTR causes real trouble, because receiving mail systems check it.

■ 25.6.6 WORDS – remember these

1. RRset — all records of one type at one name — a resource record set, the atomic unit of DNS data and of DNSSEC signing.
2. Apex — the very top of a domain — the zone origin, where SOA and NS records are mandatory and a CNAME is therefore forbidden.
3. CNAME — an alias to another name — canonical name record, which restarts resolution at the target and excludes all other data at that node.

4. NODATA — the name exists but not with that type — a NOERROR response with an empty answer section and an SOA in the authority section.
5. NXDOMAIN — the name does not exist — response code 3, cached for the time given by the SOA minimum field.
6. PTR — the record that maps an address to a name — a pointer record under `in-addr.arpa` or `ip6.arpa`, controlled by the address holder.

25.7 Caching and TTL

■ 25.7.1 PLAIN - in simple words

1. If every lookup walked the whole tree, the root servers would receive trillions of queries a day and the internet would stop.
2. They do not, because of caching. Almost every answer is reused.
3. Every DNS answer carries a number called the **TTL**, short for time to live. It is a count of seconds.
4. The TTL means: you may keep this answer and reuse it for this many seconds. After that, throw it away and ask again.
5. A resolver that just answered a question for one customer can answer the same question for a million more without doing any work.
6. This is why a busy resolver like `1.1.1.1` answers most questions in a few milliseconds. The answer was already sitting there.
7. Caching is not an optimization bolted on later. It is load bearing. DNS without caching does not work at internet scale at all.

■ 25.7.2 PLAIN - a picture in your head

1. Think of a carton of milk with a date printed on it.
2. The date is a promise from the dairy: this is good until then.
3. You are allowed to use it any time before the date without checking.
4. After the date you must go and get fresh milk. You do not get to argue.
5. Now the important part. If the dairy changes its recipe today, every carton already in every fridge still has the old recipe until its date passes.
6. Nobody comes to your house and swaps it. There is no recall.
7. That is exactly what happens when someone changes a DNS record. Old answers sit in caches everywhere until each one expires on its own schedule.

Where this comparison breaks: milk actually goes bad. A DNS answer past its TTL may still be perfectly correct. The TTL is a policy about how long you may assume, not a statement about when the fact changes. Also, a resolver is allowed to throw the answer away early, for example when it runs short of memory. So the TTL is a maximum, not a minimum.

■ 25.7.3 PLAIN - a worked example

1. You can watch a TTL count down with your own eyes. Ask the same question twice, a few seconds apart, through a caching resolver.

```
$ dig +noall +answer github.com A
github.com. 60 IN A 20.207.73.82

... wait 10 seconds ...

$ dig +noall +answer github.com A
github.com. 50 IN A 20.207.73.82
```



```

... wait 20 seconds ...

$ dig +noall +answer github.com A
github.com. 30 IN A 20.207.73.82

... wait 35 seconds ...

$ dig +noall +answer github.com A
github.com. 60 IN A 20.207.73.82

```

2. Read it carefully. The number falls by exactly one per second.
3. The resolver is not re-asking. It is subtracting the time it has held the record and reporting what is left.
4. When it reaches zero, the entry is dropped. The next query goes back to GitHub's name servers and comes back at the full 60 again.
5. github.com publishes a short TTL of 60 seconds. That is a deliberate choice. It lets GitHub move traffic quickly at the cost of more queries.
6. Compare that with a TTL of 86400 seconds, one day, which is common for records that never move, such as MX records for a stable mail provider.

■ 25.7.4 PLAIN - what is really happening inside

1. There is not one cache. There are usually four or five, stacked.

```

Application (browser has its own cache)
|
Operating system stub cache (mDNSResponder on macOS)
|
Home router forwarder cache (often, on 192.168.0.1)
|
Recursive resolver cache (1.1.1.1)
|
Authoritative server (the truth)

```

2. The reader has removed one of those layers by pointing the Mac straight at 1.1.1.1 instead of at the router at 192.168.0.1.
3. Each layer counts down independently. A record can be expired at one layer and still fresh at the layer above it.
4. Browsers make this worse. Chrome and Firefox keep their own DNS cache and have historically ignored short TTLs, holding entries for around 60 seconds regardless. That is an implementation detail and it changes between versions.
5. There is also **negative caching**: the fact that a name does not exist is itself cached, so a typo does not generate a fresh tree walk every time.
6. Negative caching uses a different number: the last field of the SOA record, called the minimum field, which was repurposed for exactly this.
7. For github.com that field is 3600, so a nonexistent name under github.com is remembered as nonexistent for up to an hour.

■ 25.7.5 TECHNICAL - the engineer's version

1. TTL is a 32-bit field in every resource record. RFC 2181, July 1997, section 8, states it must be treated as an unsigned value between 0 and 2147483647, and that values with the top bit set must be treated as zero.

2. Negative caching is specified in RFC 2308, March 1998. The negative cache time is the **lesser** of the SOA minimum field and the SOA record's own TTL. Most implementations additionally cap it, commonly at 3 hours.
3. Real TTL values, read from live output, showing how wide the spread is:

Record	TTL seen	Meaning
github.com A	60 s	moves often
github.com NS	1348 s	of 172800 max
github.com SOA	3600 s	1 hour
github.com CAA	3600 s	1 hour
example.com A	300 s	5 minutes
root NS set	87203 s	of 518400 max

4. Notice that the NS and root values shown are partly counted down already. The com delegation NS records are published with a TTL of 172800 seconds, which is 48 hours, and the root NS set with 518400 seconds, six days.
5. Standard migration procedure, and this is genuinely what operations teams do, in this order:
 1. At least 48 hours before the move, lower the TTL on the record from its normal value to 60 or 300 seconds. Wait for the old, long TTL to expire everywhere. This waiting step is the part people skip.
 2. Perform the change. Because the TTL is now short, caches drop the old answer within a minute or five.
 3. Watch traffic drain from the old address. Keep the old address serving until it is quiet.
 4. Once stable, raise the TTL back up to reduce query load.
6. The phrase “DNS propagation” is common and slightly wrong. Nothing is pushed anywhere. There is no distribution process to wait for.
7. The honest version: an authoritative change is live the instant the zone is published. What you are waiting for is other people's caches to *expire*. The correct mental model is expiry, not propagation. The practical consequence is the same waiting, but the fix is different: you cannot speed up propagation, but you can plan the TTL in advance.
8. There is one real exception. A change to the NS records must also be made at the parent registry, and registries publish their zones on a schedule, commonly every few minutes for com. That step genuinely is a publication delay rather than a cache expiry.
9. Useful commands:

```
dig +noall +answer github.com A          # see the TTL
dig +norecurse @1.1.1.1 github.com A     # cache hit or not
watch -n1 'dig +short +noall +answer github.com A'
sudo dscacheutil -flushcache             # macOS: clear stub cache
sudo killall -HUP mDNSResponder          # macOS: restart resolver
```

10. On macOS both of the last two lines are needed together. Flushing the directory service cache alone does not clear mDNSResponder. This has been true since macOS 10.10 and remains the standard pair today.

■ 25.7.6 WORDS - remember these

1. TTL — how many seconds you may reuse an answer — a 32-bit unsigned field per record, counted down by each cache that holds it.

2. Negative caching — remembering that a name does not exist — caching of NXDOMAIN and NODATA responses, bounded by the SOA minimum field, RFC 2308.
3. Cache hit — an answer given without asking anyone — a response served from local cache, identifiable by a TTL below the published maximum.
4. Propagation — the common but inaccurate word for waiting after a change — what is really happening is expiry of previously cached answers.
5. Authoritative TTL — the full value the zone owner publishes — the value seen when querying the authoritative server directly, never counted down.

25.8 Why the reader's machine uses 1.1.1.1

■ 25.8.1 PLAIN - in simple words

1. Your computer has to be told which resolver to use. It cannot guess.
2. Normally it is told automatically, by the router, at the moment it joins the network.
3. The router hands out a small bundle of settings: your address, the gateway to use, and the resolver to use. That bundle is called DHCP.
4. By default the router usually names itself as the resolver, at 192.168.0.1, and then quietly passes queries on to the internet provider.
5. The reader's Mac is not doing that. It is using 1.1.1.1 directly.
6. That happens for one of a few reasons: somebody set it by hand in network settings, an application set it, a privacy or VPN tool set it, or the provider's own resolver was slow or unreliable and it was changed.
7. The reader's session also showed several utun interfaces on the Mac, which are the kind of virtual network interfaces that VPN and tunnel software create. Such software very often changes the resolver as well.
8. That is a strong hint, but it is a hint, not proof. Only the machine's own settings can prove which of those it was.

■ 25.8.2 PLAIN - a picture in your head

1. Think of moving to a new town and needing a doctor.
2. When you register at the town hall, they give you a default doctor: the nearest surgery, chosen for you, no thought needed.
3. Most people keep the default forever, and it is fine.
4. Some people deliberately switch to a different practice: it opens earlier, or has better equipment, or they trust it more with their records.
5. There is a real trade. The default surgery is round the corner. The one you chose might be three towns away.
6. Also, the local surgery knows which local hospital to send you to. The distant one might send you to a hospital near itself instead of near you.

Where this comparison breaks: a doctor is a person with duties to you. A public resolver is a company with no contract with you at all, and its incentives may be quite different from yours. Also you can change resolver in four seconds and change back, whereas changing doctor is paperwork.

■ 25.8.3 PLAIN - a worked example

1. The three best known public resolvers, with real launch dates.

Resolver	Operator	Launched
8.8.8.8	Google	3 Dec 2009
9.9.9.9	Quad9	Nov 2017
1.1.1.1	Cloudflare, APNIC	1 Apr 2018

- Cloudflare announced 1.1.1.1 on 1 April 2018, which several people initially assumed was a joke because of the date. It was not.
- The address is not Cloudflare's own. It is held by APNIC, the regional internet registry for Asia and the Pacific, and operated jointly under a research agreement, because 1.1.1.1 had historically been so polluted with stray traffic that nobody could use it.
- Quad9 was announced in November 2017 by IBM Security, Packet Clearing House and the Global Cyber Alliance. It is now a Swiss foundation based in Zurich, and it blocks known malicious domains by default.
- Google Public DNS, 8.8.8.8, was announced on 3 December 2009 and by 2018 was handling over a trillion queries per day.
- Each also has a second address, so a single failure does not cut you off: 1.0.0.1, 8.8.4.4, and 149.112.112.112 respectively.

■ 25.8.4 PLAIN - what is really happening inside

- When the Mac joins the home network it broadcasts a DHCP request.
- The router replies with a DHCP offer containing numbered options. Option 3 is the gateway. Option 6 is the list of resolvers.
- For the reader, option 3 gave 192.168.0.1, the router. Something then overrode option 6 with 1.1.1.1.
- macOS keeps a per-interface resolver list. A manually set value in the network settings for Wi-Fi wins over whatever DHCP offered.
- A VPN or tunnel client goes further and installs its own resolver settings, often scoped so that some names go one way and others another.
- Now the important consequence. Your resolver is the thing that talks to authoritative servers. Those servers see the resolver's address, not yours.
- If your resolver is answering you from Mumbai, then a service that steers traffic by location sees a query from Mumbai, and answers as if you are in Mumbai. That is usually right and occasionally very wrong.

■ 25.8.5 TECHNICAL - the engineer's version

- The resolver list arrives as DHCP option 6, "Domain Name Server", defined in RFC 2132, March 1997. IPv6 uses a different mechanism, RDNSS in router advertisements, RFC 8106, March 2017.
- On macOS, inspect what is actually in force with these commands. Note that `/etc/resolv.conf` on macOS is generated and is not the real source of truth.

```
scutil --dns | grep -A2 'resolver #1'
networksetup -getdnsservers Wi-Fi
cat /etc/resolv.conf                # generated, informational only
```

- Honest trade-offs of using a public resolver instead of the provider's:

Aspect	Provider resolver	Public resolver
Latency to resolver	usually lower	often similar
CDN steering quality	usually better	can be worse
Filtering by ISP	common	depends
Query logging	ISP sees all	operator sees all
Encrypted transport	rare	usually offered

4. The most misunderstood item in that table is CDN steering. Content delivery networks decide which server to give you based on where the *query* appears to come from, which is the resolver, not you.
5. A provider's resolver sits inside the provider's network, and content networks have spent twenty years mapping providers' resolvers to providers' customers. It is a very good signal.
6. A large public resolver breaks that signal unless something replaces it.
7. The replacement is **EDNS Client Subnet**, defined in RFC 7871, published May 2016 as an Informational document. The resolver adds a truncated version of your address, typically the first 24 bits for IPv4, to the query it sends to the authoritative server.
8. That is enough for the content network to place you approximately, and not enough to identify you individually.
9. The honest version, and the RFC says this itself: EDNS Client Subnet is a privacy loss. The RFC recommends it be off by default and used only where the benefit is clear. Google Public DNS sends it to networks that ask. Cloudflare's 1.1.1.1 deliberately does not send it, and instead relies on having its own points of presence very close to users.
10. This is a genuine disagreement between two serious operators. Google's position is that better steering helps users more than the privacy cost. Cloudflare's position is that the privacy cost is not worth paying and that a dense anycast network makes it unnecessary. Both are defensible.
11. For the reader in India this matters concretely. Cloudflare operates points of presence in several Indian cities, so 1.1.1.1 answers from inside India, and the location signal is roughly correct even without client subnet. In a country with fewer points of presence the result can be an answer pointing at a server on another continent.

■ 25.8.6 WORDS - remember these

1. DHCP option 6 — how your machine is told which resolver to use — the Domain Name Server option in a DHCP offer, RFC 2132.
2. Public resolver — a resolver anyone may use — an open recursive resolver operated as a service, such as 1.1.1.1, 8.8.8.8 or 9.9.9.9.
3. EDNS Client Subnet — sending a rough version of your address upstream — an EDNS0 option carrying a truncated client prefix, RFC 7871.
4. CDN steering — choosing which server you are sent to — authoritative responses varied by the apparent location of the querying resolver.
5. scutil — the macOS tool that shows the real resolver configuration — the system configuration utility, queried with `scutil --dns`.

25.9 Why the same name gives different answers

■ 25.9.1 PLAIN - in simple words

1. Ask for `github.com` in India and you may get `20.207.73.82`. Ask in the United States and you may get `140.82.112.3`. Both are correct.

2. There is no single true answer to a DNS question for a large service. There is only the best answer for the asker, right now.
3. Four different mechanisms produce this, and they are constantly mixed up.
4. **Anycast** does it at the routing layer. Everyone gets the same address, and the network delivers them to a different machine.
5. **Geographic steering** does it at the DNS layer. Different people are given different addresses based on where they seem to be.
6. **Latency steering** also does it at the DNS layer, but based on measured speed rather than on a map.
7. **Weighted answers** do it at the DNS layer too, sending a chosen share of people to each of several addresses.
8. The difference matters because you fix them in completely different places.

■ 25.9.2 PLAIN - a picture in your head

1. Two ways to run a chain of shops with one advertised phone number.
2. Way one: print one number on every poster, and arrange with the phone company that calls are routed to the nearest branch automatically. Everyone dials the same digits. That is anycast.
3. Way two: print a different number on the posters in each city. Everyone dials different digits, and each set of digits reaches one specific branch. That is geographic DNS steering.
4. From the customer's side the experience looks identical. From the operator's side they are utterly different problems.
5. With way one, moving a customer to a different branch means changing routing and everyone in a region shifts at once.
6. With way two, moving a customer means changing what the directory says, and the change only reaches people whose cached answer has expired.

Where this comparison breaks: a phone company's routing is a private arrangement with one provider. Anycast routing decisions are made independently by thousands of networks with no coordination, using their own preferences, so the operator cannot fully predict who lands where.

■ 25.9.3 PLAIN - a worked example

1. Here is the difference in one table, using real observed data.

Mechanism	Layer	Who decides	Same reply?
Anycast	Routing	Every network	Yes
Geo steering	DNS	Authoritative server	No
Latency steering	DNS	Authoritative server	No
Weighted round robin	DNS	Authoritative server	No

2. 1.1.1.1 is pure anycast. Everybody in the world is told the same four digits. The routing system decides which of Cloudflare's locations answers.
3. github.com uses DNS steering. The reader in India was told 20.207.73.82. A machine in the United States was told 140.82.112.3 at the same time. Different digits, from the same question.
4. Both were checked and both are real. This is not a fault or an inconsistency.
5. To see it yourself, ask the authoritative server directly and compare with what a distant resolver returns.

```
$ dig +short @1.1.1.1 github.com A      # answers near you
$ dig +short @8.8.8.8 github.com A      # may answer differently
$ dig +short @dns1.p08.nsone.net github.com A
```

■ 25.9.4 PLAIN - what is really happening inside

1. With anycast, one address block is announced into the global routing system from many separate locations.
2. Every router on the internet learns several routes to that block and keeps the one its own policy prefers.
3. Your packet therefore arrives at whichever location the chain of routers between you and it happened to prefer. Nobody consulted a map.
4. Anycast is invisible to DNS. The DNS answer never changes. Two people in different countries can compare notes and see the same address while talking to different machines.
5. With geographic steering, the authoritative name server looks at the address of the resolver that asked, or at the client subnet if one was sent, decides which region that belongs to, and returns a different set of records.
6. This happens before any connection is made. The steering decision is baked into the answer, and it stays baked in for the length of the TTL.
7. That is why steered records almost always carry short TTLs. `github.com` uses 60 seconds. A steering decision that stuck for a day would be useless.
8. Latency steering is the same as geographic steering but the decision comes from continuous measurement rather than from a geography database. It reacts to real network conditions, including congestion.
9. Weighted answers simply divide traffic by proportion, for example 90 percent to a new stack and 10 percent to the old one during a migration.
10. Plain round robin, where a server returns several addresses and rotates the order, is the oldest and dumbest form. It still exists and it still works, but it balances queries, not load, and it has no idea if a server is down.

■ 25.9.5 TECHNICAL - the engineer's version

1. Anycast operates at layer 3 using BGP, the Border Gateway Protocol. The same prefix is originated from multiple sites, usually with the same origin AS number.
2. Failure behaviour is the key property. If a site withdraws its announcement, routers converge on the next best path, typically within seconds to a few tens of seconds, with no DNS change and no TTL to wait for.
3. The classic objection to anycast is that TCP sessions break if routing changes mid-connection. In practice this is rare, because routes are stable for the seconds a typical connection lasts, and operators use techniques such as consistent hashing at the site to limit the damage.
4. DNS-based steering is implemented by authoritative servers that vary answers per query. Common products and services doing this include NS1 Managed DNS, Amazon Route 53 with latency and geolocation routing policies, Akamai's authoritative service, and Azure Traffic Manager.
5. `github.com` is served by NS1 and Route 53 in parallel, both of which support per-query steering, which is consistent with the two very different answers observed in India and in the United States.
6. Route 53 alone offers simple, weighted, latency, failover, geolocation, geoproximity and multivalued answer policies. These are product features, an implementation detail, not standards.
7. The two layers combine. Cloudflare uses anycast for `1.1.1.1` and also returns different answers to different customers for hosted names. Microsoft fronts services with anycast edge addresses and also steers by DNS.
8. Diagnosing which one is in play, in order:

Symptom	Likely mechanism
Same IP everywhere, different rtt	Anycast
Different IP per country	DNS geo steering
IP changes every few minutes	Latency or weighted
Several IPs, order rotates	Round robin

9. The honest version: for a big service it is usually both at once, plus a load balancer inside the destination network doing a third kind of distribution that you cannot see from outside at all. Do not expect a clean single answer.

■ 25.9.6 WORDS – remember these

1. Anycast — one address served from many places — the same BGP prefix originated at multiple sites, with routing selecting the destination.
2. GeoDNS — different answers by location — authoritative responses selected by the apparent geography of the querying resolver or client subnet.
3. Latency steering — answers chosen by measured speed — a routing policy that returns the endpoint with the lowest observed round trip time.
4. Weighted answer — a chosen share of traffic per address — a policy that returns records in proportions set by the operator.
5. Round robin — rotating the order of several addresses — the original and simplest DNS load spreading, which balances queries rather than load.

25.10 The reader's own answer decoded

■ 25.10.1 PLAIN – in simple words

1. The reader asked for `github.com` and got `20.207.73.82`.
2. The first surprise is that this address does not belong to GitHub.
3. It belongs to Microsoft. That sounds wrong until you know one fact.
4. Microsoft bought GitHub. The deal was announced on 4 June 2018 and completed on 26 October 2018, for 7.5 billion United States dollars, paid in Microsoft shares rather than cash.
5. Since then, GitHub has been a Microsoft company, and a lot of GitHub's traffic is carried and fronted by Microsoft's own global network.
6. So a Microsoft address answering for `github.com` is not a mistake and not a hijack. It is the company's own infrastructure.
7. The second thing about this address is where it is. Everything points at India, and specifically at Pune.
8. The reader is in India. So this is the nearby doorway into Microsoft's network, which is exactly what should have been handed out.
9. That makes the reader's fault more interesting, not less. DNS did its job perfectly. The right nearby address was returned. What failed came later.

■ 25.10.2 PLAIN – a picture in your head

1. Imagine a well known bookshop chain that gets bought by a supermarket group.
2. The sign over the door still says the bookshop's name. Customers still call it the bookshop.
3. But the delivery lorries now say the supermarket's name, the warehouses are the supermarket's warehouses, and the loading bay you are sent to is a supermarket depot.
4. If you looked only at the lorry you would think you had the wrong shop.

5. You do not. You have the same shop, reached through a bigger company's logistics network.
6. And the depot you are sent to is the one near your town, not the one near the head office.

Where this comparison breaks: a supermarket owns its lorries outright and you can read the name on the side. On the internet you cannot see ownership directly. You have to look it up in a registry, and the registry records who was *assigned* the address, which is not always who is *using* it today.

■ 25.10.3 PLAIN - a worked example

1. Here is how to check the ownership yourself. Two independent methods, and they should agree.
2. Method one, the registry lookup. Ask the regional internet registry who was assigned this address.

```
| whois 20.207.73.82
```

3. The answer, checked in August 2026, contains these fields:

```
| NetRange: 20.192.0.0 - 20.255.255.255
| CIDR: 20.192.0.0/10
| NetName: MSFT
| OrgName: Microsoft Corporation
| Country: US
| RegDate: 2017-10-18
| Parent: NET20 (NET-20-0-0-0-0)
```

4. Method two, the routing lookup. Ask which network is actually announcing this address into the global routing system right now.

```
| prefix: 20.192.0.0/10
| origin: AS8075
| holder: MICROSOFT-CORP-MSN-AS-BLOCK, Microsoft Corporation
```

5. The two agree. The address is registered to Microsoft and is announced by Microsoft's own autonomous system number, 8075.
6. Now do the same for the answer a machine in the United States received for the same name, and see the contrast.

Address	Prefix	Announced by
20.207.73.82	20.192.0.0/10	AS8075 Microsoft
140.82.112.3	140.82.112.0/24	AS36459 GitHub

7. Same name, same moment, two different companies' networks. That is section 25.9 in action, and it is the clearest possible demonstration of it.

■ 25.10.4 PLAIN - what is really happening inside

1. Let us be strict about what is proven and what is only suggested, because this is exactly where people overclaim.
2. **Proven.** The address 20.207.73.82 falls inside 20.192.0.0/10, which the registry records as assigned to Microsoft Corporation, registered on 18 October 2017.
3. **Proven.** That prefix is announced into the global routing system by AS8075, Microsoft's autonomous system.
4. **Proven.** Microsoft acquired GitHub, announced 4 June 2018 and closed 26 October 2018, for 7.5 billion dollars in stock.

5. **Proven from the reader's own trace.** Every router from hop 7 onwards had a hostname ending in `ntwk.msn.net`, which is Microsoft's backbone naming. The final visible hops were `pnq20` and `pnq21`, Pune site codes.
6. **Strongly suggested.** The address is an India-facing edge. A public geolocation database places it in Pune, and the reader's own traceroute reached Microsoft routers in Pune. Two independent signals agree.
7. **Inferred, not proven.** That this specific address belongs to a particular Microsoft edge product. It is consistent with a Microsoft edge front end serving GitHub, but the address publishes no reverse DNS record at all, so there is nothing naming the service.
8. **Not knowable from outside.** Whether GitHub or Microsoft configured this steering, and which internal system chose it.
9. Notice how much can be established with two lookups and a traceroute, and how carefully the last two lines have to be worded.

■ 25.10.5 TECHNICAL - the engineer's version

1. Microsoft holds several very large IPv4 blocks under NetName MSFT. Two of them, both registered on 18 October 2017, are:

Block	Range	Size
20.0.0.0/11	20.0.0.0 - 20.31.255.255	2,097,152
20.192.0.0/10	20.192.0.0 - 20.255.255.255	4,194,304

2. The honest version: people say "20.0.0.0/8 is Microsoft". Large parts of it are, and both of the blocks above are, but the statement is loose. Check the specific prefix, not the first octet. The registry answer for the exact address is the only thing worth quoting.
3. IP registry lookups are separate from domain name lookups. For domains, WHOIS was formally retired for generic top level domains on 28 January 2025 in favour of RDAP, the Registration Data Access Protocol. For IP addresses the regional registries still answer WHOIS and also serve RDAP.
4. The RDAP equivalent of the lookup above is a plain HTTPS request to the registry's RDAP endpoint for the address, which returns JSON with the same fields as structured data rather than free text.
5. Reverse DNS on this address returns nothing:

```
$ dig +short -x 20.207.73.82
$
```

6. Compare with GitHub's own address space, which does publish reverse names and even encodes a site code:

```
$ dig +short -x 140.82.112.3
lb-140-82-112-3-iad.github.com.
```

7. `lb` for load balancer and `iad` for Washington Dulles. That naming is a convention, not a standard, but it is a widely used one and it is often the single most useful clue in a trace.
8. Putting the whole picture together for the reader's session:

```
Name asked:      github.com
Answered by:    1.1.1.1 (Cloudflare, anycast, likely in India)
Answer given:   20.207.73.82
Registered to:  Microsoft Corporation (20.192.0.0/10)
Announced by:  AS8075 Microsoft
```

```
Path observed:  ISP -> msn.net Delhi -> Mumbai -> Pune -> silence
Reverse DNS:    none published
Conclusion:     correct nearby Microsoft edge for GitHub
```

9. The last line matters for the rest of the reader's investigation. The name resolved, quickly, to a plausible and verifiable address near them. There was never a DNS fault. The failure was in reaching that address, which is a different layer and a different chapter.

■ 25.10.6 WORDS - remember these

1. AS number — the identifier of one network operator — an autonomous system number used in BGP, such as AS8075 for Microsoft.
2. Prefix — a block of addresses written as a range — an address plus a mask length, such as 20.192.0.0/10.
3. RIR — the body that hands out address blocks — a regional internet registry such as ARIN, RIPE NCC or APNIC.
4. RDAP — the modern replacement for WHOIS — the Registration Data Access Protocol, returning JSON over HTTPS, mandatory for gTLDs since Jan 2025.
5. Edge — the nearest point of a big network to you — a front end site that terminates connections close to users and carries traffic onward internally.

25.11 DNS on the wire

■ 25.11.1 PLAIN - in simple words

1. A DNS query is a small message sent over the network, usually as a single packet, and usually answered by a single packet coming back.
2. It goes to port 53. A port is a number that says which service on a machine the message is for.
3. It normally uses UDP, which is the fast, simple way to send a packet: fire and forget, no connection, no guarantee.
4. That choice is why DNS is quick. There is no handshake to set up and tear down for a question that fits in one packet.
5. It also means nothing is confirmed. If the packet is lost, nobody is told. The asker simply waits, gives up, and asks again.
6. If the answer is too big for one packet, the server sets a flag meaning "this was cut short", and the asker tries again over TCP.
7. TCP is the slower, careful way, with a connection and acknowledgements. Same port 53, different transport.
8. Every query carries a random number so the asker can match the answer to the question. That number is where a lot of the trouble in section 25.12 begins.

■ 25.11.2 PLAIN - a picture in your head

1. Think of a postcard versus a registered letter.
2. A postcard is cheap, quick, and you write the reference number of your enquiry on it so the reply can be matched up.
3. Nobody signs for a postcard. If it is lost in the sorting office, you find out only by noticing that no reply came.
4. That is UDP. Most DNS is postcards.
5. A registered letter needs a signature, a receipt, and several exchanges before anything is even delivered. It is much slower and much more certain.

- That is TCP. DNS uses it only when the reply will not fit on a postcard, or when the whole conversation needs to be private.

Where this comparison breaks: a postcard can be read by anyone handling it, and so can a plain DNS query, which is a fair part of the comparison. But a postcard cannot be forged into your letterbox by a stranger who guessed your reference number, and a DNS answer very much can. That is the attack described in the next section.

■ 25.11.3 PLAIN - a worked example

- A DNS message has a fixed 12-byte header followed by four sections.

+-----+		
	Header	12 bytes
+-----+		
	Question	what was asked
+-----+		
	Answer	the records asked
+-----+		
	Authority	who is in charge
+-----+		
	Additional	extras and glue
+-----+		

- In a normal successful reply, the answer section has the records you wanted and the other two are often empty.
- In a referral, the answer section is empty, the authority section has the NS records of the next level down, and the additional section has their addresses as glue.
- In a negative reply, the answer section is empty and the authority section has an SOA record, which supplies the negative caching time.
- Here is a real reply with those parts labelled by dig itself.

```
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 51021
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0,
;; ADDITIONAL: 1
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;github.com.                IN      A
;; ANSWER SECTION:
github.com.                 21      IN      A      20.207.73.82
;; Query time: 4 msec
;; SERVER: 1.1.1.1#53(1.1.1.1) (UDP)
;; MSG SIZE rcvd: 55
```

- Read the last line. The whole reply was 55 bytes. That is the normal size of DNS traffic, and it is why UDP suits it so well.

■ 25.11.4 PLAIN - what is really happening inside

- The header is only 12 bytes and every bit of it earns its place.

0		15
+-----+		
	Query ID	16 bits, random
+-----+		
	QR Opcode AA TC RD RA Z RCODE	flags
+-----+		

	QDCOUNT		questions
+-----+			
	ANCOUNT		answers
+-----+			
	NSCOUNT		authority records
+-----+			
	ARCOUNT		additional records
+-----+			

2. **Query ID** is a random 16-bit number. The reply must carry the same one or it is discarded.
3. **QR** says query or response. **AA** says the answer came from an authoritative server. **TC** says the reply was truncated.
4. **RD** and **RA** are the recursion flags from section 25.3.
5. **RCODE** is the result code: 0 for success, 2 for server failure, 3 for name does not exist, 5 for refused.
6. Sixteen bits of query ID means 65,536 possible values. Hold that number. It is the whole story of the next section.
7. The original limit on a UDP DNS message was 512 bytes of payload. Anything longer had to be truncated and retried over TCP.
8. That limit was raised by adding a fake record to the additional section that carries options and announces a larger acceptable size. This is EDNS0, and the fake record is called an OPT record. You can see it in the output above, labelled OPT PSEUDOSECTION.

■ 25.11.5 TECHNICAL - the engineer's version

1. Port 53 for both UDP and TCP was assigned in RFC 1035. TCP is mandatory to support, not optional, and RFC 7766, March 2016, made that requirement explicit and encouraged persistent TCP connections.
2. TCP is used in four situations: a truncated UDP reply, zone transfers (AXFR and IXFR), any reply the resolver knows will be large, and as the base for DNS over TLS.
3. EDNS0 was introduced in RFC 2671, August 1999, and revised as RFC 6891, April 2013. The OPT pseudo-record advertises the largest UDP payload the sender will accept.
4. Common advertised sizes and why they are argued about:

Size	Where it comes from
512 bytes	Original RFC 1035 limit
1232 bytes	DNS Flag Day 2020 recommendation
1400 bytes	Common older default
4096 bytes	Old BIND default, causes fragments

5. The 1232 figure comes from the smallest IPv6 MTU of 1280 minus 48 bytes of IPv6 and UDP headers. It was recommended by the DNS Flag Day 2020 effort precisely because larger values cause IP fragmentation, and fragmented UDP is dropped by many firewalls and is itself a security problem.
6. Response codes worth memorizing:

RCODE	Name	Meaning
0	NOERROR	Success, or NODATA
1	FORMERR	Malformed query
2	SERVFAIL	Server could not answer

RCODE	Name	Meaning
3	NXDOMAIN	Name does not exist
5	REFUSED	Policy refusal

7. SERVFAIL is the one that misleads people. It very often means a DNSSEC validation failure, not a broken server.
8. Name compression, RFC 1035 section 4.1.4, lets a name that has already appeared be replaced by a two-byte pointer. This is why thirteen root servers fit in 512 bytes and why message sizes are hard to predict.
9. The design flaw, stated plainly: the only things binding a reply to a query are the 16-bit ID, the source and destination addresses, the ports, and the question section. In the original design the source port was often fixed. That left roughly 16 bits of entropy for an attacker to guess.

■ 25.11.6 WORDS - remember these

1. Port 53 — the number DNS listens on — the IANA assigned port for DNS over both UDP and TCP.
2. Query ID — the number matching a reply to a question — a 16-bit field that must match, giving only 65,536 possibilities.
3. TC bit — “this reply was cut short” — the truncation flag, telling the client to retry over TCP.
4. EDNS0 — the extension that allows bigger replies and new options — an OPT pseudo-record in the additional section, RFC 6891.
5. SERVFAIL — “I could not answer” — RCODE 2, most often caused by a DNSSEC validation failure rather than an unreachable server.

25.12 DNS security and privacy

■ 25.12.1 PLAIN - in simple words

1. DNS was designed in 1983 for a network of a few hundred trusted machines. Nobody was expected to lie.
2. So the original design has no signatures, no encryption, and no way to tell a genuine answer from a forged one.
3. There are two separate problems, and mixing them up causes bad decisions.
4. Problem one is **integrity**: is this answer true, or did someone forge it?
5. Problem two is **privacy**: who else can see which names I am looking up?
6. The fix for integrity is DNSSEC, which signs records cryptographically.
7. The fix for privacy is encrypting the conversation between you and your resolver, using DNS over TLS or DNS over HTTPS.
8. They are independent. You can have either, both, or neither. DNSSEC does not hide anything, and encryption does not prove anything is true.
9. There is also a third topic which is not a bug at all: DNS is deliberately used to block things, by governments, by companies, and by security teams.

■ 25.12.2 PLAIN - a picture in your head

1. Think of asking a stranger in the street for directions.
2. The integrity problem: how do you know they are not lying and sending you to a pickpocket? A signature you can check is the answer. That is DNSSEC.

3. The privacy problem: everyone standing nearby heard you ask for the address of the bank. Whispering solves that. That is encrypted DNS.
4. Notice that whispering to a liar does not help, and getting a signed note from an honest person shouted across the street does not hide anything.
5. You need both to be safe on both counts, and they are separate purchases.

Where this comparison breaks: a stranger has no reputation and no infrastructure. Real resolvers are large operators with published policies and sometimes with audits. Also, whispering to one person means that one person now knows everything you asked, which is the real trade in encrypted DNS: you have narrowed the audience from many to one, but that one now sees all of it.

■ 25.12.3 PLAIN - a worked example

1. In 2008 Dan Kaminsky found a way to poison a resolver's cache reliably.
2. The idea, simplified: the attacker makes the resolver ask about a name that is certainly not cached, such as `aaa001.example.com`.
3. While the resolver waits, the attacker floods it with forged replies, guessing the 16-bit query ID.
4. The forged reply says "I do not know that name, but here is the address of the name server for the whole of `example.com`, and it is my machine".
5. If one guess lands before the real reply arrives, the resolver caches a poisoned delegation for the entire domain, not just one name.
6. If a guess fails, the attacker simply tries again with `aaa002.example.com`, and can keep trying thousands of times per second.
7. Work the numbers. With 65,536 possible IDs and thousands of attempts per second, a successful poisoning takes seconds to minutes, not centuries.
8. That is what made it serious. Earlier attacks needed luck. This one needed only patience.

■ 25.12.4 PLAIN - what is really happening inside

1. The fix deployed in 2008 was to randomize the source port as well as the ID.
2. The source port is another 16-bit number, so the attacker now has to guess both. That is roughly 32 bits instead of 16.
3. Two to the power of 32 is about 4.3 billion. The same attack that took minutes now takes an infeasibly long time on a normal network.
4. Note that this is a mitigation, not a cure. It makes forgery expensive rather than impossible.
5. The real cure is signing. With DNSSEC, each record set is signed by the zone owner's private key, and the matching public key is vouched for by the parent zone, all the way to the root.
6. The chain looks like this: the root signs a fingerprint of the key for `com`, `com` signs a fingerprint of the key for `github.com`, and `github.com` signs its own records.
7. If you trust the root's key, which is published and widely distributed, you can check every link. A forged answer fails the check and is discarded.
8. Encryption is a separate change entirely. It wraps the query in TLS so that the network between you and your resolver sees only that you contacted the resolver, not what you asked.

■ 25.12.5 TECHNICAL – the engineer’s version

1. The Kaminsky flaw is CVE-2008-1447. Vendors released a coordinated patch on 8 July 2008, the first large multi-vendor synchronized DNS patch. Details leaked on 21 July 2008, ahead of the intended embargo, and the full talk was given at Black Hat that August.
2. Source port randomization was formalized in RFC 5452, January 2009, “Measures for Making DNS More Resilient against Forged Answers”. Daniel J. Bernstein’s djbdns and PowerDNS had already been doing it for years.
3. DNSSEC core specifications are RFC 4033, RFC 4034 and RFC 4035, all published March 2005, replacing the earlier and unworkable RFC 2535 of 1999.
4. The record types added are DNSKEY, RRSIG, DS, NSEC and NSEC3.

Record	Job
DNSKEY	The zone’s public key
RRSIG	Signature over one RRset
DS	Hash of a child’s key, in parent
NSEC	Proof a name does not exist
NSEC3	Same, with hashed names

5. The root zone was signed on 15 July 2010. The key signing key was rolled for the first time on 11 October 2018.
6. Adoption is genuinely patchy and it is dishonest to pretend otherwise. As of 2025, about 48 percent of country code TLDs operate DNSSEC, but second level adoption is roughly 4 percent in com and 5 percent in net, while exceeding 50 percent in nl, cz, no, se and nu.
7. You can see a real DS record for com with dig:

```
$ dig +noall +answer com DS
com. 53825 IN DS 19718 13 2
      8ACBB0CD28F41250A80A491389424D341522D946B0DA0C029...
```

8. Experts disagree about DNSSEC. Supporters say it is the only real fix for forgery and enables things such as publishing certificate fingerprints in DNS. Critics say the complexity causes more outages than the attacks it prevents, that key management errors take whole domains offline, and that TLS already authenticates the destination. Both positions are held by serious engineers and both have evidence.
9. Encrypted transports:

Name	Port	Standard	Date
DNS over TLS	853	RFC 7858	May 2016
DNS over HTTPS	443	RFC 8484	Oct 2018
Oblivious DoH	443	RFC 9230	Jun 2022

10. DNS over TLS is easy for a network operator to see and to block, because it uses a dedicated port. DNS over HTTPS hides among ordinary web traffic on port 443 and is therefore very hard to distinguish or block.
11. That difference is the whole political argument. Firefox enabled DNS over HTTPS by default for United States users on 25 February 2020 and Chrome followed in May 2020, and network operators, schools and some governments objected loudly, because it removed their visibility and their filtering.

12. Oblivious DNS over HTTPS, RFC 9230, June 2022, splits the knowledge: a proxy sees your address but not your query, and the resolver sees your query but not your address. It is deployed but not widespread.
13. DNS-based filtering and blocking, stated neutrally, comes in three forms:
 1. Refusing to answer, returning NXDOMAIN or REFUSED for a blocked name. Used by Quad9 for malware, by corporate networks, and by regulators.
 2. Answering with a different address, sending users to a warning page.
 3. Sinkholing, where a known malicious name is deliberately pointed at a researcher's machine so infected computers can be counted and contained.
14. NXDOMAIN hijacking is when an internet provider replaces a nonexistent answer with an address of its own, usually showing search results and advertisements. The most famous case was Verisign's Site Finder, which did this for the whole of com and net from 15 September 2003 and was withdrawn on 4 October 2003 after heavy technical objection.
15. It breaks things beyond the browser. Any software that relies on NXDOMAIN to detect a typo or to finish a search list is confused by a fake address.
16. The most famous sinkhole is the WannaCry kill switch domain, registered on 12 May 2017 by the researcher Marcus Hutchins, which halted the spread of the ransomware because the malware checked whether that name resolved.

■ 25.12.6 WORDS - remember these

1. Cache poisoning — tricking a resolver into storing a false answer — off-path injection of forged responses matching a pending query.
2. Kaminsky attack — the 2008 method that made poisoning practical — repeated forced lookups of random subdomains with forged in-bailiwick delegations.
3. DNSSEC — cryptographic signatures on DNS records — a chain of trust from the root using DNSKEY, RRSIG and DS records, RFC 4033 to 4035.
4. DoT and DoH — encrypting the query to your resolver — DNS over TLS on port 853 and DNS over HTTPS on port 443.
5. Sinkholing — pointing a bad name at a controlled machine — deliberate redirection of malicious domains for containment and measurement.
6. NXDOMAIN hijacking — replacing “no such name” with an advert page — an operator substituting a synthesized A record for a negative response.

25.13 Owning a name

■ 25.13.1 PLAIN - in simple words

1. You cannot buy a domain name. You rent it, for a fixed period, from a chain of organizations.
2. Three roles matter, and they are almost always confused.
3. The **registrant** is you. The person or company the name is registered to.
4. The **registrar** is the shop you buy from. It takes your money, holds your account, and passes your instructions upward.
5. The **registry** runs the whole top level domain. There is exactly one registry per TLD. It holds the actual zone file.
6. So when you register `example.com`, you pay a registrar, and the registrar tells the com registry to add your name.
7. Above all of that sits ICANN, which decides which top level domains exist and accredits registrars, and IANA, which maintains the root zone contents.

8. Renting matters. If you stop paying, you lose the name, and someone else can take it. There is no ownership to fall back on.

■ 25.13.2 PLAIN - a picture in your head

1. Think of a market stall pitch in a covered market.
2. The market authority owns the building and decides how many halls there are. That is ICANN.
3. Each hall has one manager who keeps the master plan of every pitch in that hall. That is a registry.
4. You do not talk to the hall manager. You go to one of several letting agents who have a contract with the hall. That is a registrar.
5. You sign with the agent, pay yearly, and your name goes on the hall's master plan. You are the registrant.
6. If you stop paying, the pitch is cleared and relet. Your sign comes down and somebody else's goes up, in the same spot, with your old customers still walking towards it.

Where this comparison breaks: a market stall is physical and somebody would notice a stranger standing in it. A domain can be transferred away silently by someone who gets into your account with the agent, and the first sign is your email and your website quietly arriving somewhere else.

■ 25.13.3 PLAIN - a worked example

1. Here is what actually happens when a name is registered, in order.

1. You choose `example.com` at a registrar.
 2. Registrar checks availability with the `com` registry.
 3. You pay, and give contact details.
 4. Registrar creates the domain object at the registry, using the EPP protocol, with your NS records attached.
 5. Registry adds to the `com` zone:
`example.com. 172800 IN NS ns1.yourhost.net.`
`example.com. 172800 IN NS ns2.yourhost.net.`
 6. Registry republishes the `com` zone (minutes, not days).
 7. Your name servers now receive queries for the name.
 8. You add A, MX and TXT records at your DNS host.
2. Read step 5 again. The registry does not store your website address. It stores only the names of your name servers.
3. That is the delegation. The `com` zone contains hundreds of millions of NS records and almost no A records for customer domains.
4. Step 8 is at a completely different company. Your DNS hosting can be at Cloudflare, Route 53 or NS1 while your registrar is somewhere else entirely.
5. `github.com` shows this split clearly: the delegation at `com` names servers at NS1 and at Amazon Route 53, neither of which is a registrar.

■ 25.13.4 PLAIN - what is really happening inside

1. There are two copies of your name server list, and they must agree.
2. One is at the parent, in the `com` zone, put there by your registrar. This is the one the world actually follows.
3. The other is inside your own zone, as NS records at your apex. This one is what your own servers claim.
4. If they disagree, resolvers follow the parent, and you get confusing half-working behaviour that is hard to debug. Check both when in doubt.

5. When a name expires, it does not vanish that day. There is a sequence of grace periods, and it is worth knowing because it decides whether you can still get your name back.
6. Roughly: the name stops working, then there is a period where you can renew normally, then a redemption period where you can recover it at a much higher fee, then a short pending-delete window, and then it is released.
7. Domain hijacking usually has nothing to do with DNS itself. It is theft of the registrar account, or a fraudulent transfer request, or a compromised email address used for password resets.

■ 25.13.5 TECHNICAL - the engineer's version

1. ICANN, the Internet Corporation for Assigned Names and Numbers, was formed on 30 September 1998 as a California non-profit corporation.
2. The IANA functions, including maintaining the root zone, are performed by Public Technical Identifiers, an ICANN affiliate created when United States government stewardship ended on 1 October 2016.
3. Registrars communicate with registries using EPP, the Extensible Provisioning Protocol, RFC 5730 to RFC 5734, published August 2009.
4. Example registries:

TLD	Registry	Note
com, net	Verisign	Also root zone maintainer
org	Public Interest Registry	Non-profit
uk	Nominet	UK, since 1996
in	NIXI	India

5. The standard gTLD lifecycle, as set by ICANN policy:

Phase	Typical length
Registered term	1 to 10 years
Auto-renew grace	0 to 45 days
Redemption grace	30 days
Pending delete	5 days
Add grace period	5 days

6. The Add Grace Period allows a registrar to delete a new registration for a refund within five days. This was abused at enormous scale around 2007 in a practice called domain tasting, and ICANN largely ended it in 2009 by charging fees on excessive AGP deletions.
7. WHOIS changed fundamentally in 2018. The GDPR took effect on 25 May 2018, and ICANN adopted a Temporary Specification in May 2018 requiring registrars to redact most registrant contact data from public WHOIS output.
8. Before that, a WHOIS query returned a name, postal address, phone number and email for most domains. After it, most gTLD records show only the country, the state, and an anonymized contact form or relay address.
9. WHOIS itself was formally sunset for generic top level domains on 28 January 2025, replaced by RDAP, which returns structured JSON over HTTPS and supports differentiated access for authorized parties.
10. Protection against hijacking is layered, and the important controls are:

1. EPP status codes such as `clientTransferProhibited` and `clientUpdateProhibited`, set by the registrar on your instruction.
 2. Registry lock, a manual out-of-band process at the registry level, offered for high value names. Changes require human verification.
 3. Two-factor authentication on the registrar account, and a dedicated email address that is not hosted on the domain itself.
11. Real cases worth knowing. On 15 January 2005 the New York provider `panix.com` was transferred away without authorization and its mail and web traffic redirected for most of a day. On 27 August 2013 the Syrian Electronic Army altered DNS delegations for the New York Times and other sites through a compromised reseller of the registrar Melbourne IT.
 12. Both were registrar-level compromises. No DNS protocol weakness was involved in either. That is the pattern.

■ 25.13.6 WORDS - remember these

1. Registrant — the person or company a name is registered to — the legal holder of a domain registration for its term.
2. Registrar — the shop you buy the name from — an ICANN-accredited entity that creates and manages domain objects at a registry.
3. Registry — the operator of a whole top level domain — the sole operator of a TLD zone, such as Verisign for `com`.
4. EPP — the protocol registrars use to talk to registries — the Extensible Provisioning Protocol, RFC 5730 to 5734.
5. Redemption grace period — the expensive last chance to get an expired name back — a 30-day ICANN-mandated recovery window before pending delete.
6. Registry lock — the strongest anti-hijack control — a registry-level status requiring manual out-of-band authorization for any change.

25.14 Diagnosing DNS for real

■ 25.14.1 PLAIN - in simple words

1. Most problems blamed on DNS are not DNS. The fastest thing you can do is prove whether it is or not, in one command.
2. The test is simple. Does the name turn into an address, quickly, with a sensible value?
3. If yes, DNS worked, and you must look somewhere else. Stop blaming DNS.
4. If no, then find out where it failed: your own machine, your resolver, or the authoritative servers.
5. The main tool is `dig`. It is precise, it shows everything, and it does not hide anything from you.
6. `nslookup` is older and friendlier and lies to you a little by simplifying. `host` is the short one for a quick answer.
7. On macOS there are two extra commands you need, because macOS caches in a place the others do not touch.
8. The reader's own case is a perfect worked example, and the answer is clean: DNS was never the problem.

■ 25.14.2 PLAIN - a picture in your head

1. Think of a parcel that never arrived.
2. The first question is not "was it stolen". It is "did the address get written on it correctly".
3. If the label on the tracking record shows the right house number, then the address lookup worked and the parcel was lost on the road.

4. If the label shows no address at all, the failure happened at the desk before the parcel ever moved.
5. Those are two entirely different investigations, and doing them in the wrong order wastes hours.
6. In DNS the label is the address `curl` prints before it starts connecting.

Where this comparison breaks: with a parcel you can ask the depot what happened. On the internet, a silent drop tells you nothing at all, and the absence of any reply is itself the most common evidence you will get. You must reason from what did not happen.

■ 25.14.3 PLAIN - a worked example

1. Here is the reader's own session, and the exact line that settles it.

```
$ curl -v https://github.com
* Trying 20.207.73.82:443...
... 15 seconds of nothing ...
* Connection timed out after 15001 milliseconds
```

2. Look at the first line. `curl` printed an IP address.
3. To print that address, `curl` had to resolve `github.com` first. It did, and it did so fast enough that the delay is not visible.
4. Therefore DNS succeeded. Completely. Before the timeout ever started.
5. The 15 seconds of silence happened during the TCP connection attempt, which is a later and separate step.
6. This is the single most useful diagnostic habit in networking: read whether your tool printed a name or a number. A number means resolution finished.
7. For contrast, a genuine DNS failure looks like this, and it fails in under a second rather than after fifteen.

```
$ curl -v https://githubb.com
* Could not resolve host: githubb.com
* Closing connection
curl: (6) Could not resolve host: githubb.com
```

8. Different message, different timing, different layer, different fix.

■ 25.14.4 PLAIN - what is really happening inside

1. Work down the chain, one layer at a time, and stop at the first thing that fails.
2. Layer one, your own cache. Ask twice and see if the answer is instant. If a stale answer is suspected, flush it.
3. Layer two, your configured resolver. Ask it directly by name, with `@`.
4. Layer three, a different resolver. If `1.1.1.1` fails and `8.8.8.8` works, the problem is your resolver or the path to it, not the domain.
5. Layer four, the authoritative servers. Ask them directly. If they answer and your resolver does not, the problem is in between, often DNSSEC.
6. Layer five, the delegation itself. Check that the NS records at the parent match the NS records in the zone.
7. Each of those is one command, and the whole sequence takes under a minute.

■ 25.14.5 TECHNICAL - the engineer's version

1. The `dig` flags that actually matter, with what each is for.

```
dig github.com A                # the full, normal query
dig +short github.com          # just the answer, for scripts
```



```

dig +noall +answer github.com      # answer section only, with TTL
dig @1.1.1.1 github.com            # ask a specific resolver
dig @dns1.p08.nsone.net github.com # ask the authoritative server
dig +trace github.com              # walk the whole delegation chain
dig +norecurse @1.1.1.1 github.com # cache hit test, RD cleared
dig +dnssec github.com             # request signatures, see AD flag
dig +tcp github.com                # force TCP instead of UDP
dig -x 20.207.73.82                # reverse lookup
dig github.com ANY                 # mostly useless now, see below
dig +nssearch github.com           # find and query all its NS

```

2. `+trace` is the one to reach for when a delegation looks wrong. It shows every referral, so you can see exactly which level gives the bad answer.
3. `+norecurse` against a recursive resolver is the cache test. If you get a full answer with the RD bit clear, it was cached. If you get nothing, it was not.
4. `dig ANY` is largely dead. RFC 8482, January 2019, allows servers to return a minimal response to ANY queries, and Cloudflare and others do exactly that, because ANY was heavily used for amplification attacks.
5. The other tools:

Tool	Best for	Caution
<code>dig</code>	Everything, precisely	Verbose by default
<code>host</code>	One quick answer	Hides detail
<code>nslookup</code>	Interactive, on Windows	Own resolver logic
<code>dscacheutil</code>	macOS cache view	Not the whole cache
<code>scutil -dns</code>	Real macOS config	Long output

6. `nslookup` deserves the warning. It has its own resolution behaviour and does not always use the system resolver the way applications do, so an answer from `nslookup` is not proof that an application will get the same.
7. macOS cache handling, and both lines are needed:

```

sudo dscacheutil -flushcache
sudo killall -HUP mDNSResponder

```

8. A single command that separates DNS from connectivity, because it does the two steps in order and reports each one:

```

dig +short github.com && \
nc -vz -w 5 $(dig +short github.com | head -1) 443

```

9. If the first part prints an address and the second part times out, DNS is fine and the transport is not. That is precisely the reader's situation.
10. Applied to the reader's session, stated as findings:
 1. `github.com` resolved to `20.207.73.82`. Resolution succeeded.
 2. The address is verifiably Microsoft's, registered and announced by AS8075, and geolocates to Pune, near the reader.
 3. `curl` printed the address and then received nothing for 15 seconds.
 4. The traceroute reached Microsoft routers in Pune and then stopped.
 5. The same request succeeded immediately over mobile data.
11. Therefore DNS worked perfectly and can be excluded with confidence. The fault lies on the path between the reader's fixed line and that address.

12. Be careful about the last step. The evidence supports a silent drop somewhere on that path. It does not identify who dropped it, and it does not support saying the server was down, because the same address answered fine over a different access network minutes later.

■ 25.14.6 WORDS - remember these

1. `dig` — the standard DNS query tool — domain information groper, from BIND, showing the full message including flags and timings.
2. `+trace` — walk the tree yourself — `dig` mode that issues iterative queries from the root and prints every referral.
3. `+short` — answer only — `dig` output mode giving bare record data, suitable for scripts.
4. `+norecurse` — ask without asking for work — clears the RD bit, used to test whether a resolver already holds an answer in cache.
5. `mDNSResponder` — the macOS resolver service — the system daemon handling both unicast DNS caching and Multicast DNS.

25.98 Common wrong ideas

1. Wrong: my computer walks the DNS tree and asks the root servers. Right: your computer asks one resolver, once. The resolver walks the tree. Your machine almost never speaks to a root server in its whole life.
2. Wrong: there are thirteen root servers. Right: there are thirteen root server *names* and addresses, answered by over 1,900 physical instances worldwide, 2,003 of them on 13 August 2026.
3. Wrong: a DNS change has to propagate around the world. Right: nothing is pushed anywhere. The change is live immediately at the authoritative servers. You are waiting for other people's caches to expire.
4. Wrong: TTL guarantees how long an answer is kept. Right: TTL is a maximum. A cache may drop an answer sooner, and some browsers and applications ignore short TTLs entirely.
5. Wrong: `www.example.com` and `example.com` are the same name. Right: they are two different names. They often point at the same place, but only because somebody configured records for both.
6. Wrong: I can put a CNAME at the top of my domain. Right: you cannot. The apex must hold SOA and NS records, and RFC 1034 forbids any other data alongside a CNAME. The alternatives are provider features, not standards.
7. Wrong: DNSSEC encrypts my DNS queries. Right: DNSSEC signs answers so forgery can be detected. It hides nothing. Encryption is a separate thing, DNS over TLS or DNS over HTTPS.
8. Wrong: using 1.1.1.1 always makes browsing faster. Right: it usually makes lookups fast, but it can give you a content server further away than your provider's resolver would have, because location steering is based on the resolver, not on you.
9. Wrong: the reverse lookup of an address should give back the name you started with. Right: forward and reverse are controlled by different parties and often disagree, and many addresses publish no reverse record at all. The reader's own `20.207.73.82` publishes none.
10. Wrong: a SERVFAIL means the server is down. Right: it means the resolver could not produce an answer, and by far the most common cause today is a DNSSEC validation failure.

25.99 Chapter summary in 20 lines

1. DNS turns names people remember into numbers machines route to.
2. Before it, one hand-edited file, `HOSTS.TXT`, was copied by every machine from Stanford Research Institute, and it could not scale.
3. Paul Mockapetris designed DNS in 1983, published as RFC 882 and RFC 883, replaced in November 1987 by RFC 1034 and RFC 1035, still the base today.
4. Names are a tree read right to left, from an invisible root written as a single dot, through top level domains, down to whatever you create.
5. Four participants: a stub resolver in your machine, a recursive resolver that does all the work, and authoritative servers in three tiers.
6. Your machine asks once and waits. The recursive resolver does the running around. That single sentence prevents most confusion about DNS.
7. A cold lookup of `github.com` takes about eight packets and roughly 90 milliseconds; a warm one takes two packets and under 10 milliseconds.
8. Thirteen root names, A to M, run by twelve organizations, served by 2,003 instances as of 13 August 2026, using anycast. Thirteen was a 512-byte limit.
9. Record types each answer a different question: A and AAAA for addresses, CNAME for aliases, MX for mail, NS for delegation, SOA for zone settings.
10. A CNAME cannot sit at the apex of a domain, because SOA and NS must be there and a CNAME excludes all other data. This catches everyone once.
11. Caching is what makes DNS survive. Every answer carries a TTL, which is a promise of a maximum reuse time, not a guarantee.
12. Lower the TTL at least two days before a migration, make the change, then raise it again. There is no propagation, only expiry.
13. Your resolver normally arrives by DHCP option 6 from the router. The reader overrode this and uses `1.1.1.1`, launched by Cloudflare on 1 April 2018.
14. Public resolvers trade local knowledge for scale and privacy. EDNS Client Subnet, RFC 7871, partly restores the location signal, at a privacy cost.
15. Different people get different answers for two separate reasons: anycast at the routing layer, and geographic or latency steering at the DNS layer.
16. The reader's `github.com` resolved to `20.207.73.82`, inside `20.192.0.0/10`, registered to Microsoft and announced by AS8075.
17. That is correct: Microsoft acquired GitHub for 7.5 billion dollars, announced 4 June 2018 and closed 26 October 2018, and fronts its traffic.
18. On the wire DNS is UDP port 53, a 12-byte header and four sections, with a 16-bit query ID that made the 2008 Kaminsky cache poisoning attack possible.
19. DNSSEC signs answers, from the root signed on 15 July 2010; DNS over TLS on port 853 and DNS over HTTPS on port 443 encrypt them. They are separate.
20. In the reader's own fault, DNS worked perfectly: the name resolved fast to a correct nearby address, so the failure was definitely somewhere later.

Packets and Routing

26.0 What this chapter gives you

1. You will be able to explain why data is chopped into small pieces at all, instead of being sent as one long stream, and name the four separate reasons.
2. You will be able to draw a real packet, layer by layer, with byte counts, and work out how much of the wire is spent on wrapping rather than content.
3. You will be able to read every field of an IPv4 header, say what each one is for, and decode a real header from the reader's laptop by hand.
4. You will be able to explain what happens when a packet is too big for a link, what IPv4 does, what IPv6 refuses to do, and why a filtered ICMP message can make a connection die in silence.
5. You will be able to list, in order, the eight things a router does to a packet, and say which addresses change at every hop and which never change.
6. You will be able to read a real routing table, apply longest prefix match to a real address, and explain the reader's default route through 192.168.0.1.
7. You will be able to say how routers learn routes, from static lines typed by hand up to BGP, and explain what convergence is and why it is not instant.
8. You will be able to explain autonomous systems, transit, peering and internet exchanges, look up which network owns an address, and read AS8075 correctly.
9. You will be able to explain why BGP is about money and policy rather than distance, and describe three real incidents where it went badly wrong.
10. You will be able to explain why the reader can see twelve hops outward and still know nothing at all about the way back.

26.1 Why data is split up at all

■ 26.1.1 PLAIN – in simple words

1. When your computer sends something big, it does not send it as one lump.
2. It cuts the data into many small pieces and sends the pieces one after another.
3. Each piece is called a **packet**. A packet is a small parcel of data with a label on the front saying where it is going.
4. A typical packet on a home network carries about 1,500 bytes at most.
5. There are four reasons for doing it this way, and all four matter.
6. Reason one is fairness. A wire is shared. If one person sent a whole film in one unbroken lump, everyone else would wait until it finished.
7. Reason two is the cost of mistakes. If something goes wrong, you only resend the small piece that was damaged, not the whole file.

8. Reason three is memory. A machine in the middle only needs room to hold a few small pieces at a time, not the whole file.
9. Reason four is mixing. Small pieces from many different conversations can be interleaved on one wire, so many things happen at once.

■ 26.1.2 PLAIN – a picture in your head

1. Think about how you would move a house full of furniture across a country.
2. You would pack it into boxes. Each box has a label with the new address.
3. The boxes travel separately. Some go on one lorry, some on another.
4. If one box is dropped, you replace the contents of that box only. The other boxes are unaffected.
5. Other people's boxes ride on the same lorries at the same time. The lorry is never held for one customer.
6. This is exactly what packets do, and it is why the postal system is a much better picture of a network than a pipe is.
7. A pipe suggests a continuous stream that arrives in order, complete, with nothing else in it. None of that is true of a network.
8. Where this comparison breaks: real parcels are almost never lost or reordered, and the post office never sends two boxes by two different routes on purpose. A network does all three, routinely, every second.

■ 26.1.3 PLAIN – a worked example

1. Suppose two people share one 100 megabit per second link.
2. Person A wants to send a one gigabyte file. Person B wants to load a web page of 50 kilobytes.
3. One gigabyte is 8,000,000,000 bits. At 100 megabits per second that takes 80 seconds to push onto the wire.
4. If the link carried whole files one at a time, person B would wait 80 seconds for a page that needs half a second of transfer.
5. With packets, the link carries a piece from A, then a piece from B, then a piece from A, and so on. Person B finishes in about a second.
6. Now the error case. Assume 1 packet in 1,000 is lost.
7. Sent as packets of 1,448 bytes each, the file is 690,608 packets, and one lost packet costs 1,448 bytes, which is about 0.116 milliseconds of link time.

Way of sending	Cost of one loss	Wait for other user
One 1 GB lump	1 GB resent, 80 s	Up to 80 s
1,448 B packets	1,448 B resent	About 0.12 ms

■ 26.1.4 PLAIN – what is really happening inside

1. There are two old ways to share a network, and they are worth naming clearly.
2. The first is **circuit switching**. Before you send anything, the network reserves a path from end to end and holds it for you until you hang up.
3. The classic telephone call worked this way. When you dialled, exchanges along the way set up a chain of connections, and that chain was yours alone.
4. A voice circuit was fixed at 64,000 bits per second. That capacity was yours whether you were talking or silent.
5. The bad part: silence is wasted. Two people on a call are silent about half the time, and that half of the capacity is thrown away.

6. The worse part: setup takes time, and if any link on the reserved path fails, the call drops. There is no way to slip round the break.
7. The second way is **packet switching**. Nothing is reserved. Every packet carries its own destination address and is dealt with on arrival.
8. A machine in the middle looks at the address, picks an outgoing link, and sends it. If the link is busy, the packet waits briefly in a queue.
9. Nothing about the earlier packets is remembered. Each packet is judged on its own. That property is called being **stateless**.
10. The bad part: delay is not steady. A packet may wait behind others. Under heavy load, packets are dropped rather than delayed forever.

■ 26.1.5 TECHNICAL – the engineer’s version

1. Packet switching was invented twice, independently, in the early 1960s.
2. Leonard Kleinrock published queueing analysis of message flow in large communication networks at MIT in 1961, later expanded into his 1962 thesis.
3. Paul Baran at the RAND Corporation published the eleven-volume report On Distributed Communications in August 1964, describing what he called distributed adaptive message block switching, designed to survive damage.
4. Donald Davies at the National Physical Laboratory in the United Kingdom reached the same design in 1965 and gave it the name that stuck: **packet**.
5. The ARPANET, the first large working packet network, carried its first message on 29 October 1969 from UCLA to the Stanford Research Institute.
6. Vint Cerf and Robert Kahn published A Protocol for Packet Network Intercommunication in IEEE Transactions on Communications in May 1974, which is the direct ancestor of TCP and IP.
7. The ARPANET switched from the older NCP to TCP/IP on 1 January 1983, a single cut-over now remembered as flag day.
8. Serialization delay is the time to clock a frame onto the wire, and it is exactly frame bits divided by link rate.

Link rate	1,518 B frame	64 B frame
10 Mbit/s	1,214 us	51.2 us
100 Mbit/s	121.4 us	5.12 us
1 Gbit/s	12.14 us	0.512 us
10 Gbit/s	1.214 us	0.0512 us

9. Tools: `ping -s` varies packet size, `iperf3` measures throughput, and `tc qdisc` on Linux shows the queue behaviour underneath.

■ 26.1.6 WORDS – remember these

1. Packet — a small labelled parcel of data — a protocol data unit carrying a header and a payload, forwarded independently.
2. Circuit switching — reserve a path first, then talk — connection-oriented networking with resources committed for the call duration.
3. Packet switching — no reservation, address on every parcel — connectionless store-and-forward networking with per-packet forwarding decisions.
4. Multiplexing — many conversations sharing one wire — statistical multiplexing of independent flows onto a shared link.

5. Serialization delay — the time to push a packet onto the wire — frame size in bits divided by the link rate in bits per second.

26.2 What a packet actually is

■ 26.2.1 PLAIN – in simple words

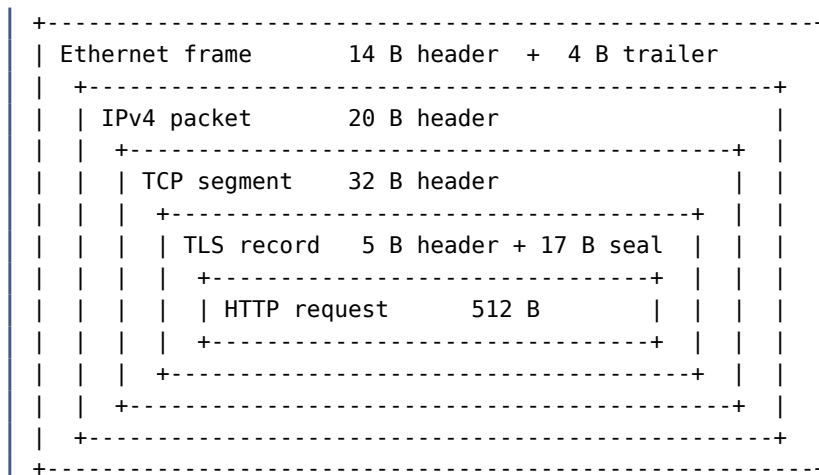
1. Every packet has exactly two parts: a **header** and a **payload**.
2. The header is the label on the outside. It says where the packet is going, where it came from, and what kind of thing is inside.
3. The payload is the thing being carried. The network does not read it.
4. Here is the part that surprises people: the payload of one packet is usually another whole packet, with its own header.
5. The outermost label is used by the local wire. The next one in is used by the routers. The next one in is used by the two computers at the ends.
6. The next one in is used by the encryption layer. The innermost one is the thing you actually wanted, like a web request.
7. Putting a box inside a box is called **encapsulation**.
8. Taking the boxes apart again at the far end is called **decapsulation**.

■ 26.2.2 PLAIN – a picture in your head

1. Think of a diplomatic pouch being sent between two countries.
2. Inside the outer canvas sack there is a sealed courier bag.
3. Inside the envelope there is a single sheet of paper with the message.
4. The airline handles the sack. It reads the sack's tag and nothing else.
5. The courier company handles the bag. It reads the bag's tag and nothing else.
6. The embassy opens the case. The ambassador opens the envelope. Only the ambassador reads the paper.
7. Where this comparison breaks: in a real network the outer wrapper is thrown away and rebuilt at every single hop, not carried end to end. The airline tag is replaced at every airport. That is section 26.5.

■ 26.2.3 PLAIN – a worked example

1. Let us draw one real packet: the reader's laptop sending an HTTPS request to github.com at 20.207.73.82.
2. Assume the request itself, the plain text a web server would read, is 512 bytes.



3. Now add it up from the inside out.

Layer	Adds	Running total
HTTP request	512 B	512 B
TLS 1.3 record	22 B	534 B
TCP segment	32 B	566 B
IPv4 packet	20 B	586 B
Ethernet frame	18 B	604 B

- The TCP 32 bytes are the fixed 20 byte header plus 12 bytes of options, which on a modern machine are two padding bytes and a timestamp.
- So 604 bytes go on the wire to deliver 512 useful bytes.
- The overhead is 604 minus 512, which is 92 bytes, or 15.2 percent.

■ 26.2.4 PLAIN – what is really happening inside

- Encapsulation happens in a fixed order on the way out, top to bottom.
- The browser writes the HTTP request bytes and hands them to the TLS layer.
- TCP cuts the stream into segments no bigger than the agreed maximum, puts its header on each one, and hands each segment to IP.
- IP puts its own 20 byte header on the front, with the source and destination addresses, and hands the packet to the network card driver.
- The driver wraps it in an Ethernet or Wi-Fi frame with hardware addresses, and the card turns the bits into signals.
- Decapsulation at the far end is the same list read backwards.
- IP checks the destination is itself, strips the IP header, reads the protocol number, and hands the rest to TCP.
- TCP checks the port number, strips its header, puts the bytes back in order, and hands a byte stream to TLS.

■ 26.2.5 TECHNICAL – the engineer's version

- Ethernet II framing carries a 6 byte destination MAC, a 6 byte source MAC and a 2 byte EtherType, followed by payload and a 4 byte frame check sequence.
- EtherType 0x0800 means IPv4, 0x86DD means IPv6 and 0x0806 means ARP. These are assigned by the IEEE Registration Authority.
- The standard Ethernet payload maximum is 1,500 bytes, so the standard frame is 1,518 bytes including the check sequence, and the wire slot is 1,538.
- On 802.11 Wi-Fi the picture differs: a data frame header is 24 to 30 bytes plus an 8 byte LLC/SNAP shim, so the same IP packet costs more.
- The reader is on home broadband over a router at 192.168.0.1. If the laptop is on Wi-Fi, the first hop uses 802.11 framing and only the wired hops beyond the router use Ethernet framing. The IP packet inside is identical.
- TLS 1.3, specified in RFC 8446 from August 2018, adds 5 bytes of record header, 1 byte of inner content type and a 16 byte AEAD tag per record.

Packet kind	Useful bytes	Frame bytes	Overhead
Pure TCP ack	0	70	100 percent

Packet kind	Useful bytes	Frame bytes	Overhead
Small request	512	604	15.2 percent
Full segment	1,426	1,518	6.1 percent

7. Tools: `tcpdump -e -x` prints the frame header and the raw bytes; Wireshark shows the same nesting as a tree you can expand layer by layer.

■ 26.2.6 WORDS – remember these

1. Header — the label on the front — the fixed-format control fields a protocol layer prepends to the data it is carrying.
2. Payload — the thing being carried — the opaque body of a protocol data unit, normally the entire packet of the layer above.
3. Encapsulation — putting a box inside a box — wrapping an upper-layer PDU as the payload of a lower-layer PDU on transmission.
4. Decapsulation — unwrapping — stripping a header and demultiplexing the payload to the correct upper-layer protocol on reception.
5. Frame — the outermost wrapper on one wire — a link-layer PDU with hardware addresses, valid only between two directly connected devices.

26.3 The IPv4 header field by field

■ 26.3.1 PLAIN – in simple words

1. The IP header is the only part of the packet that routers in the middle read.
2. It is 20 bytes long in almost every packet you will ever see.
3. The two most important fields are the source address and the destination address. Together they take 8 of the 20 bytes.
4. The next most important is a countdown number that stops packets circling forever if the routers get confused.
5. There is a number saying what is inside, so the receiver knows whether to hand the payload to TCP, to UDP, or to something else.
6. The rest is machinery for cutting a packet into smaller pieces, plus a couple of bits for marking traffic as more or less urgent.

■ 26.3.2 PLAIN – a picture in your head

1. Picture a customs declaration form stuck to the outside of a parcel.
2. Line one says which version of the form this is, so the clerk knows the layout.
3. Line three says how urgent the parcel is and whether the depot was crowded.
4. Lines five to seven exist only for parcels that had to be split: a batch number, a flag saying whether more pieces follow, and where this piece sits.
5. Line eight is a stamp saying how many more depots this parcel may pass through before being thrown away.
6. Line nine says what is inside so the right department gets it.
7. Where this comparison breaks: a customs form is read once. An IP header is read, altered and re-stamped at every single depot, and the checksum has to be recomputed each time.

■ 26.3.3 PLAIN – a worked example

1. Here is the IP header of the first packet the reader's laptop sent towards `github.com`, the TCP SYN that opens the connection to port 443.
2. The laptop's own address on the home network is not recorded in the session notes, so we write it as `192.168.0.14`. Treat that one number as a stand-in. Every other value here is exactly what such a packet carries.

```
0000 45 00 00 40 4a 3b 40 00 40 06 d1 a5 c0 a8 00 0e
0010 14 cf 49 52
```

3. Read it byte by byte.

Bytes	Value	Meaning
45	4, 5	IPv4, header is 20 bytes
00	0, 0	No priority, no congestion
00 40	64	Total length 64 bytes
4a 3b	19003	Identification
40 00	DF set	Do not fragment, offset 0
40	64	TTL 64 hops left
06	6	Payload is TCP
d1 a5	0xd1a5	Header checksum
c0 a8 00 0e	192.168.0.14	Source
14 cf 49 52	20.207.73.82	Destination

4. Total length 64 makes sense: a 20 byte IP header plus a 44 byte TCP header, because a SYN carries options and no data.
5. TTL 64 is the macOS and Linux default. Windows uses 128. Many routers use 255 for their own packets. These are conventions, not requirements.
6. When the reader's router at `192.168.0.1` forwards this packet, TTL drops to 63 and the checksum becomes `0xd2a5`.

■ 26.3.4 PLAIN – what is really happening inside

1. The checksum is a one's complement sum, and it is worth seeing once.
2. Split the header into ten 16 bit words, with the checksum field set to zero.
3. Add them all. Whenever the total goes past 16 bits, wrap the carry round and add it back at the bottom.
4. Flip every bit of the result. That is the checksum you store.
5. For the header above the running total is `0x2e5a`, and flipping every bit gives `0xd1a5`, which is exactly what is in the packet.
6. A receiver checks it by summing all ten words including the checksum. If the header is intact the answer is always `0xffff`.
7. This checksum covers the header only, not the payload. TCP and UDP have their own checksums for the data.
8. That split is intentional. Routers change header fields at every hop, so the header checksum must be recomputed at every hop, and it is kept cheap.

■ 26.3.5 TECHNICAL – the engineer’s version

1. The IPv4 header is defined in RFC 791, Internet Protocol, September 1981, and its layout has not changed since.

Field	Bits	Purpose
Version	4	Always 4
IHL	4	Header length in 32-bit words
DSCP	6	Class-of-service marking
ECN	2	Congestion experienced signal
Total Length	16	Header plus payload, in bytes
Identification	16	Groups fragments of one datagram
Flags	3	Reserved, DF, MF
Fragment Offset	13	Position, in 8-byte units
TTL	8	Hop budget
Protocol	8	Next header type
Header Checksum	16	One’s complement, header only
Source Address	32	Sender IPv4 address
Destination Address	32	Target IPv4 address
Options	0 to 320	Rare, usually absent

2. IHL is measured in 32 bit words, so the minimum legal value is 5, meaning 20 bytes, and the maximum is 15, meaning 60 bytes.
3. The 8 bits after version and IHL were originally a Type of Service byte. RFC 2474, December 1998, redefined the top 6 bits as the Differentiated Services Code Point, and RFC 3168, September 2001, took the bottom 2 for Explicit Congestion Notification.
4. Total Length is 16 bits, so an IPv4 datagram cannot exceed 65,535 bytes.
5. RFC 6864, February 2013, updated the Identification field: it is only meaningful when a datagram is actually fragmented, and senders that always set DF may leave it as anything, including zero.
6. Common Protocol numbers, assigned by IANA: 1 is ICMP, 6 is TCP, 17 is UDP, 41 is IPv6 encapsulation, 47 is GRE, 50 is ESP, 89 is OSPF, 132 is SCTP.
7. RFC 1812, Requirements for IP Version 4 Routers, June 1995, is the document that tells a router exactly how to treat each of these fields.
8. Tools: `tcpdump -vnv -x host 20.207.73.82` prints the decoded fields and the raw hex; in Wireshark the display filter `ip.ttl < 5` finds nearly expired packets, and `ip.flags.df == 1` finds Do Not Fragment packets.

■ 26.3.6 WORDS – remember these

1. TTL — a countdown that stops packets circling forever — time to live, an 8 bit hop budget decremented by every forwarding router.
2. DF bit — do not cut this packet up — the Don’t Fragment flag, which forces a router to drop rather than fragment an oversized packet.
3. Protocol number — a label saying what is inside — an 8 bit IANA-assigned value selecting the next-layer protocol.
4. Header checksum — a small sum that catches corruption — a 16 bit one’s complement checksum over the IPv4 header only.
5. DSCP — a priority marking — the 6 bit Differentiated Services Code Point from RFC 2474.

26.4 Fragmentation and MTU

■ 26.4.1 PLAIN – in simple words

1. Every link has a limit on how big a single packet may be.
2. That limit is called the **MTU**, short for maximum transmission unit.
3. A packet travels over many links in a row, and they need not all have the same limit.
4. So a packet can start out legal, travel happily for six hops, and then arrive at a link where it is too big.
5. There are only three things a router can do with a packet that is too big.
6. It can cut the packet into smaller pieces and send them separately.
7. It can throw the packet away and send a complaint back to the sender saying “too big, the limit here is this number”.
8. Or it can throw the packet away and say nothing at all.

■ 26.4.2 PLAIN – a picture in your head

1. Imagine driving a lorry across a country with a series of low bridges.
2. Option one: the bridge keeper unloads your lorry into three smaller vans and sends them on. Slower, more paperwork, but the goods arrive.
3. Option two: the bridge keeper turns you back and hands you a note saying “maximum height here is two point five metres”. You repack and try again.
4. Option three: the bridge keeper waves you into a ditch and goes back to reading the newspaper. You are simply never seen again.
5. Where this comparison breaks: a lorry driver knows immediately that they hit a bridge. A packet sender has no way to know unless somebody tells it.

■ 26.4.3 PLAIN – a worked example

1. Take a 1,500 byte IPv4 packet: 20 bytes of header and 1,480 bytes of data.
2. It reaches a link with an MTU of 1,400 bytes, and the DF bit is not set, so the router is allowed to cut it up.
3. Every fragment except the last must be a multiple of 8 bytes, because the offset field counts in 8 byte units.
4. The biggest data chunk that fits in 1,400 bytes is 1,380, and the biggest multiple of 8 not above 1,380 is 1,376.

Fragment	Offset field	Data bytes	MF flag
1	0	1,376	1
2	172	104	0

5. Offset 172 means 172 times 8, which is 1,376 bytes into the original data. That is exactly where fragment 1 stopped.
6. 1,376 plus 104 is 1,480, which is the original data. Nothing is lost.

■ 26.4.4 PLAIN – what is really happening inside

1. IPv4 lets any router on the path fragment a packet. IPv6 does not.
2. IPv6 made a deliberate decision: routers never fragment. If a packet is too big, the router drops it and sends back a “packet too big” message.
3. Modern IPv4 senders behave the same way voluntarily. They set the DF bit on every packet, which forbids any router from cutting it up.

4. That means the sender must find out for itself how big it may go. The method is called **path MTU discovery**.
5. It works like this. Send a full-size packet with DF set. If a link on the way cannot take it, that router drops it and sends back an ICMP message saying “fragmentation needed” together with the MTU it can accept.
6. The sender reads that number, shrinks its packets to fit, and retransmits.
7. The whole scheme depends on one thing: the ICMP message must get back.
8. When that happens, the big packets vanish and no message comes back. The sender keeps retransmitting the same too-big packet forever.
9. This is called a **path MTU black hole**, and it produces a very particular symptom: small things work, big things hang.

■ 26.4.5 TECHNICAL - the engineer’s version

1. Path MTU discovery for IPv4 is RFC 1191, November 1990. For IPv6 it is RFC 8201, July 2017, which obsoleted RFC 1981.
2. The IPv4 signal is ICMP type 3, code 4, Destination Unreachable with Fragmentation Needed and DF set. The next-hop MTU is carried in the message.
3. The IPv6 signal is ICMPv6 type 2, Packet Too Big. IPv6 forbids router fragmentation entirely, by RFC 8200, so the sender must react or fail.
4. IPv6 sets a hard floor: every link must support at least 1,280 bytes. IPv4’s equivalent floor is 68 bytes to forward and 576 bytes to reassemble.

Link type	Typical MTU	Note
Ethernet	1,500	The default everywhere
PPPoE DSL	1,492	8 bytes of PPPoE header
IPsec tunnel	1,400 to 1,438	Depends on ciphers
WireGuard	1,420	Common default
IPv6 minimum	1,280	Floor set by RFC 8200
Jumbo frames	9,000	Data centre convention

5. Because ICMP filtering is so common, the IETF specified a method that does not need ICMP at all: Packetization Layer Path MTU Discovery, RFC 4821, March 2007, extended to datagram transports in RFC 8899, September 2020.
6. A quick manual probe on macOS, which sends a 1,472 byte payload plus 8 bytes of ICMP header plus 20 bytes of IP header, exactly 1,500 bytes:

```
ping -D -s 1472 20.207.73.82
ping -D -s 1400 20.207.73.82
ping -D -s 1200 20.207.73.82
```

7. If the 1,472 probe fails and the 1,200 probe succeeds, you have found an MTU restriction on the path. On Linux the flag is -M do instead of -D.
8. Tunnels are the usual cause, and the reader’s machine showed several utun interfaces, which on macOS are tunnel devices. A tunnel always reduces the usable MTU.
9. Now the honest part, because it matters. A path MTU black hole is a genuine and common cause of exactly the kind of silent failure the reader saw, and it belongs on the candidate list.
10. But the evidence here points elsewhere. curl printed Trying 20.207.73.82:443... and then nothing for 15 seconds. curl prints Connected to only after the TCP handshake completes, and it never did.

11. So the packet that got no answer was the SYN, which is 64 bytes. An MTU problem does not stop a 64 byte packet.
12. What is PROVEN: the first small packet of the connection received no reply of any kind. What is merely SUGGESTED: something on that path silently discarded it.
13. The classic MTU black hole signature is the opposite shape: handshake succeeds, small responses arrive, and the connection stalls on the first full-size packet. That is not what happened.

■ 26.4.6 WORDS - remember these

1. MTU — the biggest packet a link will carry — maximum transmission unit in bytes, 1,500 for standard Ethernet.
2. Path MTU — the smallest limit anywhere along the route — the minimum MTU over all links on a given path.
3. Fragmentation — cutting a packet into smaller ones — splitting an IPv4 datagram across multiple packets sharing an identification value.
4. Path MTU discovery — asking the network how big you may go — RFC 1191 for IPv4 and RFC 8201 for IPv6, using ICMP too-big messages.
5. Black hole — packets vanish and nobody says why — a path where oversized packets are dropped and the ICMP notification is filtered.

26.5 What a router actually does, mechanically

■ 26.5.1 PLAIN - in simple words

1. A router is a machine with two or more network connections that passes packets from one to another.
2. Step one: a frame arrives on a wire and the router checks it is not damaged.
3. Step two: it throws away the outer wrapper, the one with hardware addresses, because that wrapper was only ever for that one wire.
4. Step three: it reads the destination IP address inside.
5. Step four: it looks that address up in its routing table to decide which way out to send it, and to whom.
6. Step five: it subtracts one from the hop counter. If the counter hits zero, the packet is destroyed and a complaint is sent back.
7. Step six: because it changed the hop counter, it fixes the small checksum.
8. Step seven: it finds the hardware address of the next machine along.
9. Step eight: it builds a brand new outer wrapper with that hardware address and pushes the packet out of the chosen wire.
10. The single most important thing to hold on to: the outer wrapper is destroyed and rebuilt at every hop, but the address inside stays the same.

■ 26.5.2 PLAIN - a picture in your head

1. Imagine a parcel travelling by a chain of couriers, each covering one leg.
2. The parcel has the final address written on it in permanent ink. Nobody ever changes that.
3. Each courier puts the parcel in their own bag with their own routing sticker: "from depot 4, to depot 9".
4. When it arrives at depot 9, that bag is thrown in the bin. The sticker was only ever about that one leg.
5. Depot 9 reads the permanent address, decides depot 14 is next, puts the parcel in a fresh bag with a fresh sticker, and sends it on.

6. Now the speed problem. Doing a full table lookup in software for every packet is far too slow for a serious router.
7. So real routers split in two. The **control plane** works out what the routes should be, slowly, in software.
8. The **data plane** forwards packets, fast, using a stripped-down copy of the answers built into dedicated hardware.

■ 26.5.5 TECHNICAL - the engineer's version

1. The full forwarding path for an IPv4 unicast packet, per RFC 1812:
 - 1) receive frame, verify FCS, check dst MAC is ours
 - 2) demultiplex on EtherType 0x0800 -> IPv4
 - 3) verify IHL, total length and header checksum
 - 4) if TTL <= 1: drop, send ICMP time exceeded
 - 5) longest prefix match on destination in the FIB
 - 6) decrement TTL, update header checksum incrementally
 - 7) resolve next-hop IP to MAC via ARP cache or request
 - 8) encapsulate in a new L2 header, enqueue on egress
2. Those eight steps are the forwarding plane. Nothing in the list consults any state about previous packets, which is what makes IP stateless.
3. The routing table is the Routing Information Base, the RIB. The hardware copy is the Forwarding Information Base, the FIB.
4. The RIB holds every route learned from every source, including losers. The FIB holds only the winner for each prefix, in a form the silicon can search.
5. FIB lookup in hardware is typically a TCAM, ternary content addressable memory, which compares a key against every stored prefix in one clock cycle.
6. TCAM is expensive and power hungry, which is why routers have hard limits on route counts. A switch sold as supporting 16,000 IPv4 routes cannot hold a full internet table.
7. As of the CIDR Report for 13 August 2026, the global IPv4 table held about 1,071,273 prefixes originated by 79,217 autonomous systems.
8. Address resolution is ARP for IPv4, RFC 826 from November 1982. IPv6 uses Neighbor Discovery instead, RFC 4861.
9. Tools: `arp -a` on macOS lists the resolved neighbours; `netstat -rn` shows the routing table; `ip route get 20.207.73.82` on Linux shows the exact decision for one destination.

■ 26.5.6 WORDS - remember these

1. Hop — one link between two routers — a single layer 3 forwarding step, which decrements TTL by one.
2. Next hop — who to hand the packet to next — the IP address of the adjacent router selected by the routing lookup.
3. Control plane — the part that decides routes — routing protocol software building the RIB.
4. Data plane — the part that moves packets — hardware forwarding using the FIB.
5. RIB — the full list of everything learned — Routing Information Base, holding all candidate routes.
6. FIB — the short list actually used — Forwarding Information Base, one best next hop per prefix, held in fast memory.

26.6 The routing table

■ 26.6.1 PLAIN – in simple words

1. A routing table is a list of rules of the form “packets going here should be sent that way”.
2. Each rule has a destination pattern, a way out, and sometimes a next machine to hand the packet to.
3. The destination pattern is not one address. It is a whole block of addresses, written as an address and a slash number.
4. 192.168.0.0/24 means “any address whose first 24 bits match”, which is the 256 addresses from 192.168.0.0 to 192.168.0.255.
5. Several rules can match the same address at once. That is normal and deliberate.
6. When several match, the winner is the one with the biggest slash number, the most specific one. That rule is called **longest prefix match**.
7. There is almost always one rule with a slash number of zero, written 0.0.0.0/0. Zero bits must match, so it matches everything.
8. That is the **default route**, meaning “if nothing else matched, send it here”. On the reader’s laptop it points at 192.168.0.1.

■ 26.6.2 PLAIN – a picture in your head

1. Think of the signs in a large airport.
2. One sign says “All departures, this way”. That is the default route.
3. Further on, a sign says “International departures, this way”. More specific, so it overrides the general one.
4. Further on again, “Flights to Mumbai, gate 34”. More specific still.
5. All four signs point at your flight. You obey the most specific one that applies to you, and you ignore the vaguer ones.
6. Where this comparison breaks: an airport arranges the signs so you meet the general one first and the specific one later. A router sees all its rules at once and picks the most specific in a single step.

■ 26.6.3 PLAIN – a worked example

1. Take a router with these five rules, and a packet for 20.207.73.82, which is the address the reader’s github.com resolved to.

Rule	Next hop	Matches?
0.0.0.0/0	192.168.0.1	yes
20.0.0.0/8	10.9.0.1	yes
20.192.0.0/10	10.9.0.5	yes
20.207.64.0/18	10.9.0.9	yes
20.207.73.0/24	10.9.0.13	yes

2. All five match. Now check the bits and see why.

```

20.207.73.82 = 00010100 11001111 01001001 01010010

/0    (nothing to compare)      match
/8    00010100                  match
/10   00010100 11                match
/18   00010100 11001111 01       match
/24   00010100 11001111 01001001 match

```

3. Take the third one slowly. 20.192.0.0/10 compares the first 10 bits.
4. The route's second byte is 192, which is 11000000. Its first two bits are 11.
5. The address's second byte is 207, which is 11001111. Its first two bits are also 11. So the first 10 bits agree.
6. Now a rule that does not match, for contrast: 20.207.128.0/18. The third byte 128 is 10000000, whose first two bits are 10, and the address's third byte 73 is 01001001, whose first two bits are 01. No match.
7. Winner: 20.207.73.0/24, because 24 is the largest number of matching bits. The packet goes to 10.9.0.13.
8. The default route loses every time anything else matches, which is exactly what "default" means.

■ 26.6.4 PLAIN – what is really happening inside

1. Routes get into the table in four different ways, and the difference matters.
2. **Connected routes** appear automatically. Give an interface the address 192.168.0.14/24 and the machine knows 192.168.0.0/24 is reachable directly on that interface, with no next hop at all.
3. **Static routes** are typed in by a person or set by a script. They stay until someone removes them.
4. **Dynamic routes** are learned from a routing protocol talking to other routers. Section 26.7 covers those.
5. When two sources offer a route to the very same prefix, longest prefix match cannot break the tie, because the prefixes are identical.
6. Two more tie-breakers exist, in this order.
7. First, **administrative distance**: a trust score per source. A lower number means "trust this source more". Connected beats static beats OSPF beats RIP.
8. Second, **metric**: within one protocol, the cheaper route wins. What "cheaper" means depends entirely on the protocol.

■ 26.6.5 TECHNICAL – the engineer's version

1. Here is a realistic netstat -rn from the reader's macOS machine. Only the IPv4 section is shown, because the machine reported IPv6: (none).

Routing tables

Internet:					
Destination	Gateway	Flags	Netif	Expire	
default	192.168.0.1	UGScg	en0		
127	127.0.0.1	UCS	lo0		
127.0.0.1	127.0.0.1	UH	lo0		
169.254	link#12	UCS	en0	!	
192.168.0	link#12	UCS	en0	!	
192.168.0.1/32	link#12	UCS	en0	!	
192.168.0.1	ac:22:0b:4e:1f:2a	UHLWIir	en0	978	
192.168.0.14/32	link#12	UCS	en0	!	
192.168.0.255	ff:ff:ff:ff:ff:ff	UHLWbI	en0	!	
224.0.0/4	link#12	UmCS	en0	!	
255.255.255.255/32	link#12	UCS	en0	!	

2. Line by line:
 1. default 192.168.0.1 is 0.0.0.0/0 via the home router. Every packet for 20.207.73.82 used this line.
 2. 127 and 127.0.0.1 are loopback, so the machine can talk to itself without touching any wire.
 3. 169.254 is the link-local block from RFC 3927, used when DHCP fails.

4. 192.168.0 is the connected route for the home network, reachable directly on en0 with no gateway.
 5. 192.168.0.1/32 and 192.168.0.14/32 are host routes macOS creates for the gateway and for the machine's own address.
 6. 192.168.0.1 ac:22:0b:4e:1f:2a is the resolved ARP entry for the router, with 978 seconds left before it expires.
 7. 192.168.0.255 is the broadcast address for the subnet.
 8. 224.0.0/4 is the multicast block.
 9. 255.255.255.255/32 is the all-hosts broadcast address.
3. The Flags column is a set of letters, documented in the macOS netstat manual page.

Flag	Name	Meaning
U	RTF_UP	Route is usable
G	RTF_GATEWAY	Send via an intermediary
S	RTF_STATIC	Added manually
H	RTF_HOST	A single host, not a net
L	RTF_LLINFO	Has link-address info
W	RTF_WASCLONED	Generated by cloning
I	RTF_IFSCOPE	Scoped to one interface
i	RTF_IFREF	Holds an interface reference
r	RTF_ROUTER	Peer is a default router
C	RTF_CLONING	Spawns routes when used
c	RTF_PRCLONING	Protocol-specified cloning
m	RTF_MULTICAST	A multicast address
b	RTF_BROADCAST	A broadcast address
g	RTF_GLOBAL	Destination is on the global internet

4. So UGScg on the default route reads: usable, via a gateway, static, protocol-cloning, global. The ! on some lines marks a reject route for addresses inside the block with no more specific entry.
5. Administrative distance is a Cisco convention, not an internet standard. Other vendors use the term route preference with different numbers. Juniper's defaults, for instance, differ from the list below.

Source	Cisco distance	Notes
Connected	0	Trusted absolutely
Static	1	Typed by a human
eBGP	20	From another AS
EIGRP internal	90	Cisco only
OSPF	110	Link state
IS-IS	115	Link state
RIP	120	Distance vector
iBGP	200	Inside one AS

6. Tools: netstat -rn on macOS and BSD, ip route show on Linux, route print on Windows, show ip route on Cisco IOS, and route get 20.207.73.82 on macOS to see exactly which line wins.

■ 26.6.6 WORDS – remember these

1. Prefix — a block of addresses written with a slash — an address plus a mask length, such as 20.192.0.0/10.
2. Longest prefix match — the most specific sign wins — selecting the matching route with the greatest mask length.
3. Default route — where to send anything unrecognized — 0.0.0.0/0, matching every destination with zero bits of agreement required.
4. Connected route — the network you are plugged into — a route derived automatically from an interface address and mask.
5. Static route — a line typed by hand — an administratively configured route with no protocol maintaining it.

26.7 How routers learn routes

■ 26.7.1 PLAIN – in simple words

1. A small network can have its routes typed in by hand. A large one cannot.
2. So routers talk to each other and tell each other what they can reach.
3. There are two families of ways to do this, and they think very differently.
4. The first family is **distance vector**. Each router tells its neighbours a list: “I can reach these places, and here is how far away each one is”.
5. A router receiving that list adds the cost of the link it came over and compares the result with what it already knew.
6. The second family is **link state**. Every router describes only its own direct connections, and that description is flooded to every other router.
7. Every router therefore ends up with an identical picture of the whole network, and computes the best paths itself.
8. Distance vector is “ask a neighbour for directions”. Link state is “everyone gets the same map and works it out”.

■ 26.7.2 PLAIN – a picture in your head

1. Imagine a village with no maps, where you find your way by asking people.
2. Distance vector is asking your neighbour, “how far is the market?” They say “twelve minutes that way”. You add the two minutes to reach them and tell the next person “fourteen minutes”.
3. This works, but nobody can check anything. If your neighbour is wrong, everyone downstream of you is wrong too, and confidently so.
4. Worse: if the market closes, your neighbour hears from you that the market is fourteen minutes away, and tells you it is sixteen, and you tell them eighteen. The number climbs forever while the market stays shut.
5. Link state is different. Every person writes down only what they can see from their own doorstep, and copies of every note are posted to everybody.
6. Now everyone holds the same stack of notes, draws the same map, and works out their own shortest path.
7. Where this comparison breaks: real people would not re-post the entire note stack every thirty minutes, but link state protocols do refresh, and the flooding costs real bandwidth on very large networks.

■ 26.7.3 PLAIN – a worked example

1. Four routers, A, B, C and D, in a line, with A connected to a network N.

| N --- A --- B --- C --- D

2. With RIP, which counts hops, the table for network N fills in over time. A says $N = 0$ throughout, because N is connected to it.

Round	B says	C says	D says
0	unknown	unknown	unknown
1	$N = 1$	unknown	unknown
2	$N = 1$	$N = 2$	unknown
3	$N = 1$	$N = 2$	$N = 3$

3. Each round is one update interval. RIP's default update interval is 30 seconds, so full knowledge of a four-router line takes about a minute and a half.
4. Now break the link between A and B. A knows immediately. B does not.
5. Without protection, B hears from C that N is reachable at distance 2, adds one, and announces N at distance 3. C then hears 3 and announces 4.
6. The count climbs: 3, 4, 5, and onward, one per update round, until it reaches 16, which RIP defines as unreachable. That takes many rounds.
7. This is the famous **count to infinity** problem, and it is why RIP declares 16 to mean unreachable rather than allowing large numbers.

■ 26.7.4 PLAIN – what is really happening inside

1. Link state protocols need a way to turn a map into best paths, and they all use the same method: **Dijkstra's algorithm**.
2. Here it is in plain words, with no mathematics.
3. Write down every router. Give yourself a cost of 0 and everyone else a cost of infinity. Mark nobody as finished.
4. Pick the unfinished router with the smallest cost. The first time round that is you.
5. Look at each of its neighbours. For each one, work out the cost of reaching it through the router you just picked.
6. If that total beats the neighbour's current cost, write down the better total and remember which router you came from.
7. Mark the picked router finished. It can never improve, because every other route to it would have to go through something more expensive.
8. Repeat: pick the smallest unfinished one, relax its neighbours, mark it finished.
9. Stop when everyone is finished. Following the "came from" notes backwards gives the shortest path to every destination.
10. The output is a tree rooted at you, called the shortest path tree, and the first step of each branch is the next hop you put in the routing table.
11. **Convergence** is the time from a change happening to every router having finished redoing this and agreeing again.

■ 26.7.5 TECHNICAL – the engineer's version

1. Interior gateway protocols run inside one administrative domain. Exterior gateway protocols run between domains. Sections 26.8 and 26.9 cover the exterior side.

Protocol	Family	Metric	Specification
RIPv1	Distance vector	Hop count	RFC 1058, 1988
RIPv2	Distance vector	Hop count	RFC 2453, 1998
OSPFv2	Link state	Cost	RFC 2328, 1998
IS-IS	Link state	Wide metric	ISO 10589, RFC 1195
EIGRP	Advanced DV	Composite	RFC 7868, 2016

- RIP runs over UDP port 520, updates every 30 seconds, times a route out after 180 seconds, and treats metric 16 as infinity. RIPv2 added subnet masks and uses multicast address 224.0.0.9.
- OSPFv2 was specified by John Moy. Version 1 appeared as RFC 1131 in 1989 and the current version 2 as RFC 2328 in April 1998. OSPFv3 for IPv6 is RFC 5340.
- OSPF runs directly over IP as protocol number 89. On a broadcast network it sends hellos every 10 seconds and declares a neighbour dead after 40 seconds.
- OSPF divides a large network into areas, all joined to area 0, the backbone. Link state flooding is confined within an area, which bounds both the database size and the computation.
- IS-IS comes from the OSI world, standardized as ISO/IEC 10589, and was adapted to carry IP by RFC 1195 in December 1990. Radia Perlman designed it at Digital Equipment Corporation for DECnet Phase V.
- IS-IS runs directly on the link layer rather than inside IP, which is one reason very large carriers favour it: it is harder to attack from an IP packet, and it carries IPv4 and IPv6 in one protocol without change.
- Dijkstra's algorithm was conceived by Edsger Dijkstra in 1956 and published in *Numerische Mathematik* in 1959. Its cost with a binary heap is on the order of $E \log V$, where E is links and V is routers.

Convergence stage	Typical time
Link failure detection	10 ms to 40 s
LSA flooding	10 to 100 ms
SPF computation	1 to 100 ms
FIB update	10 ms to seconds

- Tools: `show ip ospf neighbor` and `show isis adjacency` on router command lines, and `birdc` `show route` or `vttysh` for the open source stacks BIRD and FRRouting.

■ 26.7.6 WORDS – remember these

- Distance vector — ask a neighbour how far it is — a protocol where routers exchange destination and metric pairs, without topology knowledge.
- Link state — everyone gets the same map — a protocol where routers flood descriptions of their own links and each computes paths independently.
- Count to infinity — a distance that climbs while the destination is gone — a distance vector loop where metrics increment until reaching the infinity value.
- Split horizon — do not tell a neighbour what it told you — suppressing advertisement of a route out of the interface it was learned on.
- Dijkstra's algorithm — repeatedly finish the cheapest unfinished node — a shortest path first algorithm, published 1959, used by OSPF and IS-IS.
- Convergence — everyone agreeing again after a change — the interval from a topology change to a consistent forwarding state network-wide.

26.8 Autonomous systems

■ 26.8.1 PLAIN – in simple words

1. The internet is not one network. It is about eighty thousand separate networks that have agreed to carry each other's traffic.
2. Each of those separate networks is called an **autonomous system**, usually shortened to AS.
3. Autonomous means it decides its own internal routing for itself. Nobody outside gets a say in how it moves packets inside its own borders.
4. Each AS has a number. Microsoft's main one is AS8075. That number is how networks refer to each other.
5. The reader's packets started in the ISP's autonomous system and ended up in Microsoft's, which is why the names at hop 7 change to `ntwk.msn.net`.
6. Networks connect to each other in two very different commercial ways.
7. **Transit** is buying. You pay a bigger network to carry your traffic to the whole internet and to bring the whole internet back to you.
8. **Peering** is swapping. Two networks connect directly and exchange traffic between their own customers, usually with no money changing hands.

■ 26.8.2 PLAIN – a picture in your head

1. Think of the world's postal services.
2. Each country runs its own post inside its borders however it likes. Different vans, different sorting rules, different rates. That is autonomy.
3. For letters leaving the country, two arrangements exist.
4. A small country might pay a big neighbour to handle everything international. That is transit. One bill, global reach.
5. Two countries that exchange a lot of post might set up a direct exchange at the border and simply swap sacks, no invoice. That is peering.
6. A border exchange building where twenty countries all bring sacks and swap with whoever they like is an internet exchange point.
7. Where this comparison breaks: postal countries have fixed geography and cannot choose their neighbours. Networks connect to whoever they wish, anywhere, so an AS in India can peer directly with one in Germany.

■ 26.8.3 PLAIN – a worked example

1. Take the address at the end of the reader's story, `20.207.73.82`.
2. Any of these three commands tells you which AS owns it.

```
whois -h whois.cymru.com " -v 20.207.73.82"
dig +short 82.73.207.20.origin.asn.cymru.com TXT
whois 20.207.73.82
```

3. The second one reverses the four octets and asks a special DNS zone. It is the fastest, because it is just a DNS lookup.
4. The answer for this address, checked in August 2026: AS8075, Microsoft Corporation, with the covering route `20.192.0.0/10`, registered in the United States, and the address itself geolocating to Pune, India.
5. That last detail lines up neatly with the trace. Hops 10, 11 and 12 carry `pnq20` and `pnq21` in their names, and `pnq` is the airport code for Pune.

- AS8075 is registered under the name MICROSOFT-CORP-MSN-AS-BLOCK, which is also why the hop names use `msn.net`.

■ 26.8.4 PLAIN – what is really happening inside

- Networks are informally sorted into tiers, and the definition is about whether you pay anyone.
- A **tier 1** network can reach every destination on the internet without buying transit from anybody. It peers with all the other tier 1 networks.
- A **tier 2** network peers with many networks but still buys transit from at least one provider to reach everything.
- A **tier 3** network buys transit and does little or no peering. Most small ISPs are here.
- Experts disagree about who counts as tier 1, because the list depends on private contracts nobody publishes. Treat any published list as an informed estimate.
- An **internet exchange point**, or IXP, is a neutral room with a large ethernet switch in it.
- Every member plugs one port into that switch. Once connected, a member can exchange traffic with any other member that agrees to peer.
- The exchange does not route anything. It provides a shared layer 2 fabric. The peering itself is BGP sessions between members over that fabric.
- Most exchanges also run a **route server**, so a new member can get routes from dozens of others by setting up one session instead of dozens.

■ 26.8.5 TECHNICAL – the engineer’s version

- AS numbers were originally 16 bit, giving 0 to 65,535. RFC 6793, December 2012, extended them to 32 bit, giving 0 to 4,294,967,295.

Range	Purpose	Reference
0	Reserved	RFC 7607
1 to 64495	Public, 16 bit	IANA to RIRs
64496 to 64511	Documentation	RFC 5398
64512 to 65534	Private use	RFC 6996
65535	Reserved	IANA
65536 to 65551	Documentation	RFC 5398
65552 to 4199999999	Public, 32 bit	IANA to RIRs
4200000000 to 4294967294	Private use	RFC 6996

- AS23456 is reserved as AS_TRANS, a placeholder used when a 32 bit AS number must be shown to a router that only understands 16 bit numbers.
- IANA allocates blocks of AS numbers to the five regional internet registries, ARIN, RIPE NCC, APNIC, LACNIC and AFRINIC, which then assign them.
- Real AS numbers worth knowing:

AS number	Network	Role
8075	Microsoft	Owns 20.207.73.82
13335	Cloudflare	Runs the 1.1.1.1 resolver
15169	Google	Content and cloud
16509	Amazon	Most IPv4 prefixes: 15,814
32934	Meta, formerly Facebook	Subject of 26.9
3356	Lumen, formerly Level 3	Large transit

AS number	Network	Role
174	Cogent	Large transit
1299	Arelion, formerly Telia	Large transit
2914	NTT	Large transit
6453	Tata Communications	Large transit, India
3491	PCCW Global	Large transit
17557	Pakistan Telecom	Subject of 26.9

- The reader's resolver, 1.1.1.1, is announced by AS13335. The name the resolver returned, 20.207.73.82, sits inside AS8075. Two entirely separate networks were involved before a single packet went anywhere.
- Real internet exchange figures, all checkable:

Exchange	Figure	Date
DE-CIX, all sites	25 Tbit/s peak	8 April 2025
DE-CIX Frankfurt	Launched 1995	1995
DE-CIX Frankfurt	About 1,100 networks	April 2025
AMS-IX Amsterdam	15 Tbit/s peak	15 April 2026
India, all IXPs	31 active, 934 members	August 2026

- Tools: `whois -h whois.cymru.com`, the PeeringDB directory for who peers where, `bgp.he.net` and `bgp.tools` for AS relationships, and the CIDR Report for daily table statistics.

■ 26.8.6 WORDS - remember these

- Autonomous system — one network under one routing policy — a set of prefixes with a single clearly defined routing policy, identified by an AS number.
- ASN — the number that names a network — a 16 or 32 bit autonomous system number allocated by IANA through a regional registry.
- Transit — paying someone for global reach — a paid service where a provider announces your prefixes to the whole internet and carries your traffic.
- Peering — swapping traffic directly — an interconnection carrying only the two parties' own customer routes.
- Settlement-free — peering with no invoice — peering where neither side pays the other, the normal arrangement between similar-sized networks.
- IXP — a shared room where networks plug in and swap — an internet exchange point providing a neutral layer 2 fabric for bilateral BGP peering.

26.9 BGP, the protocol that glues the internet together

■ 26.9.1 PLAIN - in simple words

- Inside one network, routers use protocols like OSPF to find the best path.
- Between networks, none of that works, because the two sides do not trust each other, do not share a map, and have contracts to honour.
- So there is a second protocol just for the boundaries between autonomous systems. It is called the **Border Gateway Protocol**, or BGP.
- BGP is how one network announces to another: "I can reach these address blocks, and here is the list of networks a packet would cross to get there".

5. That list of networks is called the **AS path**, and it is the heart of the protocol.
6. There are only two things BGP ever says. “I can reach this block”, which is an announcement, and “I can no longer reach this block”, which is a withdrawal.
7. Here is the part that surprises engineers: BGP does not pick the fastest path. It picks the path its operator’s policy prefers.
8. Policy usually means money. A route through a network you do not pay is preferred over a route through a network you do pay, even if it is longer.
9. And here is the uncomfortable part: in the original design, BGP has no way to check whether an announcement is true.

■ 26.9.2 PLAIN – a picture in your head

1. Imagine a shipping company deciding how to send goods to Chennai.
2. Three freight brokers phone in. Each says “I can get it there”, and each gives you a chain of firms it would pass through.
3. Broker A quotes a chain of two firms. Broker B quotes three. Broker C quotes two, but one of them charges you per container while the others do not.
4. You do not pick the shortest chain. You pick the one that costs you least, then use chain length only to break ties.
5. Now the flaw. Nobody verifies that any broker can actually reach Chennai. Whoever claims the most direct-sounding route gets the goods.
6. Where this comparison breaks: a shipping company would notice quickly that its goods never arrived. On the internet, the sender often sees only silence, which is exactly the shape of the reader’s failed connection.

■ 26.9.3 PLAIN – a worked example

1. This is the YouTube hijack of Sunday 24 February 2008, one of the clearest real examples ever recorded, documented by the RIPE NCC.
2. YouTube, then AS36561, announced 208.65.152.0/22, a block containing its servers.
3. Pakistan’s government ordered access to YouTube blocked. Pakistan Telecom, AS17557, decided to do it by announcing a more specific route internally so its own users would be sent nowhere.
4. At 18:47 UTC, AS17557 began announcing 208.65.153.0/24. That is a /24 inside YouTube’s /22, so it is more specific.
5. The announcement escaped. Pakistan Telecom’s upstream provider, PCCW Global, AS3491, accepted it and passed it to the rest of the world.
6. Longest prefix match then did what it always does. A /24 beats a /22, so routers everywhere sent YouTube traffic to Pakistan Telecom.

Time UTC	Event
18:47	AS17557 announces 208.65.153.0/24
20:07	AS36561 announces the same /24 back
20:51	AS17557 path seen prepended
21:01	AS3491 withdraws AS17557 prefixes

7. Notice YouTube’s counter-move at 20:07. It announced the same /24 itself, so the two announcements were equally specific and the tie fell to AS path length. That recovered some traffic but not all.
8. It ended only when PCCW stopped accepting the announcement at 21:01.

■ 26.9.4 PLAIN – what is really happening inside

1. BGP is a **path vector** protocol. Each announcement carries the full list of autonomous systems it has crossed, not just a distance.
2. That list has two jobs. It breaks loops, because a router discards any announcement that already contains its own AS number. And it is a tie-break, because a shorter list is preferred.
3. Two kinds of BGP session exist and they behave differently.
4. **eBGP** runs between two different autonomous systems. Every time an announcement crosses an eBGP session, the sender adds its own AS number to the front of the path.
5. **iBGP** runs between routers inside one autonomous system. The AS path is not changed, because the announcement has not left the AS.
6. iBGP has a rule that catches everyone out: a router will not pass a route learned from one iBGP peer on to another iBGP peer. That prevents loops, but it means every iBGP router must talk to every other one, or you need route reflectors.
7. When several announcements offer the same prefix, BGP runs a fixed list of tie-breakers, in strict order, and stops at the first one that decides.
8. The first real tie-break is **local preference**, a number set by your own policy. It exists to encode “prefer the free path over the paid path”.
9. Only after local preference does AS path length matter. This is why the internet’s routes are frequently not the shortest ones available.

■ 26.9.5 TECHNICAL – the engineer’s version

1. BGP-4 is specified in RFC 4271, January 2006, which obsoleted RFC 1771 from 1995. The first version, RFC 1105, dates from June 1989 and was written by Kirk Lougheed of Cisco and Yakov Rekhter of IBM.
2. The best path selection order, as implemented by Cisco IOS. Steps 1 and parts of the ordering are vendor conventions, not parts of RFC 4271.

Step	Criterion	Prefer
1	Weight	Highest, Cisco only, local
2	Local preference	Highest
3	Locally originated	Self-originated
4	AS path length	Shortest
5	Origin type	IGP over EGP over incomplete
6	MED	Lowest
7	eBGP versus iBGP	eBGP
8	IGP metric to next hop	Lowest
9	Router ID	Lowest

3. Read that table as a statement about economics. The two strongest knobs, weight and local preference, are both purely local policy. Distance does not appear until step 4, and physical latency never appears at all.
4. Three incidents worth knowing exactly:
 1. 25 April 1997, the AS 7007 incident. A router at a small Florida provider re-announced large parts of the internet as /24s, and much of the global routing system followed the more specific routes into it.
 2. 24 February 2008, the Pakistan Telecom YouTube hijack described above.

3. 4 October 2021, the Facebook outage. Cloudflare recorded a burst of BGP updates from Facebook's network beginning around 15:39 UTC, including withdrawal of the routes covering all of its authoritative DNS servers.
5. The Facebook case is the cleanest illustration of withdrawal. Facebook's DNS servers were designed to withdraw their own routes if they lost contact with the data centres, and a backbone configuration change triggered exactly that.
6. Resolvers worldwide, including the 1.1.1.1 service the reader uses, then returned SERVFAIL for facebook.com, because the name servers were not merely down, they were unreachable at the routing layer.
7. BGP activity resumed shortly before 21:00 UTC, routing was restored at about 21:50 UTC, and services were generally usable again by about 22:45 UTC. Six to seven hours, from a routing change, not an attack.
8. A **route leak** is different from a hijack. The prefixes are genuine, but they are passed to a party that should never have received them, usually by a network announcing its providers' routes to its other providers.
9. On 16 April 2021, AS55410, Vodafone Idea in India, leaked more than 30,000 prefixes it did not own, drawing a large multiple of its normal inbound traffic and disrupting reachability well beyond India.
10. The current defence is the **Resource Public Key Infrastructure**, RPKI, architecture in RFC 6480, February 2012.
11. An address holder publishes a signed **Route Origin Authorization**, a ROA, saying "AS number X is authorized to originate this prefix, up to this length". 20.207.73.82 is covered by a valid ROA today.
12. Routers then perform **Route Origin Validation**, RFC 6811, January 2013, marking each route Valid, Invalid or NotFound, and operators normally drop the Invalid ones.
13. Deployment figures from the NIST RPKI Monitor for 1 August 2026, IPv4:

State	Prefix-origin pairs	Share
Valid	842,771	68.57 percent
NotFound	372,904	30.34 percent
Invalid	13,370	1.09 percent

14. Measured by address space rather than by prefix count, 61.64 percent of announced /24 equivalents were covered by a valid ROA on the same date.
15. Coverage is not the same as enforcement. Published assessments in 2026 put the share of autonomous systems actually enforcing ROV at roughly a quarter, and this is the number that matters.
16. RPKI origin validation only proves who may originate a prefix. It says nothing about whether the rest of the AS path is real, so it stops the 2008 style hijack but not every route leak.
17. Tools: birdc show protocols, RIPE RIS and RouteViews for historical BGP data, BGPlay for animated replays of incidents, and public looking glasses for a live view from another network.

■ 26.9.6 WORDS – remember these

1. BGP — how networks tell each other what they can reach — the Border Gateway Protocol, RFC 4271, running over TCP port 179.
2. Path vector — an announcement carrying its full route history — a protocol where each update lists the AS numbers traversed.

3. AS path — the list of networks a route crossed — the AS_PATH attribute, used for loop detection and as a tie-break.
4. Announcement — “I can reach this block” — a BGP UPDATE advertising NLRI with path attributes.
5. Withdrawal — “I can no longer reach this block” — a BGP UPDATE removing a previously advertised prefix.
6. Local preference — a policy dial for choosing paid versus free — the LOCAL_PREF attribute, compared before AS path length.
7. eBGP — a session between two networks — external BGP, which prepends the local AS number to the path.
8. iBGP — a session inside one network — internal BGP, which does not change the AS path and does not re-advertise between iBGP peers.
9. Hijack — announcing address space that is not yours — an unauthorized origin announcement, valid to every router that hears it.
10. Route leak — passing routes to somebody who should not get them — propagating routes in violation of the intended business relationship.

26.10 Why nobody knows the whole path in advance

■ 26.10.1 PLAIN – in simple words

1. There is no plan for a packet’s journey. Nowhere is one written down.
2. Each router knows only one thing about a destination: which neighbour to hand the packet to next.
3. It does not know how many hops remain, or which countries they are in, or whether the far end is even up.
4. This is called **hop-by-hop forwarding**, and it is the whole model.
5. The path exists only as the sum of many independent local decisions, made fresh for every packet.
6. So the path can change in the middle of a conversation, without warning, and nothing has to be renegotiated.
7. And the way back need not use the same routers as the way out. The two directions are decided by different networks with different policies.

■ 26.10.2 PLAIN – a picture in your head

1. Imagine posting a letter and asking every person you meet, “which way to Pune?”, then walking one street in the direction they point.
2. At the next corner you ask again. Nobody you meet knows the whole route. Each knows only the next turn.
3. You will get there, because each person is right about their own street.
4. If a street closes while you are walking, the next person simply points a different way. Nothing had to be cancelled.
5. Where this comparison breaks: you can see where you have been. A packet cannot. It carries no record of its journey unless a rare option is enabled.

■ 26.10.3 PLAIN – a worked example

1. The reader’s traceroute shows 12 hops outward before the replies stop.
2. Hop 1 is the router in the flat. Hops 2 to 6 are private ISP addresses. Hop 3 is the ISP’s public address 137.97.29.249. From hop 7 the names become ntwk.msn.net, so the packets are inside Microsoft’s network.

3. The names then read Delhi, then Mumbai, then Pune. That is a real geographic journey, visible in the site codes `del01`, `bom01`, `pnq20` and `pnq21`.
4. Now the honest limit. Every one of those 12 lines was learned from a reply sent back to the reader.
5. A reply is generated when a packet's TTL hits zero. The reply is a brand new packet, sent from that router to the reader, over whatever path that router chooses.
6. So each line tells you one address on the forward path, and nothing whatever about how the answer travelled home.
7. If the fault lies on the return path, a traceroute from the reader's laptop cannot see it at all. That is a limit of the model, not of the tool.

■ 26.10.4 PLAIN – what is really happening inside

1. Paths differing in the two directions is called **asymmetric routing**, and it is the normal case on the internet, not the exception.
2. It happens because forward and reverse decisions are made by different parties. Your ISP decides how to reach Microsoft. Microsoft decides how to reach your ISP.
3. Both apply their own local preference, so both may pick a cheaper path than the other side chose.
4. A stateful firewall that only sees one direction of a flow will drop it, because it never saw the handshake it expects.
5. To diagnose properly you need a trace from both ends, or a looking glass at the far network that can trace back towards you.

■ 26.10.5 TECHNICAL – the engineer's version

1. Traceroute was written by Van Jacobson in 1987 and released in 1988. It works by sending probes with TTL 1, 2, 3 and upward, and reading the ICMP time exceeded messages, ICMP type 11 code 0, from RFC 792.
2. Classic Unix traceroute uses UDP to high destination ports. macOS supports `-I` for ICMP echo probes and `-T` for TCP probes.
3. `traceroute -T -p 443 20.207.73.82` is the version worth running for the reader's fault, because it probes the exact port that failed. Firewalls frequently treat UDP, ICMP and TCP port 443 completely differently.
4. Silence at the end of a trace proves nothing on its own. Routers commonly rate limit ICMP generation, and many networks suppress it entirely. Hops 13 to 20 showing `* * *` is consistent with a healthy path.
5. Reverse path visibility requires cooperation: a public looking glass, a RIPE Atlas measurement from a probe near the target, or `mttr` run from both ends.
6. `mttr` is the better everyday tool, because it runs continuously and shows per-hop loss and jitter rather than a single snapshot.

■ 26.10.6 WORDS – remember these

1. Hop-by-hop forwarding — each router only knows the next step — destination based forwarding with no end-to-end path state.
2. Asymmetric routing — the way back is not the way out — forward and reverse paths differing because each direction is chosen by a different AS.
3. Time exceeded — the message a router sends when the counter runs out — ICMP type 11 code 0, the mechanism traceroute depends on.
4. Looking glass — a window into another network's view — a public interface for running traceroute or BGP queries from a remote AS.

5. ICMP rate limiting — a router answering only some probes — deliberate throttling of ICMP generation, which shows as false loss in traceroute.

26.11 Load balancing inside the network

■ 26.11.1 PLAIN – in simple words

1. Between two big routers there is rarely one cable. There are often four, eight or sixteen, run in parallel for capacity and safety.
2. When a router has several equally good ways to reach a destination, it uses all of them. This is called **equal cost multipath**, or ECMP.
3. Now the question: how does it decide which cable each packet takes?
4. The safe method is to keep each conversation on one cable. All packets of one connection follow the same link, in order.
5. The router works out which cable by mixing a few fields from the packet header into a number and taking the remainder. The same conversation always produces the same number.
6. This is why the reader's traceroute shows two or three different addresses at hops 5, 6, 8, 9 and 11.
7. Traceroute sends many probe packets, and probes are usually not all part of one conversation, so they scatter across the parallel links and come back with different router addresses.
8. That is not a fault. It is the visible fingerprint of a healthy, redundant network, and it is PROVEN evidence of parallel paths rather than of trouble.

■ 26.11.2 PLAIN – a picture in your head

1. Think of a toll plaza with eight lanes going the same way.
2. If you sent every car down whichever lane was shortest right now, cars from the same convoy would arrive out of order.
3. So the plaza uses a rule instead: take the number plate, add up the digits, and send the car to lane number remainder-of-that.
4. Every car from the same fleet, with related plates, ends up in the same lane, in order. Different fleets spread evenly across all eight.
5. Where this comparison breaks: a lane that closes forces a re-computation, and with a naive rule that reshuffles everybody, not just the affected cars. Real networks use hashing schemes designed to limit that damage.

■ 26.11.3 PLAIN – a worked example

1. The fields used are called the **five-tuple**: source address, destination address, protocol number, source port and destination port.
2. For the reader's connection those are 192.168.0.14, 20.207.73.82, protocol 6, some ephemeral source port, and destination port 443.
3. Suppose a router has four equal paths. It hashes the five-tuple to a number and takes that number modulo 4.
4. A second connection from the same laptop to the same server uses a different source port, so a different hash, so possibly a different path.

Flow	Differs by	Same path?
Same TCP connection	nothing	Yes
New connection, same host	source port	Not necessarily
Ping probes	protocol is ICMP	Different hash space

Flow	Differs by	Same path?
Different destination	dest address	Not necessarily

- Now look at the trace again. Hop 8 shows `be23.rwa02.bom01.ntwk.msn.net` and `po22.rwa04.bom01.ntwk.msn.net`. Two different Mumbai routers answered probes with the same TTL.
- That means the probes took two different equal-cost paths through Mumbai. Both are correct answers.

■ 26.11.4 PLAIN – what is really happening inside

- There are two ways to spread traffic, and the difference matters enormously.
- Per-flow** hashing keeps a conversation on one link. Packets arrive in order. Each individual conversation is limited to the speed of one link.
- Per-packet** spreading sends each packet down whichever link is free. It uses capacity perfectly and reorders packets badly.
- Reordering hurts TCP. Three out-of-order acknowledgements look exactly like a lost packet, so TCP slows down even though nothing was lost.
- That is why per-flow is the default nearly everywhere, and per-packet is used only in controlled environments.
- Bundling several physical links into one logical link, which is what `ae`, `be` and `po` refer to, uses the same hashing idea one layer lower down.

■ 26.11.5 TECHNICAL – the engineer's version

- ECMP is described in RFC 2991, Multipath Issues in Unicast and Multicast Next-Hop Selection, and RFC 2992, Analysis of an Equal-Cost Multi-Path Algorithm, both from November 2000.
- RFC 2992 analyses hash-threshold assignment, chosen specifically because it disturbs fewer existing flows when the number of paths changes than a naive modulo does.
- Typical hash inputs by layer:

Layer	Fields hashed
L2 bundle	Source and dest MAC
L3 only	Source and dest IP
L3 plus L4	The five-tuple
Tunnelled	Outer header, or entropy label

- Tunnels break flow spreading. If thousands of flows share one outer header, the hash sees one flow. MPLS entropy labels and the UDP source port in VXLAN and in QUIC exist to restore variety.
- Traceroute has a specific defence. The Paris traceroute technique keeps the flow identifier constant across probes so that all probes follow one path, which makes multipath fan-out disappear and reveals the true single path.
- On macOS and Linux, `traceroute -T -p 443` fixes the destination port and protocol, which alone removes much of the scatter.

■ 26.11.6 WORDS – remember these

1. ECMP — using several equally good paths at once — equal cost multipath forwarding across next hops with identical metrics.
2. Five-tuple — the five fields that identify a conversation — source IP, destination IP, protocol, source port and destination port.
3. Flow hashing — arithmetic that pins a conversation to one link — hashing selected header fields to choose among equal cost next hops.
4. Per-flow versus per-packet — keep a conversation together, or spread every packet — order-preserving versus capacity-maximizing load distribution.
5. Link aggregation — several cables acting as one — bundling physical links, called ae, be or po depending on the vendor.

26.12 Quality of service and the real world

■ 26.12.1 PLAIN – in simple words

1. When packets arrive at a router faster than they can leave, they wait in a queue. That queue is called a **buffer**.
2. A small buffer smooths out short bursts. That is useful.
3. A large buffer holds packets for a long time before sending them. That is harmful, and it is why a fast connection can feel slow.
4. The packet is not lost. It is delayed, sometimes by a whole second, which for a video call or a game is worse than losing it.
5. This problem has a name: **bufferbloat**.
6. When a buffer finally fills completely, the simplest routers throw away whatever arrives next. That is called **tail drop**.
7. Better routers drop or mark a few packets early, on purpose, before the queue gets long, to tell senders to slow down sooner.
8. That deliberate early dropping is called **active queue management**.

■ 26.12.2 PLAIN – a picture in your head

1. Think of a single checkout in a shop with a very long roped queue behind it.
2. Adding more rope does not make the checkout faster. It only lets more people stand in line, so everybody waits longer.
3. Somebody who wants one item now waits behind a trolley shopper's whole load.
4. The fix is not more rope. It is a shorter permitted queue, and a rule that turns people away early rather than letting the line grow without limit.
5. Where this comparison breaks: shoppers who are turned away go home. A sender whose packet is dropped simply slows down and tries again, which is exactly the message the drop was meant to send.

■ 26.12.3 PLAIN – a worked example

1. Take a home connection with a 10 Mbit/s upload and a 1 megabyte buffer in the modem, which is not unusual.
2. One megabyte is 8,000,000 bits. At 10,000,000 bits per second, a full buffer takes 0.8 seconds to drain.
3. So while a large upload is running, every other packet from that house waits up to 800 milliseconds behind it.

4. A video call needs round trips under about 150 milliseconds to feel natural. The call breaks even though the link is not full.

Uplink	Buffer	Worst added delay
10 Mbit/s	1 MB	800 ms
10 Mbit/s	64 KB	51 ms
100 Mbit/s	1 MB	80 ms

5. Reducing the buffer from 1 MB to 64 KB cuts the worst case from 800 milliseconds to 51, on the same link, with no extra bandwidth.

■ 26.12.4 PLAIN – what is really happening inside

1. Two different tools are often confused: shaping and policing.
2. **Shaping** delays traffic to make it fit a target rate. Packets queue and come out smoothly. Nothing is thrown away unless the queue overflows.
3. **Policing** simply discards anything above the target rate. There is no queue, so there is no added delay, but there is more loss.
4. Shaping is kinder to TCP. Policing is cheaper to implement and common on provider edges.
5. Marking is a third idea. The DSCP field in the IP header lets a sender label a packet as more or less urgent.
6. The honest version: your marking usually does not survive. Most networks rewrite or zero the DSCP field at their boundary, because they will not let an outsider decide their queueing.

■ 26.12.5 TECHNICAL – the engineer's version

1. Jim Gettys named bufferbloat in 2010 and wrote it up as Bufferbloat: Dark Buffers in the Internet in ACM Queue in 2011.
2. Kathleen Nichols and Van Jacobson published CoDel, Controlling Queue Delay, in ACM Queue in 2012. It became RFC 8289, and the flow-queueing variant fq_codel became RFC 8290, both in January 2018.
3. CoDel does not target a queue length. It targets a queue delay, with a default target of 5 milliseconds measured over a 100 millisecond interval.
4. fq_codel adds per-flow queues with round-robin service, so a bulk transfer cannot delay a DNS lookup or a video call sharing the same link.
5. Earlier active queue management used Random Early Detection, from the 1993 paper by Sally Floyd and Van Jacobson, which drops with rising probability as the average queue grows. RED needs tuning; CoDel does not.
6. ECN, RFC 3168, lets a router set a bit instead of dropping. The receiver echoes it, the sender slows down, and nothing is lost.

DSCP name	Value	Typical use
CS0, default	0	Everything unmarked
AF41	34	Interactive video
EF	46	Voice
CS6	48	Network control

7. Tools: `tc qdisc` on Linux shows and sets the queue discipline, and `fq_codel` is the default on most current distributions. The `flent` test suite measures bufferbloat directly.

- Whether providers should deprioritize particular traffic is a policy argument, not a technical one, and experts disagree. In India, the regulator TRAI barred discriminatory pricing of data services in February 2016 and net neutrality rules followed in 2018.

■ 26.12.6 WORDS – remember these

- Buffer — the waiting room for packets — memory holding packets queued for an outgoing interface.
- Bufferbloat — a fast link that feels slow — excessive latency caused by oversized unmanaged buffers.
- Tail drop — throw away whatever arrives when full — the default queue discipline, dropping arrivals once the buffer is full.
- AQM — dropping early on purpose — active queue management, signalling congestion before the buffer fills.
- fq_codel — many short queues, each kept short — the flow queue CoDel scheduler, RFC 8290, targeting 5 ms of queue delay.
- Shaping — delay traffic to fit a rate — queueing excess traffic for later transmission.

26.13 Reading a packet capture

■ 26.13.1 PLAIN – in simple words

- Everything in this chapter can be watched directly. You do not have to guess.
- A **packet capture** is a recording of the actual frames going in and out of a network card.
- The command line tool is tcpdump. The graphical tool is Wireshark. They read the same file format.
- There are two kinds of filter, and mixing them up wastes hours.
- A **capture filter** decides what gets recorded. It runs inside the kernel and what it rejects is gone forever.
- A **display filter** decides what you are shown from an existing recording. Nothing is lost, and you can change it as often as you like.

■ 26.13.2 PLAIN – a picture in your head

- A capture filter is the security guard at the door of a building, deciding who is allowed in at all.
- A display filter is you, later, searching the visitor book for one name.
- If the guard turned somebody away, no amount of searching the book will find them. The record simply does not exist.
- Where this comparison breaks: the guard is fast and free, and letting everybody in has a real cost. On a busy link, capturing everything can fill a disk in minutes and drop packets while doing so.

■ 26.13.3 PLAIN – a worked example

- Here is what the reader's failure looks like on the wire.

```
$ sudo tcpdump -ni en0 host 20.207.73.82
09:14:02.118 IP 192.168.0.14.51422 > 20.207.73.82.443:
    Flags [S], seq 1829384756, win 65535, length 0
09:14:03.121 IP 192.168.0.14.51422 > 20.207.73.82.443:
    Flags [S], seq 1829384756, win 65535, length 0
09:14:05.128 IP 192.168.0.14.51422 > 20.207.73.82.443:
    Flags [S], seq 1829384756, win 65535, length 0
09:14:09.142 IP 192.168.0.14.51422 > 20.207.73.82.443:
    Flags [S], seq 1829384756, win 65535, length 0
```


2. Read the four things that matter.
3. Every line is outbound. Nothing at all comes back. Not a SYN-ACK, not a reset, not an ICMP message.
4. The sequence number is identical every time, so these are retransmissions of one SYN, not four separate attempts.
5. The gaps are 1, 2 and 4 seconds. That is TCP's exponential backoff, which is the operating system trying harder and harder before giving up.
6. The source port 51422 stays the same, confirming one connection attempt.
7. That capture, on its own, PROVES the request left the laptop and no answer returned. It does not prove where the packet died.
8. Compare with a refused connection, which replies at once with Flags [R], a reset, usually within milliseconds. The reader saw nothing of the kind.

■ 26.13.4 PLAIN – what is really happening inside

1. Capturing works by putting the network card driver into a mode where it hands a copy of every frame to a special kernel facility.
2. That facility applies the capture filter, compiled into a tiny program, and copies only matching frames into a ring buffer for the tool to read.
3. Because the filter runs in the kernel, a narrow capture filter costs almost nothing even on a fast link.
4. Two cautions. On Wi-Fi you normally see only your own traffic unless the card is put into monitor mode. And on a switch you see only frames addressed to you, unless a port is configured to mirror others.
5. Encrypted payloads stay encrypted. You can see that a TLS handshake happened, its size and its timing, but not the HTTP request inside it.

■ 26.13.5 TECHNICAL – the engineer's version

1. tcpdump was written in 1988 at Lawrence Berkeley Laboratory by Van Jacobson, Craig Leres and Steven McCanne. The filtering engine underneath it, the BSD Packet Filter, was described by McCanne and Jacobson in 1993.
2. Wireshark began as Ethereal, released by Gerald Combs in 1998, and was renamed Wireshark in 2006 after a trademark problem.
3. Capture filters use BPF syntax. Display filters use Wireshark's own syntax. They are not interchangeable, and this trips up nearly everyone once.

```
sudo tcpdump -ni en0 -s0 -w gh.pcap \  
  'host 20.207.73.82 and tcp port 443' \  
tcpdump -r gh.pcap -vvn
```

4. Useful display filters in Wireshark for this fault: `ip.addr == 20.207.73.82, tcp.flags.syn == 1 && tcp.flags.ack == 0, tcp.analysis.retransmission, and icmp.`
5. `-s0` captures whole frames rather than truncating them. `-w` writes a pcap file. Analysis should always be done on a file, never on live scrolling text.
6. A checklist for diagnosing silence, in order:
 1. Is there an outbound SYN at all, and to the right address?
 2. Is anything at all coming back, including ICMP?
 3. Are the retransmissions identical, or is the stack trying new ports?
 4. Does a capture on another network, such as mobile data, differ?

7. Tools worth having beside tcpdump: mtr for continuous per-hop loss, dig for name resolution, curl -v --trace-time for per-stage timing, and tshark for scripted analysis of a saved capture.

■ 26.13.6 WORDS – remember these

1. Packet capture — a recording of real traffic — a stored copy of frames read from an interface, normally in pcap or pcapng format.
2. Capture filter — decides what is recorded — a BPF expression evaluated in the kernel before frames are copied to userspace.
3. Display filter — decides what is shown — a Wireshark expression applied to an already captured set of packets.
4. Promiscuous mode — the card keeps frames not addressed to it — an interface setting required to see other stations' traffic.
5. Retransmission — the same data sent again — a repeat of an unacknowledged segment, identifiable by an identical sequence number.

26.98 Common wrong ideas

1. Wrong: packets travel down a fixed path that is set up when the connection starts. Right: nothing is set up. Each router decides independently, per packet, and the path can change mid-conversation without anyone being told.
2. Wrong: the internet routes around damage automatically and instantly. Right: it routes around damage eventually, if an alternative exists and if policy allows it. Convergence takes seconds inside one network and can take minutes across BGP, during which traffic is lost.
3. Wrong: a traceroute shows the return path. Right: it shows one address per hop on the way out, learned from replies that came back by an unknown route. The return path is invisible from one end.
4. Wrong: more hops means slower. Right: hop count and latency are only loosely related. Twelve hops inside one well-run backbone can beat four hops over a congested link or a satellite.
5. Wrong: stars at the end of a traceroute prove a fault. Right: many routers rate limit or block ICMP replies. The reader's hops 13 to 20 showing * * * is consistent with a perfectly healthy path.
6. Wrong: seeing two or three addresses for one hop means something is broken. Right: it means parallel links are in use, which is normal and desirable.
7. Wrong: BGP picks the shortest path. Right: BGP picks the path its operator's policy prefers. Local preference, which is pure business policy, is compared before AS path length, and physical distance is never compared at all.
8. Wrong: my ISP can see which pages I loaded because it carries my packets. Right: it sees addresses, sizes and timing. With TLS it does not see the content, though it can often infer a great deal from the pattern.

26.99 Chapter summary in 20 lines

1. Data is split into packets for fairness, cheap error recovery, small buffers and the ability to interleave many conversations on one wire.
2. Circuit switching reserves a path and wastes capacity. Packet switching reserves nothing and accepts variable delay. The internet chose the second.
3. A packet is a header plus a payload, and the payload is normally another whole packet with its own header.

4. A 512 byte HTTPS request from the reader's laptop becomes a 604 byte Ethernet frame, which is 15.2 percent overhead.
5. The IPv4 header is 20 bytes and 13 fields, unchanged since RFC 791 in 1981.
6. TTL stops packets circling forever. Decrementing it forces the header checksum to be recomputed, which routers do incrementally.
7. Every link has an MTU. Oversized packets are fragmented, rejected with an ICMP message, or silently dropped, and the third case is the painful one.
8. IPv6 never fragments in the network. Modern IPv4 senders set the Do Not Fragment bit and discover the path MTU for themselves.
9. A router receives a frame, strips it, looks up the destination, decrements TTL, fixes the checksum, resolves the next hop and builds a brand new frame.
10. MAC addresses change at every hop. IP addresses do not, unless a device is performing NAT.
11. A routing table is matched by longest prefix. 20.207.73.0/24 beats 20.192.0.0/10 beats 0.0.0.0/0 for the address 20.207.73.82.
12. The reader's default route, 0.0.0.0/0 through 192.168.0.1, carried every one of these packets out of the flat.
13. Inside a network, routes are learned by RIP, OSPF, IS-IS or EIGRP. Link state protocols flood a map and run Dijkstra's algorithm on it.
14. Between networks, routes are learned by BGP, which carries a full AS path and chooses by policy, with local preference outranking path length.
15. 20.207.73.82 belongs to AS8075, Microsoft, inside the route 20.192.0.0/10, which is why hop 7 onwards reads ntwk.msn.net.
16. BGP has no built-in way to verify an announcement. RPKI adds signed origin authorizations, and about 69 percent of IPv4 prefix-origin pairs were valid in August 2026, but only about a quarter of networks enforce checking.
17. Nobody knows the whole path in advance. Each router knows only the next hop, and the return path is chosen by different networks entirely.
18. Multiple addresses at hops 5, 6, 8, 9 and 11 are equal cost multipath at work, selected by hashing the five-tuple, and are healthy rather than faulty.
19. Oversized buffers turn loss into delay and make fast links feel slow. Short managed queues, such as fq_codel, fix it without more bandwidth.
20. A capture showing four identical SYN's at 1, 2 and 4 second gaps with no reply of any kind PROVES the request left and nothing came back, and SUGGESTS a silent drop somewhere on that path.

Traceroute Decoded — Reading Your Own Path

27.0 What this chapter gives you

1. You will be able to explain why there is no “show me the route” command in the internet protocol, and why traceroute is a trick rather than a feature.
2. You will be able to describe the time-to-live field exactly: how many bits it has, what it counts, and what happens at each router.
3. You will be able to guess which operating system sent you a packet by looking at one number in it, and to say why that guess can be wrong.
4. You will be able to draw the full sequence of probes and replies that produces a traceroute, and say what each printed line really is.
5. You will be able to choose between the three main traceroute methods and say which one gets furthest through modern filtering, and why.
6. You will be able to explain why one hop shows several addresses, and point at the exact hops in the reader’s own trace where this happened.
7. You will be able to read a row of stars honestly, and to say out loud that stars at the end of a trace prove nothing on their own.
8. You will be able to read every hop of the reader’s real trace, including every router name, city code and interface name.
9. You will be able to say precisely what a traceroute cannot tell you, which is a longer and more important list than what it can.
10. You will be able to run a much better traceroute next time, with the exact flags, and to get a second opinion from the far end of the path.

27.1 Traceroute is a clever trick, not a feature

■ 27.1.1 PLAIN – in simple words

1. When you send data across the internet, it passes through many machines before it arrives.
2. Each of those machines is a **router**: a box whose only job is to take a packet in one door and push it out the right other door.
3. The internet protocol has no field that records the route. Packets do not carry a list of where they have been.
4. Routers do not report to you. A router does not know or care who you are. It looks at the destination, picks a door, and forgets you instantly.
5. Traceroute gets the answer anyway, by cheating.

6. It sends packets that are designed to fail. It sends them so they die at the first router, then the second, then the third.
7. Every time a packet dies, the router that killed it sends back a short complaint. That complaint has the router's own address on it.
8. Nobody meant for this to be possible. It is a side effect of two safety rules that were written for completely different reasons.

■ 27.1.2 PLAIN – a picture in your head

1. Imagine you want to know the route your letters take to a friend in another city, and the postal service will not tell you.
2. But there is a rule: if a letter has been handled too many times without being delivered, an office must destroy it and post you a note saying so.
3. The note is stamped with that office's own name and town, because the rule also says the office must identify itself.
4. So you write a letter and mark it "destroy after one handling". Back comes a note from your local sorting office. You now know step one.
5. Twenty letters later you have the whole route, and you never once asked anyone a question they were allowed to refuse.

Where this comparison breaks: a real postal route is stable, so twenty letters would follow the same path. On a network each of your twenty probes can legitimately take a different road. A traceroute is a sketch assembled from many separate journeys, not a photograph of one. That single fact explains most of the confusing things in this chapter.

■ 27.1.3 PLAIN – a worked example

1. Here is the reader's own trace, exactly as their machine printed it. We are going to spend most of this chapter on these twenty lines.

```
1 192.168.0.1
2 172.31.0.17
3 137.97.29.249
4 172.26.22.235
5 172.16.18.33 / 172.26.14.75 / 172.26.22.234
6 172.26.14.75 / 172.16.18.33
7 ae66-0.del01-96cbe-1b.ntwk.msn.net (104.44.196.187)
8 be23.rwa02.bom01.ntwk.msn.net (104.44.55.163)
  po22.rwa04.bom01.ntwk.msn.net (104.44.51.55)
9 be1010.owr02.bom01.ntwk.msn.net (104.44.53.119)
  be1013.owr01.bom01.ntwk.msn.net (104.44.53.47)
10 be5.ibr02.pnq21.ntwk.msn.net (104.44.31.62)
11 ae104-0.rwa03.pnq21.ntwk.msn.net (104.44.20.52)
  be1.ibr01.pnq20.ntwk.msn.net (104.44.16.51)
12 ae106-0.rwa04.pnq20.ntwk.msn.net (104.44.20.42)
13-20 * * * (no reply)
```

2. Twenty lines. Twelve of them answered. Eight of them are silence.
3. Every single line is a death notice. Something threw the reader's packet away and sent back a note about it.
4. The router names on hops 7 to 12 were not sent by the routers either. Those came from a separate name lookup done afterwards by the reader's own machine, on the addresses in brackets.
5. So even the names are a second-hand fact. We will come back to that, because it matters when a name is wrong or missing.

■ 27.1.4 PLAIN – what is really happening inside

1. There were two serious attempts to build a real “record the route” feature into the internet, and both failed for good reasons.
2. The first is called the **record route option**. It is part of the original internet protocol from 1981. A packet can carry a small list, and each router writes its address into the list.
3. It does not work in practice. The space for options in the packet header is only 40 bytes, which holds at most nine addresses. Real paths are longer.
4. The second attempt was a proper traceroute protocol, published in 1993. It would have used one packet instead of dozens. Almost nobody implemented it.
5. So the working method is the trick. It relies on two rules that every router already obeys for its own reasons.
6. Rule one: every packet carries a counter, and every router must reduce that counter by one. When it hits zero the packet is destroyed. This exists to stop packets circling forever when routing goes wrong.
7. Rule two: when a router destroys a packet for that reason, it must send a short error message back to whoever sent it. This exists so the sender can find out its packets are dying.
8. Traceroute exploits both. It sets the counter deliberately low, so the packet dies on purpose at a chosen distance, and the error message that comes back carries the address of the killer.

■ 27.1.5 TECHNICAL – the engineer’s version

1. The IPv4 header defined in RFC 791, September 1981, is 20 bytes without options. There is no field in it that accumulates a path.
2. The Record Route option is IP option number 7, defined in RFC 791. Options live in the space between byte 20 and byte 60 of the header, so the whole option area is at most 40 bytes.
3. Packets with IP options are process-switched or punted to the control plane on most modern hardware routers, rather than handled by the forwarding silicon. Many operators drop them outright as a denial-of-service defence.
4. RFC 1393, “Traceroute Using an IP Option”, by G. Malkin of Xylogics, January 1993, is an Experimental specification. It proposed a new IP option and a new ICMP message so that one packet could map a whole path.
5. The mechanism that actually works is emergent behaviour from two normative requirements. RFC 1812, “Requirements for IP Version 4 Routers”, edited by Fred Baker, June 1995, section 5.3.1, requires that a router “MUST decrement the TTL field of every datagram by exactly one”, and that when the result is zero the router MUST discard the datagram and MUST send an ICMP Time Exceeded message.
6. The complementary requirement is in RFC 792, “Internet Control Message Protocol”, by Jon Postel, September 1981, which defines that Time Exceeded message as type 11, code 0.
7. Traceroute itself was written by Van Jacobson in 1987, from a suggestion by Steve Deering. The manual page has said for decades: “Implemented by Van Jacobson from a suggestion by Steve Deering.” Guy Almes and Matt Mathis are credited with having had the same idea independently.
8. Van Jacobson worked in the Network Research Group at Lawrence Berkeley National Laboratory in California, which released traceroute along with tcpdump and other network tools.

Attempt	Year	Why it is not used
Record Route option	1981	Only 9 addresses fit
RFC 1393 traceroute	1993	Never deployed
TTL trick	1987	This is the one in use

Attempt	Year	Why it is not used

9. Observe it with traceroute on macOS, Linux and BSD, tracert on Windows, tracepath on Linux, and mtr everywhere.

■ 27.1.6 WORDS – remember these

1. Router — a box that passes packets along — a layer 3 forwarding device that makes a next-hop decision per packet from a routing table.
2. Traceroute — a tool that lists the machines on the way out — a utility that elicits ICMP Time Exceeded responses by manipulating the IP TTL field.
3. Probe — one test packet — a single datagram sent with a specific TTL and a specific identifier so its response can be matched to it.
4. Record Route option — an old idea for packets to log their own path — IP option 7 from RFC 791, limited to nine addresses by the 40-byte option area.
5. Emergent behaviour — a use nobody designed for — functionality arising from the interaction of unrelated normative requirements.

27.2 The time to live field, properly

■ 27.2.1 PLAIN – in simple words

1. Every packet on the internet carries a small number called the **time to live**, almost always shortened to TTL.
2. It is one byte. That means it can hold any whole number from 0 to 255.
3. The name is misleading. It sounds like a clock. It is not a clock.
4. It counts hops, not seconds. A hop is one router handling the packet.
5. Every router that forwards the packet subtracts one from it.
6. When the number reaches zero, the packet is thrown away. It goes no further, no matter how important it was.
7. This is a safety belt. Without it, a mistake in the routing tables could make a packet loop between two routers forever, and the network would slowly fill up with immortal garbage.
8. Different computers start the number at different values. That turns out to be a small accidental fingerprint.

■ 27.2.2 PLAIN – a picture in your head

1. Think of a bus ticket with a row of boxes on it, and a number printed on the front saying how many boxes are left.
2. Every time you board a bus, the driver punches a box and reduces the number by one.
3. When the number reaches zero the driver refuses you and puts you off the bus, wherever you happen to be.
4. The ticket does not expire at a time of day. It expires after a count of journeys. A slow bus and a fast bus cost exactly the same: one punch.
5. Now imagine the rule says the driver who puts you off must post a postcard to whoever bought your ticket, saying which stop it happened at.

Where this comparison breaks: a router that holds a packet for a whole second still only subtracts one. The 1981 specification really did intend a clock, in seconds, and asked routers to subtract at

least one per second of holding time. No router implemented the clock part, and in 1995 the rule was rewritten to say plainly: subtract exactly one, every time. The name is a fossil.

■ 27.2.3 PLAIN – a worked example

1. The reader's machine is a Mac. macOS starts outgoing packets with TTL 64.
2. Follow one packet from the reader's flat toward hop 12 of their own trace.

Packet leaves the Mac with TTL = 64

```

at 192.168.0.1      64 -> 63  forward
at 172.31.0.17     63 -> 62  forward
at 137.97.29.249   62 -> 61  forward
at 172.26.22.235   61 -> 60  forward
at hop 5 router    60 -> 59  forward
at hop 6 router    59 -> 58  forward
at 104.44.196.187  58 -> 57  forward (Delhi)
at 104.44.55.163   57 -> 56  forward (Mumbai)
at 104.44.53.119   56 -> 55  forward (Mumbai)
at 104.44.31.62    55 -> 54  forward (Pune)
at 104.44.20.52    54 -> 53  forward (Pune)
at 104.44.20.42    53 -> 52  forward (Pune)

```

3. Now run the trick backwards. Suppose a reply comes back to you with a TTL of 52 in it. You guess it started at 64 and crossed 12 routers.

Received TTL	Likely start	Hops crossed
52	64	12
118	128	10
245	255	10
63	64	1

4. From that you can often guess what kind of machine sent it. A reply arriving with 118 was probably sent by Windows. One arriving with 52 was probably sent by something built on Unix, such as Linux or macOS.
5. This is a guess, not a measurement. Say so out loud when you use it.

■ 27.2.4 PLAIN – what is really happening inside

1. The TTL sits at a fixed place in the packet header, so a router can find it without understanding anything else about the packet.
2. On arrival the router reads it. If it is already 0 or 1, the packet cannot be forwarded, because forwarding would require subtracting one and reaching zero or below.
3. In that case the router drops the packet and builds an error message to send back to the source address printed on the dead packet. Otherwise it subtracts one and forwards.
4. There is a hidden cost here. The header carries a checksum, a small number that proves the header was not corrupted in transit.
5. Change the TTL and the checksum is now wrong, so the router must fix the checksum too, on every packet, at full line rate.
6. Note carefully who does the subtracting. Only devices that **forward** the packet. A plain network switch does not touch the TTL, because it works at a lower level and is not a router. This is why a large office network with many switches still shows as one hop.

■ 27.2.5 TECHNICAL – the engineer’s version

1. In IPv4 the TTL is an 8-bit unsigned field at byte offset 8 of the header, the ninth byte, immediately before the protocol field. Range 0 to 255.
2. RFC 791, September 1981, described it in time units and required a decrement of at least one per hop and per second of queuing.
3. RFC 1812, June 1995, section 5.3.1 replaced that with the flat rule: a router **MUST** decrement by exactly one. The time semantics are dead.
4. In IPv6 the field was renamed to **Hop Limit**, still 8 bits, at byte offset 7 of the 40-byte fixed header. The rename admits what the field always was.
5. The IANA IP Parameters registry states: “The current recommended default time to live (TTL) for the Internet Protocol (IP) is 64”, citing RFC 791 and RFC 1122.
6. Linux exposes the default as the `sysctl net.ipv4.ip_default_ttl`, documented as “Default value of TTL field (Time To Live) for outgoing (but not forwarded) IP packets”, with a documented default of 64.
7. Common starting values, all of them product defaults rather than standards, and all of them changeable by an administrator:

System	Start TTL	Kind of fact
Linux	64	kernel default
macOS and FreeBSD	64	kernel default
Windows, modern	128	product default
Cisco IOS and IOS-XR	255	product default
IANA recommendation	64	RFC 791, RFC 1122

8. Guessing the sender from a received TTL is called passive OS fingerprinting and is implemented in tools such as `p0f` and in the `ttl` matcher of several firewalls. It is weak evidence, for four reasons.
9. First, a tunnel changes the count. The reader’s machine showed several `utun` interfaces, which are point-to-point tunnel devices used by VPN and similar software. Traffic through a tunnel gets a different hop count.
10. On macOS you can read the local default with `sysctl net.inet.ip.ttl`, and macOS `traceroute` uses that same value as the default maximum hop count, documented as “`net.inet.ip.ttl` hops (the same default used for TCP connections)”.

■ 27.2.6 WORDS – remember these

1. TTL — a hop counter carried in every packet — an 8-bit IPv4 header field decremented by exactly one at each forwarding router, per RFC 1812 5.3.1.
2. Hop — one router handling the packet — one layer 3 forwarding decision.
3. Hop Limit — the IPv6 name for TTL — the 8-bit field at offset 7 of the IPv6 fixed header, semantically identical.
4. Header checksum — a small proof the header is intact — the 16-bit ones complement checksum over the IPv4 header only, updated incrementally.
5. Passive OS fingerprinting — guessing the sender’s system from clues — the inference of the originating stack from TTL, window size and option order.

27.3 The mechanism, step by step

■ 27.3.1 PLAIN – in simple words

1. Traceroute does the same small thing over and over, with one number changed each time.
2. Step one. Send a packet toward the final destination, but set its hop counter to 1.
3. The first router receives it, subtracts one, gets zero, and must throw it away. It sends back an error message that says “your packet ran out of hops at my place”.
4. That error message has the router’s own address as its sender. That is how the reader learned that hop 1 is 192.168.0.1.
5. Step two. Send another packet, exactly the same, but with the hop counter set to 2. It survives the first router and dies at the second.
6. The second router complains, and now the reader knows hop 2 is 172.31.0.17. Repeat with 3, then 4, then 5, until you reach the destination or give up.
7. Three packets are sent for each hop by default, not one, so a slow or lost probe does not hide a router.
8. Every line you see on screen is therefore a different packet. The trace is not one journey observed. It is dozens of separate short journeys.

■ 27.3.2 PLAIN – a picture in your head

1. Picture a long corridor of doors, one behind the other, and you standing at the near end. You cannot see past the first door.
2. You have a supply of paper darts. Each dart has a number written on it.
3. The rule of the corridor is that each person who catches a dart writes one less on it and throws it to the next person, unless the number would go to zero.
4. If the number would go to zero, that person keeps the dart, writes their own name on a card, and walks the card back to you.
5. So you throw a dart marked 1 and the nearest person hands you a card. Throw one marked 2 and the second person does. Twenty darts give you the whole list in order.
6. You also time each card. You start a stopwatch when the dart leaves your hand and stop it when the card arrives.

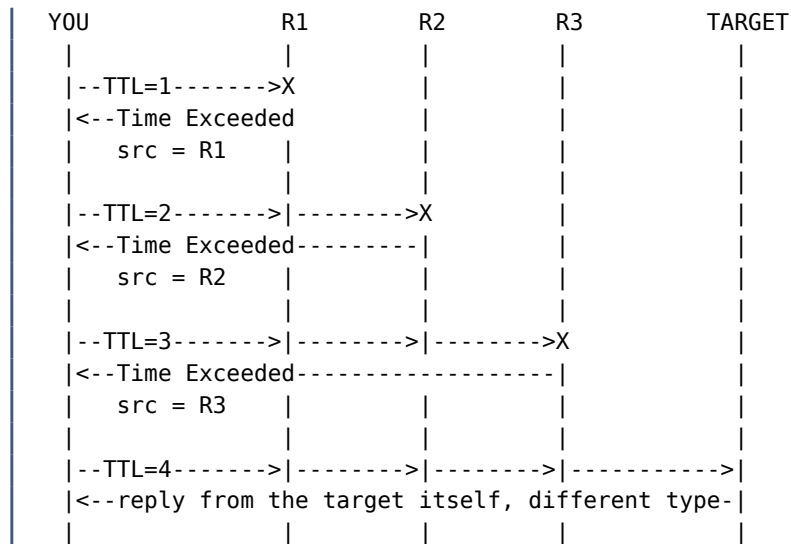
Where this comparison breaks: the person walking the card back may take a completely different corridor, which you never see, so the stopwatch measures a loop and not a distance. Also, some people are under orders never to walk cards back, and some are so busy they return only one card a second. Both turn into stars on your screen.

■ 27.3.3 PLAIN – a worked example

1. Here is the full sequence for the first three hops of the reader’s trace. Nine packets out, nine replies in.

Probe	TTL sent	Who replied
1, 2, 3	1	192.168.0.1
4, 5, 6	2	172.31.0.17
7, 8, 9	3	137.97.29.249

2. Notice that the destination address on all nine packets was the same: the GitHub address 20.207.73.82. Only the TTL changed.
3. Now the whole process as a diagram. X marks where a probe dies.



4. The last line is different on purpose. When a probe finally survives to the destination, the destination does not send a Time Exceeded message, because the packet did not run out of hops. It sends something else, and what it sends depends on which traceroute method you used.
5. Timing. Each of the three probes at each hop is timed on its own, which is why a normal traceroute prints three separate millisecond figures per line.
6. The reader's captured trace does not show those figures. It records only the addresses. We will come back to that gap in section 27.7, because it is a real limitation of the evidence they collected.

■ 27.3.4 PLAIN - what is really happening inside

1. The error message that comes back is not empty. It carries a copy of the dead packet inside it.
2. That copy is the key to the whole system. Without it, traceroute could not tell which of its many probes a given reply belongs to.
3. Replies arrive from different routers, in any order, some late, some never. Each reply must be matched to exactly one probe.
4. The trick is that the router quotes the dead packet back at you. Inside the error message you find the original header, including whatever identifying number traceroute put in it.
5. Classic traceroute puts that identifier in the destination port number. It uses a different port for every single probe.
6. So probe number 7 goes to port 33440, and when a reply comes back quoting port 33440, traceroute knows this is the answer to probe 7, sent at a known moment, with a known TTL.
7. This clever identifier scheme has one bad side effect, and it causes most of the confusion in the reader's trace. Changing the port for every probe changes what load-balancing routers see, so different probes get sent down different links. Section 27.5 is entirely about that.

■ 27.3.5 TECHNICAL - the engineer's version

1. The response of interest is ICMP type 11, code 0. RFC 792 gives the code meanings verbatim as "time to live exceeded in transit" for code 0 and "fragment reassembly time exceeded" for code 1.
2. In IPv6 the same event is ICMPv6 type 3, code 0, "Hop limit exceeded in transit", defined in RFC 4443, March 2006, by Conta, Deering and Gupta.
3. The ICMP error payload contains the IP header of the offending datagram plus at least the first 8 bytes of its payload, which is enough to capture the UDP or TCP source and destination ports, or the ICMP identifier and sequence number.

4. RFC 1812 section 4.3.2.3 upgraded that to say the error should contain as much of the original datagram as possible without exceeding a total ICMP message size of 576 bytes.
5. Source address selection matters for reading a trace. RFC 1812 section 4.3.2.4 requires the source address of a router-originated ICMP message to be an address of the interface over which the ICMP message is transmitted.
6. On a point-to-point link, that interface is the same one the probe arrived on. This is why traceroute conventionally shows the **ingress** interface address of each router, not a loopback address or a router identifier.
7. That single fact explains the interface names in the reader's hops 7 to 12. ae66-0 and be23 are names of the specific interfaces the probes came in on, not names of the routers.
8. The macOS manual page states the defaults as “-q nqueries: Set the number of probes per ttl to nqueries (default is three probes)”, “-p port: default is 33434”, and “-w: default 5 sec.”.

Element	Classic default	Where set
Probes per hop	3	-q
Base UDP port	33434	-p
Wait per probe	5 s	-w
Max hops	64 on macOS	-m

■ 27.3.6 WORDS – remember these

1. ICMP — the internet's error and status messaging — Internet Control Message Protocol, RFC 792, IP protocol number 1.
2. Time Exceeded — the “your packet ran out of hops” message — ICMP type 11 code 0 in IPv4, ICMPv6 type 3 code 0 in IPv6.
3. Quoted packet — the copy of the dead packet inside the error — the original IP header plus at least 8 payload bytes, used for demultiplexing.
4. Ingress interface — the door the packet came in through — the interface whose address a router normally uses as the ICMP error source.
5. Round-trip time — how long the there-and-back took — the measured interval between probe transmission and matching ICMP response reception, per probe.

27.4 The variants, and which one to use

■ 27.4.1 PLAIN – in simple words

1. The trick with the hop counter works no matter what kind of packet you send. The counter is in the outer wrapper, which every packet has.
2. So there are several traceroutes, differing only in what they put inside the wrapper.
3. The original sends a small packet to a strange, unused port number on the destination. This is the classic method and it is still the default on Mac and Linux.
4. The Windows version sends a ping instead. A ping is the simplest possible “are you there” message.
5. The third version sends the first packet of a real connection attempt, aimed at a real service port, such as the one used by secure websites.
6. That second difference is the whole point. Networks are much more willing to pass traffic that looks like normal web traffic than traffic that looks like a diagnostic tool.
7. For the reader's problem, which was a web connection failing, the third version is clearly the right one, and it is not the default.

■ 27.4.2 PLAIN – a picture in your head

1. Imagine three people trying to reach the same office in a guarded building.
2. The first wears a plain uniform nobody recognizes, and asks for room 33434, which does not exist. Guards find this suspicious and often turn them away.
3. The second shouts “hello, is anyone there?” from the doorway. Guards at many buildings have standing orders to ignore that shout entirely.
4. The third walks in wearing normal clothes and asks for the reception desk, room 443, which is the room everyone visits all day long.
5. The third person gets much further, because they look exactly like the thousands of legitimate visitors the guards are there to let in.

Where this comparison breaks: the guards are not usually deciding that they dislike diagnostics. Very often nobody blocked anything, and the difference is simply that the third kind of traffic is what the path is built and tuned to carry. Also, asking for reception has a real cost: you make the far end start work on a connection it will never finish.

■ 27.4.3 PLAIN – a worked example

1. The three commands, for the reader’s exact target, on macOS.

```
# 1. classic UDP, the default. No special permission needed.
traceroute 20.207.73.82

# 2. ICMP echo, the same method Windows tracert uses.
sudo traceroute -I 20.207.73.82

# 3. TCP to port 443, the one to use for a web problem.
sudo traceroute -P TCP -p 443 20.207.73.82
```

2. What each looks like at the moment it reaches the destination.

Method	Probe sent	Success reply
UDP classic	UDP to port 33434+	Port Unreachable
ICMP echo	Echo Request	Echo Reply
TCP to 443	TCP SYN to 443	SYN-ACK or RST

3. Notice that the classic method succeeds by provoking an error. It picks a port nothing is listening on precisely so the destination answers “nothing here”, which proves you reached it.
4. The TCP method succeeds by getting an actual answer from an actual service. If the destination replies at all, you have proven far more than a map.

■ 27.4.4 PLAIN – what is really happening inside

1. The important idea is that the intermediate hops behave identically in all three cases. Time Exceeded is generated for any packet whose counter runs out, regardless of what is inside.
2. But whether a probe reaches a middle hop at all depends on every device before it.
3. A firewall five hops along that drops odd UDP traffic will make everything past it look dead, even though the routers there are perfectly healthy.
4. Swap to TCP port 443 and the same probes sail through, and the hops appear.
5. There is one more family worth knowing: tools that repeat the whole trace continuously and build statistics, rather than running once and stopping.

- Running once gives you a snapshot. Running for two minutes tells you whether a hop loses one packet in a hundred, which a snapshot can never show.

■ 27.4.5 TECHNICAL – the engineer’s version

- UDP traceroute, the 1987 Van Jacobson design, sends UDP datagrams to destination ports starting at 33434 and increasing by one per probe, giving the familiar 33434 to 33534 window.
- Termination is by ICMP type 3 code 3, Destination Unreachable, Port Unreachable, from the target’s own stack.
- ICMP traceroute sends ICMP Echo Request, type 8 code 0, and terminates on Echo Reply, type 0 code 0. This is the default of Windows `tracert`, and is available on macOS as `-I`, documented as “Use ICMP ECHO instead of UDP datagrams. (A synonym for `-P icmp`)”.
- TCP traceroute sends a TCP SYN to a chosen port, typically 80 or 443, and terminates on SYN-ACK from an open port or RST from a closed one.
- The macOS manual page documents `-P proto` as “Send packets of specified IP protocol. The currently supported protocols are: UDP, TCP, GRE and ICMP”. Raw socket creation means `-I` and `-P TCP` need root.
- Why TCP to 443 gets further, in order of practical importance:

Reason	Effect on UDP or ICMP
Stateful firewall policy	High UDP ports dropped
ICMP policy on edges	Echo dropped or limited
Anycast and load balancers	Only serve real ports
QoS and CoPP classing	Diagnostics deprioritized

- `mttr`, originally written by Matt Kimball around 1996 and maintained by Roger Wolff from October 1998, combines traceroute and ping. It re-probes continuously and reports per-hop loss percentage, best, worst, average and standard deviation. Modern builds support ICMP, UDP and TCP probes.
- Paris traceroute was introduced in “Avoiding traceroute anomalies with Paris traceroute” by Brice Augustin, Xavier Cuvellier, Benjamin Orgogozo, Fabien Viger, Timur Friedman, Matthieu Latapy, Clemence Magnien and Renata Teixeira, at the Internet Measurement Conference in October 2006.
- What it fixes: classic traceroute varies the destination port per probe, which changes the flow identifier that load balancers hash on, so consecutive probes take different paths and the printed path is a blend of several real paths that may never have existed as one route.
- Paris traceroute holds the flow identifier constant across probes by varying a field the load balancer does not hash, classically the UDP checksum field, compensated so the checksum still validates.
- Recommendation for the reader’s exact situation, a failing HTTPS connection to `20.207.73.82`: use TCP to port 443. It is the only method whose probes are treated by every device on the path the same way the real failing traffic is treated. Anything else measures a different journey.

■ 27.4.6 WORDS – remember these

- Probe type — what kind of packet the tool sends — the transport and port combination that determines both filtering treatment and ECMP hashing.
- Port Unreachable — “nothing is listening there” — ICMP type 3 code 3, the normal termination signal of UDP traceroute.
- SYN — the first packet of a connection request — a TCP segment with the SYN flag set, opening the three-way handshake.

4. Paris traceroute — a traceroute that keeps probes looking identical — a variant holding the ECMP flow identifier constant across probes, from the 2006 Internet Measurement Conference paper.
5. MTR — a traceroute that never stops — a continuous hybrid of traceroute and ping reporting per-hop loss and latency statistics.

27.5 Why one hop shows several addresses

■ 27.5.1 PLAIN – in simple words

1. Look at hop 5 of the reader's trace. It has three addresses on one line.
2. Neither is true. It is normal, healthy, and a sign of a well-built network.
3. Remember that three separate packets were sent for that hop. Three packets, three journeys, three answers.
4. Big networks do not join two places with one cable. They join them with several cables side by side, for capacity and for safety.
5. When several equally good roads exist, a router shares traffic across them.
6. So probe one went left, probe two went straight, probe three went right. Each landed on a different router, and each of those routers answered.
7. It means the network has spare capacity in that place. If one of those cables was cut, traffic would keep flowing on the others.
8. Five of the reader's hops show this. Hops 5, 6, 8, 9 and 11. That is not a fault, and it is not a sign of anything wrong.

■ 27.5.2 PLAIN – a picture in your head

1. Imagine a wide river with three parallel bridges between the same two towns, and a traffic officer at the near end sending cars over all three.
2. You want to map the route. You send three cars, each told to stop at the far end of the first bridge and report which bridge they were on.
3. Now suppose the middle bridge is a little longer, with an extra toll booth halfway. Cars told to stop after the same number of booths stop in different places on different bridges.
4. That is why the reader's hop 11 shows two routers in two different cities. The parallel roads are not exactly the same length.

Where this comparison breaks: the officer chooses at random and a real router almost never does. It computes a number from the addresses and ports on each packet and uses that to pick a bridge, so two packets of one conversation always get the same bridge and cannot arrive out of order. Traceroute defeats that on purpose by changing the port on every probe.

■ 27.5.3 PLAIN – a worked example

1. Here are the reader's five multi-address hops, exactly as recorded.

Hop	Addresses seen	What differs
5	three private	three separate routers
6	two private	same two as hop 5
8	rwa02, rwa04	two Mumbai routers
9	owr01, owr02	two Mumbai routers
11	rwa03, ibr01	two cities, Pune

2. **Hop 5.** 172.16.18.33, 172.26.14.75 and 172.26.22.234. All three probes took different links. Three answers, three addresses.
3. **Hop 6.** 172.26.14.75 and 172.16.18.33. Both of these already appeared at hop 5.
4. That looks impossible until you remember the bridges. On one parallel road that router is five hops away; on another it is six.
5. **Hop 8.** Two Mumbai routers named rwa02 and rwa04. Different machines in the same city doing the same job.
6. **Hop 9.** Two Mumbai routers named ovr01 and ovr02. Same pattern, one step further in.
7. **Hop 11.** The most interesting one. One probe was answered by a router in Pune site 21, another by a router in Pune site 20.
8. So at the same hop count, two probes were in two different buildings. The two parallel paths through Pune are not the same length.

■ 27.5.4 PLAIN – what is really happening inside

1. The technique is called **equal-cost multi-path**, usually shortened to ECMP. It means: when the routing table offers two or more equally good next steps, use all of them.
2. The router must decide, per packet, which of them to use, and there are two ways to decide.
3. The first way is **per-packet**: alternate strictly, one packet each. This uses the links perfectly evenly and it is almost never used, because packets of one conversation arrive out of order and confuse the receiver.
4. The second way is **per-flow**: work out which conversation the packet belongs to, and always send that conversation the same way. This is what virtually every network does.
5. To identify the conversation, the router takes a handful of fields from the packet and mixes them into a single number. Then it takes the remainder of that number divided by the number of links.
6. The fields are usually the source address, the destination address, the protocol, the source port and the destination port. Five things, so this is called the five-tuple.
7. Now the punchline. Classic traceroute changes the destination port on every single probe, because it uses the port as a serial number. Change the port, change the mixed number, change the chosen link.
8. Your web browser does the opposite. One TCP connection keeps the same five fields for its whole life, so it stays on one link the entire time.
9. This is the single most important thing to understand about reading a traceroute: the tool is not following your traffic. It is sampling all the roads your traffic could have taken.

```

      +- -> R-a  172.16.18.33  ---+
      |
hop 4  -----+- -> R-b  172.26.14.75  ---+----> onward
      |
      +- -> R-c  172.26.22.234  ---+

probe 1 hashes to R-a, probe 2 to R-b, probe 3 to R-c
all three answer, so hop 5 prints three addresses

```

■ 27.5.5 TECHNICAL – the engineer's version

1. ECMP is standard behaviour in every modern routing platform. Inside a single operator, it is usually driven by an interior gateway protocol such as OSPF or IS-IS installing multiple equal-metric next hops.
2. The hash input is configurable. Typical defaults include the IPv4 source and destination addresses, the protocol number, and the layer 4 source and destination ports. Some platforms add the in-

coming interface index or a per-device seed so that two routers in series do not make correlated choices, which is called hash polarization.

3. Load balancing also happens below layer 3, on link aggregation groups. A bundle of physical links presented as one logical interface hashes frames across its members, often on MAC and IP fields.
4. That distinction matters for reading names. An interface called ae66-0 or be23 is a bundle. Traffic is already being spread across the physical members of that bundle, invisibly, without any change to the hop count.
5. So the reader's trace shows two layers of load balancing at once: ECMP across distinct routers, which is visible as extra addresses, and link bundling inside each hop, which is completely invisible.
6. Classic traceroute increments the UDP destination port per probe, mutating the five-tuple and therefore the hash, which is the anomaly Paris traceroute was written to eliminate.
7. Paris traceroute keeps the five-tuple constant and varies the UDP checksum field instead, adjusting payload bytes so the checksum remains valid. Routers do not hash on the checksum, so all probes follow one path.
8. The reader's hop 11 is the classic textbook case of the anomaly Paris traceroute describes. Reading hops 10, 11 and 12 as a single chain gives a route that probably never existed as one route.

Reader's hop	Route A	Route B
10	ibr02.pnq2l	ibr02.pnq2l
11	rwa03.pnq2l	ibr01.pnq20
12	rwa04.pnq20	rwa04.pnq20

9. **Proven** by this data: at hop 11 the probes reached at least two different routers in two different Pune sites.
10. **Inferred, not proven:** that these form two clean parallel chains as shown. A single-run traceroute cannot separate the chains. Paris traceroute with several fixed flow identifiers, or a multipath detection run that enumerates every parallel path, would settle it.

■ 27.5.6 WORDS – remember these

1. ECMP — using several equally good roads at once — equal-cost multi-path forwarding across multiple next hops of identical routing metric.
2. Five-tuple — the five fields that identify a conversation — source address, destination address, protocol, source port, destination port.
3. Per-flow load balancing — keep each conversation on one road — hashing the flow identifier to select a next hop, preserving packet order.
4. Per-packet load balancing — alternate every packet — round-robin next-hop selection, giving even utilization at the cost of reordering.
5. Link aggregation — several cables acting as one — LAG or bundle interfaces, hashing frames across members below the routing layer.
6. Hash polarization — two routers making the same bad choice — correlated next-hop selection when devices in series use identical hash inputs.

27.6 Stars, and how to read them honestly

■ 27.6.1 PLAIN – in simple words

1. A star in a traceroute means one thing only: no reply arrived for that probe within the waiting time.
2. It does not mean the packet stopped there. It does not mean a router is down. It does not mean anything is broken.
3. There are five common reasons for a star, and only one of them is a fault.
4. Reason one. The router is configured not to send these messages at all. Some operators disable them on purpose.
5. Reason two. The router sends them, but only a few per second, and yours was over the limit. This is very common and it is the default on many systems.
6. Reason three. Something between you and the router blocks the reply on the way back, or blocks your probe on the way out.
7. Reason four. A tunnel technology carries your packet across several machines while showing you only one hop.
8. Reason five. The destination itself is silent by policy, which is normal for large web services.
9. Stars in the middle of a trace, with normal hops after them, always mean a quiet router and never mean a fault. The packets clearly got through.
10. Stars at the end of a trace are the hardest to read, and they are extremely common. Most traces of most large services end in stars.

■ 27.6.2 PLAIN – a picture in your head

1. Go back to the corridor of doors and the paper darts.
2. You throw a dart marked 13 and no card comes back.
3. Six things could have happened, and from where you stand they look identical.
4. Nobody was at position 13, so the dart flew on and nobody kept it.
5. Someone was there but has orders never to walk cards back to strangers.
6. Someone was there, would normally reply, but has already walked one card this second and refuses to walk another.
7. Someone was there and wrote a card, but a doorman between you tore it up.
8. There was a covered walkway between position 12 and position 18, so positions 13 to 17 do not exist as far as darts are concerned.
9. Or the corridor really does end in a wall at position 13.
10. You cannot tell which. Standing at your end and staring harder does not help. You need a different experiment.

Where this comparison breaks: in the corridor, silence is unusual. On the internet, silence is the normal, expected, default state of most routers near a large service. Treat a silent tail as the baseline, not as a surprise.

■ 27.6.3 PLAIN – a worked example

1. The reader's trace ends like this.

```
12  ae106-0.rwa04.pnq20.ntwk.msn.net (104.44.20.42)
13  * * *
14  * * *
15  * * *
16  * * *
17  * * *
18  * * *
```

```

19 * * *
20 * * *

```

2. What is proven: no ICMP Time Exceeded message reached the reader's machine for any probe with a hop counter of 13 or more, within the wait time.
3. What is not proven: that the packets stopped at hop 13. That is a guess, and there is no evidence for it in this trace.
4. Compare with a control. Run the same trace to a target known to answer, on the same connection, at the same moment.
5. If that trace also ends in eight stars, the stars belong to the reader's own network or to the general environment, not to this destination.
6. If it completes cleanly, the stars are specific to this path, which narrows things a great deal without proving anything about the cause.
7. The reader did not run that control. It is the single cheapest improvement available, and section 27.13 shows how.

■ 27.6.4 PLAIN – what is really happening inside

1. Generating a Time Exceeded message is expensive for a router, in a way that forwarding is not.
2. Forwarding happens in dedicated hardware: a chip that looks up an address and moves the packet, millions of times a second, without a general-purpose processor involved.
3. Building an error message is different work. The router must copy part of the dead packet, build a new packet around it, work out a source address, and send it. That is software, on the router's ordinary processor.
4. So every serious router limits how much of it can be spent on courtesy messages to strangers. Not out of malice. Out of survival.
5. On Linux the default limit is one such message per second to any given destination, and carrier gear is usually tighter still.
6. Now count. Traceroute sends three probes to the same hop, back to back, in a few milliseconds. Under a one-per-second limit, two of them get nothing.
7. This is why partial stars, like one answer and two stars on the same line, are so common. It is a rate limit, almost every time.
8. Tunnels are the other big cause. If an operator carries your packet inside a wrapper across five internal routers, and the wrapper is set up not to copy your hop counter, those five routers never see your counter reach zero. They vanish from the trace entirely.

■ 27.6.5 TECHNICAL – the engineer's version

1. Linux rate limiting is exposed as sysctls, with documented defaults:

Sysctl	Default	Meaning
icmp_ratelimit	1000	ms between replies
icmp_msgs_per_sec	1000	global cap per second
icmp_msgs_burst	50	burst allowance
icmp_ratemask	6168	which types are limited

2. The mask value 6168 is binary with bits 3, 4, 11 and 12 set, meaning ICMP types 3, 4, 11 and 12. Type 11 is Time Exceeded. So Time Exceeded is rate limited by default on Linux, at one per second per destination.

3. RFC 1812 section 4.3.2.8 requires it: a router **MUST** implement rate limiting of ICMP error messages to avoid being used in a denial-of-service attack. Silence is standards-compliant behaviour, not misconfiguration.
4. Carrier platforms add control-plane policing, a hard rate limit on traffic punted to the route processor, usually classed so that routing protocol packets are protected and ICMP generation is not.
5. MPLS is the main cause of hidden hops. RFC 3032, “MPLS Label Stack Encoding”, January 2001, by Rosen and others, specifies that when an IP packet is first labelled, the label’s TTL is set from the IP TTL, and on the final pop the IP TTL is set from the label TTL.
6. Operators frequently disable that copying, so the label TTL starts at 255 instead of inheriting yours. Then the whole tunnel decrements a counter your probes never see, and the tunnel interior is invisible.
7. This is an implementation detail with different names per vendor: on Cisco IOS it is `no mpls ip propagate-ttl`, on Junos it is `no-propagate-ttl` under the MPLS protocol stanza.
8. RFC 4950, “ICMP Extensions for Multiprotocol Label Switching”, August 2007, by Bonica, Gan, Tappan and Pignataro, allows a label switching router to attach its label stack to ICMP Time Exceeded and Destination Unreachable messages, so a traceroute can display MPLS labels when the operator allows it.
9. Honest ranking of causes of a silent tail, most likely first, for a trace toward a large cloud provider:

Rank	Cause	How common
1	Policy: no ICMP errors	very common
2	Rate limiting	very common
3	Destination silent	very common
4	Filtered on return	common
5	MPLS hiding hops	common
6	Actual black hole	uncommon

10. Note that the genuine fault is last on that list. That ordering is the whole lesson of this section.
11. For the reader’s hops 13 to 20: the evidence is consistent with every one of those six causes, and distinguishes between none of them.

■ 27.6.6 WORDS – remember these

1. Star — no reply arrived in time — a probe timeout, recorded as an asterisk, carrying no information about the cause.
2. Rate limiting — only a few replies per second — an enforced minimum interval between generated ICMP messages, required by RFC 1812 4.3.2.8.
3. Control plane — the router’s thinking part — the route processor running protocols and generating ICMP, distinct from the forwarding hardware.
4. Control-plane policing — protecting the thinking part — a hard rate limit on traffic punted from the data plane to the route processor.
5. MPLS — a tunnelling technology used inside carriers — Multiprotocol Label Switching, which can hide interior hops by not propagating the IP TTL.
6. Black hole — a place where packets vanish silently — a forwarding or filter state that discards traffic without generating any ICMP notification.

27.7 Why the times look strange

■ 27.7.1 PLAIN – in simple words

1. A normal traceroute prints three times on each line, in milliseconds. One for each probe.
2. People expect those numbers to grow steadily down the page, because each hop is further away. Often they do not.
3. You will regularly see hop 9 answer faster than hop 8. That looks like the packet went backwards in time.
4. It did not. Each number is a full round trip: out to that router, and all the way back to you. It is not the time for that one step of the journey.
5. And the way back can be completely different from the way out. You never see the way back.
6. There is a second cause, and it is bigger than most people expect. Some routers are simply slow at producing the reply, even though they forward traffic at full speed.
7. A router that takes 30 milliseconds to write a complaint will show a high number, while the packets it forwards are delayed by almost nothing.
8. So a slow-looking hop in the middle of a trace, with fast hops after it, is proof that the router is slow at complaining and nothing else.
9. The rule to remember: you cannot subtract two lines to get the time of the link between them.

■ 27.7.2 PLAIN – a picture in your head

1. Imagine measuring the distance to towns along a road by sending a runner to each one with a message, and timing until a reply comes back to you.
2. The reply might come back by a different road, or by train, or by post. Your stopwatch includes all of that.
3. Also, each town has one clerk who writes the replies, and clerks vary. One town's clerk answers in seconds. The next town's clerk finishes lunch first.
4. Now try to work out the distance between town four and town five by subtracting your two stopwatch readings.
5. The answer is meaningless. You have subtracted two different return journeys and two different clerks.

Where this comparison breaks: the outward journey really is shared, so times do usually trend upward over many hops, and a jump that persists across every later hop is real evidence of a long link. The caution is against reading small differences, not the overall shape.

■ 27.7.3 PLAIN – a worked example

1. First, an honest statement about the evidence. The reader's captured trace contains no times at all. It records only hop numbers and addresses.
2. So the example here is a typical trace of the same shape, to show what the reader would have seen.

7	104.44.196.187	18.412 ms	17.998 ms	18.203 ms
8	104.44.55.163	41.902 ms	42.118 ms	41.777 ms
9	104.44.53.119	88.640 ms	39.115 ms	39.402 ms
10	104.44.31.62	42.881 ms	43.006 ms	42.744 ms
11	104.44.20.52	42.560 ms	44.881 ms	43.019 ms
12	104.44.20.42	43.702 ms	43.244 ms	43.610 ms

3. Read hop 9. One probe took 88 ms and the other two took 39 ms. That is not a congested link. If it were, all three would be slow.

4. It is one reply that queued behind other work on that router's processor. The forwarding path was fine, as hops 10, 11 and 12 prove by being fast.
5. Read hop 10. It is 0.7 ms faster than hop 9, despite being further away. That is well within noise and means nothing.
6. Read the jump between hop 7 and hop 8: 18 ms to 42 ms, and it sticks for every later hop. That is real. Something long sits between them.
7. That is the correct way to use these numbers. Look for jumps that persist, ignore single slow probes, and never subtract.

■ 27.7.4 PLAIN – what is really happening inside

1. Break one printed number into its parts. It contains four things and you can separate none of them.
2. Part one: the time for your probe to travel from you to that router.
3. Part two: the time that router took to notice the dead packet, hand it to its processor, and build the reply. This can be microseconds or tens of milliseconds.
4. Part three: the time for the reply to travel back to you, along a path you cannot see and which need not be the reverse of the outward path.
5. Now consider why the return path is invisible. Internet routing is chosen independently by each network for traffic entering it. The road from A to B and the road from B to A are separate decisions made by different people.
6. Paths being different in the two directions is called asymmetry, and it is the normal case between large networks, not the exception.
7. So when hop 8 shows 42 ms and hop 9 shows 39 ms, the honest reading is: hop 9's reply happened to come home by a quicker road, or hop 9's processor was less busy. Both are ordinary.

■ 27.7.5 TECHNICAL – the engineer's version

1. The measured value is the interval between the probe's transmission timestamp and the reception timestamp of the matching ICMP response, taken in user space by the traceroute process.
2. That value decomposes as forward path delay, ICMP generation latency, and return path delay, plus scheduling jitter on the measuring host itself.
3. ICMP generation latency is the dominant unexplained term. Time Exceeded generation is a control-plane function. On platforms with hardware forwarding, the probe is punted from the data plane to the route processor, queued behind protocol traffic, and answered at low priority.
4. Observed generation latencies of 10 to 100 ms on a busy carrier router are routine while the same router forwards transit traffic with under 100 microseconds of added delay.
5. This is why an intermediate hop can show a far higher RTT than hops beyond it. The high value describes the router's own responsiveness, not the path.
6. Path asymmetry means the return leg is chosen by BGP policy in the far network and may traverse entirely different transit providers, exchange points and even continents.
7. Consequently, per-link latency cannot be derived by differencing adjacent hop RTTs. The difference mixes two different return paths and two different generation latencies, and can be negative.
8. Useful practice for reading times, in order:

Signal	Meaning
One slow probe of three	Control-plane blip
All three slow, one hop	That router is busy
Step up, all later hops	Real long link

Signal	Meaning
Rising loss to the end	Silent policy, not loss

9. Critical caution when reading mtr loss columns: loss at an intermediate hop that does not appear at later hops is rate limiting, not packet loss. Only loss that continues to the final hop is real.

■ 27.7.6 WORDS – remember these

1. Round-trip time — there and back — the interval from probe transmission to matching response reception, including generation latency.
2. Path asymmetry — the way back is not the way out — independent forward and reverse routing decisions between autonomous systems.
3. Data plane — the fast part that moves packets — hardware forwarding, typically an ASIC, operating at line rate.
4. Punt — hand a packet to the slow part — transferring a packet from the forwarding hardware to the route processor for software handling.
5. Jitter — variation in delay — the statistical spread of one-way or round-trip latency across repeated measurements.

27.8 What a traceroute cannot tell you

■ 27.8.1 PLAIN – in simple words

1. This is the most important section in the chapter, so read it twice.
2. A traceroute is a weak instrument. It answers one narrow question, and people routinely use it to answer five questions it cannot touch.
3. It cannot tell you the way back. You see only the outward direction.
4. It cannot tell you whether the destination is working. Silence from a destination is normal for almost every large service.
5. It cannot tell you where a block is. The last hop that answered is the last hop that was willing to answer, which is a different thing.
6. It cannot tell you anything at all about what lies past the last reply. Not whether there are more routers, not whether your packets got there.
7. It cannot prove that a hop showing high times is causing your problem.
8. It cannot prove the path is the path your real traffic takes, because your real traffic does not change its port number on every packet.
9. Now say this plainly, because it is the reader's own case.
10. The reader's traceroute, on its own, would NOT have proved a fault. Not the stars, not the twelve hops, not any part of it.
11. Everything in that trace is compatible with a completely healthy network in which GitHub simply does not answer diagnostic probes.
12. Something else proved the fault. The trace only narrowed the region.

■ 27.8.2 PLAIN – a picture in your head

1. A traceroute is a torch shone down a corridor, at night, from one end.
2. It lights up the people who choose to wave back. It does not light up the corridor.
3. Someone standing still and silent is invisible. So is a wall. So is an open door with nothing beyond it.
4. Now imagine you conclude “the corridor ends at position 13” because nobody waves past 12.

5. You have confused “nobody waved” with “nothing is there”. That is the error almost everyone makes with traceroute.

Where this comparison breaks: the torch is worse than it sounds, because it also does not show you the route the wave took to come back. Two people standing next to each other can wave back along completely different routes, and their apparent distance from you will differ for that reason alone.

■ 27.8.3 PLAIN – a worked example

1. Here is the reader’s situation, split into what was proven and what was merely suggested.

Observation	What it proves
12 hops answered	Those 12 forwarded and replied
Hops 13–20 silent	Nothing on its own
Path enters Microsoft	Handoff happened in Delhi
Names resolve	Reverse DNS exists

2. Now the observations that came from outside the traceroute, in the same session, and what those proved.

Observation	What it proves
curl: no reply in 15 s	No TCP answer at all
No RST, no ICMP error	Silent drop, not refusal
Works on mobile data	Server is up and serving
DNS resolved fine	Not a name problem

3. Compare the two tables. The second one carries almost all of the weight.
4. The decisive fact is the combination: the same request to the same address succeeded over a different network, at the same time.
5. That rules out the destination being down, because it served the phone.
6. It also rules out DNS, because both used the same address.
7. What remains is that something on the path used by the home connection discarded the traffic without saying so.
8. The traceroute contributed one thing to that conclusion: it showed which networks were involved, so the problem could be described precisely instead of vaguely.

■ 27.8.4 PLAIN – what is really happening inside

1. The deep reason traceroute is weak is that it measures the wrong traffic.
2. Its probes are not your traffic. They have different ports, often a different protocol, a different size, and a deliberately broken hop counter.
3. Every device on the path is entitled to treat them differently from your real traffic, and many do.
4. So a perfect traceroute does not prove your real connection will work, and a completely silent traceroute does not prove it will fail.
5. There is a second, subtler weakness. Traceroute reports on the forward direction only, but a connection needs both directions to work.
6. A failure in which your packets arrive perfectly and the replies are discarded looks, from your end, exactly like a failure in which your packets never arrive.

7. Traceroute cannot separate those two cases. Nothing you run from one end can. You need a measurement from the other end.
8. This is why the honest diagnostic sequence is: use traceroute to describe the region, then use a test with real traffic to establish the fault, then get a second viewpoint from elsewhere to locate it.

■ 27.8.5 TECHNICAL – the engineer’s version

1. Enumerated limits of a single-ended forward traceroute:

Cannot determine	Why
Reverse path	Never observed
Destination liveness	Silence is policy
Filter location	Only shows repliers
State past last reply	No data at all
Real traffic’s path	Different flow ID
Which direction failed	One-ended view

2. The probe traffic differs from production traffic in transport, port, payload size, packet rate and TTL, so per-hop treatment can differ under access control lists, QoS classification, policy-based routing and stateful inspection.
3. The last responding hop is the last hop willing and able to generate an ICMP error that reached you. It carries no information about topology beyond itself.
4. A common and serious error is naming the operator of the last responding hop as the cause. In the reader’s case that would mean blaming Microsoft’s Pune router, which the data does not support in any way.
5. Correct statements for the reader’s evidence, split by strength:
6. **Proven.** Twelve hops responded, ending at 104.44.20.42, a Microsoft address inside 104.40.0.0/13, announced by AS8075. No response arrived for TTL values 13 through 20.
7. **Proven, from outside the trace.** A TCP connection to 20.207.73.82 port 443 produced no response of any kind in 15 seconds, and the identical request over a mobile network succeeded immediately.
8. **Supported, not proven.** Traffic on this path was silently discarded somewhere at or beyond the last responding hop.
9. **Not supported by any evidence here.** That a specific named party blocked it, that the discard point is hop 13, or that the trace itself demonstrates a fault.

■ 27.8.6 WORDS – remember these

1. Single-ended measurement — testing from one side only — a measurement with no observation point in the far network, blind to the reverse path.
2. Last responding hop — the last router that answered — the furthest TTL for which an ICMP error was received, with no topological meaning beyond that.
3. Silent drop — discarded without notice — a discard action that generates no ICMP error, defeating all ICMP-based diagnosis.
4. Liveness — whether something is actually working — reachability and service availability, which ICMP silence cannot establish either way.
5. Looking glass — a window into someone else’s network — a public read-only interface for running ping, traceroute and BGP queries from an operator’s own routers.

27.9 The reader's trace, decoded hop by hop

■ 27.9.1 PLAIN – in simple words

1. The trace has four zones, and once you see them the whole thing reads easily.
2. Zone one is hop 1. That is the box in the reader's own flat.
3. Zone two is hops 2 to 6. That is the reader's internet provider, using addresses that are not visible from the outside world.
4. Inside zone two, hop 3 stands out. It has an address of the kind used on the public internet, sitting between two hops that do not.
5. Zone three is hops 7 to 12. Every one of those belongs to Microsoft, which owns GitHub, and the names say which city each is in.
6. Zone four is hops 13 to 20, which is silence.
7. So the journey visible in this trace is: home, provider, Microsoft, quiet.

■ 27.9.2 PLAIN – a picture in your head

1. Think of posting a parcel and watching the tracking page fill in.
2. First scan: your own doorstep.
3. Next few scans: your local courier's vans and depots, with internal codes that mean nothing to anyone outside that company.
4. Then a scan at a handover point, where the parcel changes companies.
5. Then several scans inside the second company, whose codes are different and happen to include city names you recognize.
6. Then the tracking page stops updating.

Where this comparison breaks: a tracking page stops updating because nobody scanned the parcel, and that is exactly right here too. But a parcel is one object with one history. This trace is twenty different parcels, and some of the scans belong to journeys that never met.

■ 27.9.3 PLAIN – a worked example

1. The whole trace in one table. Names are shown without the shared ending `.ntwk.msn.net`, which every one of them carries.

Hop	Address	Name	What it is
1	192.168.0.1	none	reader's own router
2	172.31.0.17	none	ISP access, private
3	137.97.29.249	none	ISP, public address
4	172.26.22.235	none	ISP core, private
5a	172.16.18.33	none	ISP core, private
5b	172.26.14.75	none	ISP core, private
5c	172.26.22.234	none	ISP core, private
6a	172.26.14.75	none	same as hop 5b
6b	172.16.18.33	none	same as hop 5a
7	104.44.196.187	ae66-0.del01-96cbe-1b	Microsoft, Delhi
8a	104.44.55.163	be23.rwa02.bom01	Microsoft, Mumbai
8b	104.44.51.55	po22.rwa04.bom01	Microsoft, Mumbai
9a	104.44.53.119	be1010.owr02.bom01	Microsoft, Mumbai
9b	104.44.53.47	be1013.owr01.bom01	Microsoft, Mumbai
10	104.44.31.62	be5.ibr02.pnq21	Microsoft, Pune

Hop	Address	Name	What it is
11a	104.44.20.52	ae104-0.rwa03.pnq21	Microsoft, Pune
11b	104.44.16.51	be1.ibr01.pnq20	Microsoft, Pune
12	104.44.20.42	ae106-0.rwa04.pnq20	Microsoft, Pune
13-20	none	none	no reply at all

2. Rows marked a, b and c are different probes at the same hop count. They are not separate hops.

■ 27.9.4 PLAIN – what is really happening inside

1. **Hop 1, 192.168.0.1.** The reader's own router, in the flat. It is the default gateway: the machine every packet leaving the home goes to first. The address is in 192.168.0.0/16, a range reserved for private use, so millions of homes use this exact number at the same time.
2. **Hop 2, 172.31.0.17.** The first machine inside the provider. Still a private address, from the 172.16.0.0/12 range, which covers everything from 172.16.x.x to 172.31.x.x. This is the provider's access network.
3. **Hop 3, 137.97.29.249.** A public address, and the only one before Microsoft. It belongs to the reader's provider. Its presence here tells you the provider mixes public and private numbering inside its own core.
4. **Hop 4, 172.26.22.235.** Back to private. This is not a mistake and not a loop. Providers commonly number their internal links privately even when the same routers also hold public addresses on other interfaces.
5. **Hop 5, three addresses.** 172.16.18.33, 172.26.14.75 and 172.26.22.234. Three probes, three parallel links, three routers. Note that .234 is one below hop 4's .235, which is what you expect from two interfaces on the same small link subnet.
6. **Hop 6, two addresses.** 172.26.14.75 and 172.16.18.33. Both appeared at hop 5. The same routers are five hops away by one road and six by another, so they answer at both distances.
7. **Hop 7, 104.44.196.187.** The handover. The name ends in ntwk.msn.net, which is Microsoft's backbone naming domain, and the site code is del01, meaning Delhi. This is where the provider hands the traffic to Microsoft.
8. **Hop 8, two Mumbai routers.** 104.44.55.163 and 104.44.51.55, named rwa02 and rwa04 at site bom01. Mumbai. Two parallel paths, both inside Microsoft.
9. **Hop 9, two more Mumbai routers.** 104.44.53.119 and 104.44.53.47, named ovr02 and ovr01. Still Mumbai, one layer deeper in.
10. **Hop 10, 104.44.31.62.** Named ibr02.pnq21. Pune. Both parallel paths converge here, because only one address appeared.
11. **Hop 11, two Pune routers in two buildings.** 104.44.20.52 at site pnq21 and 104.44.16.51 at site pnq20. The paths diverge again, and the two roads are not the same length.
12. **Hop 12, 104.44.20.42.** Named rwa04.pnq20. Pune. The last machine on the whole internet that said anything at all to the reader's computer.

■ 27.9.5 TECHNICAL – the engineer's version

1. Address classification, with the governing document:

Address	Block	Status
192.168.0.1	192.168.0.0/16	private, RFC 1918
172.31.0.17	172.16.0.0/12	private, RFC 1918

Address	Block	Status
137.97.29.249	public	reader's ISP
172.26.x, 172.16.x	172.16.0.0/12	private, RFC 1918
104.44.x.x	104.40.0.0/13	Microsoft, AS8075

- RFC 1918, "Address Allocation for Private Internets", February 1996, reserves 10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16. The /12 spans 172.16.0.0 to 172.31.255.255, so all of hops 2, 4, 5 and 6 fall inside it.
- Proven** for the Microsoft hops: 104.44.196.187 lies inside 104.40.0.0/13, a prefix announced by AS8075, registered to Microsoft Corporation. Every hop from 7 to 12 shares that block.
- GitHub was acquired by Microsoft in 2018, and github.com in this session resolved to 20.207.73.82, inside 20.192.0.0/10, also AS8075. So the Microsoft hops are not a detour; they are the destination network.
- On hop 4 and hop 5c: 172.26.22.235 and 172.26.22.234 differ by one. Two readings are possible and a traceroute cannot separate them.
- Reading one: they are the two ends of a single point-to-point link, addressed as a /31 under RFC 3021, "Using 31-Bit Prefixes on IPv4 Point-to-Point Links", December 2000, which gives exactly two usable addresses, or as a /30 giving two usable out of four.
- Reading two: they are unrelated interfaces that merely sit close together in the same /24 the operator uses for core links.
- Mark this clearly as **inferred**. Nothing in a traceroute reveals a prefix length.
- Hop 3 being public while hops 2, 4, 5 and 6 are private is worth stating carefully. It does not imply the packet left the ISP at hop 3 and came back. A single router can hold public addresses on some interfaces and private addresses on others, and the ICMP source is the interface the reply left by.
- Reverse DNS: the names on hops 7 to 12 come from PTR lookups performed by the reader's own resolver, 1.1.1.1, after the trace collected the addresses. Hops 1 to 6 show no names because the ISP publishes no PTR records for that private space, which is normal and correct.
- Verify any of this yourself with `whois 104.44.196.187`, `dig -x 104.44.196.187 @1.1.1.1`, or an RDAP query against the ARIN registry.

■ 27.9.6 WORDS – remember these

- Default gateway — the router your machine sends everything to — the next hop for destinations outside the local subnet.
- Private address — an address only meaningful inside one network — a block from RFC 1918, never routed on the public internet.
- Reverse DNS — turning an address back into a name — a PTR record lookup under `in-addr.arpa`, published at the address holder's discretion.
- Autonomous system — one network under one routing policy — an AS such as AS8075 for Microsoft, identified by number in BGP.
- Point-to-point link — a cable with exactly two ends — a link addressed with a /30 or, under RFC 3021, a /31.

27.10 Reading carrier router names

■ 27.10.1 PLAIN – in simple words

- Large network operators give every router a name, and the name is built from parts that describe the machine.

2. The parts usually say: which cable the packet came in on, which building the router is in, which city that is, and what job the router does.
3. Once you can read them, a trace stops being a wall of characters and becomes a sentence about geography.
4. Important warning first. There is no standard for this. No specification says how to name a router.
5. Every operator invents its own scheme, and schemes change over time inside the same company.
6. So everything in this section is a **convention**: a pattern people follow because it is useful, not a rule anyone is obliged to obey.
7. Read names as strong hints, never as proof. If a name says Mumbai and the times say otherwise, believe the times.

■ 27.10.2 PLAIN – a picture in your head

1. Think of a hospital room label: 2W-14B-ICU.
2. Floor 2, west wing, room 14, bed B, intensive care.
3. Nobody outside that hospital could invent that code, but anyone inside reads it instantly and knows exactly where to walk.
4. Another hospital across the road uses a completely different code for the same idea, and both are perfectly sensible.
5. Router names work exactly like that, and `ntwk.msn.net` is one hospital's labelling system.

Where this comparison breaks: a hospital label is put there by the hospital for its own use, and so is a router name, but you are reading it from outside with no key. Some parts are guessable and some, like a device serial code, are not. Say which is which rather than inventing meanings for the rest.

■ 27.10.3 PLAIN – a worked example

1. Take the reader's hop 7 and split it at the dots.

```

ae66-0 . del01-96cbe-1b . ntwk . msn . net
|           |           |           |           |
|           |           |           |           +-- top-level domain
|           |           |           +----- Microsoft's domain
|           |           +----- network naming zone
|           +----- site and device
+----- interface on that device

```

2. Read it right to left, the way domain names are actually structured.
3. `net` is the top-level domain. `msn` is a Microsoft domain, in use since the Microsoft Network launched in 1995. `ntwk` is short for network, the zone Microsoft uses for its backbone infrastructure.
4. `del01` is the site: Delhi, facility 01. `96cbe-1b` is a device identifier that is not publicly documented, so do not guess at it.
5. `ae66-0` is the interface: aggregated Ethernet bundle 66, logical unit 0.
6. Putting it in a sentence: “logical unit 0 of bundle 66 on a device at Microsoft's Delhi site 1”.

■ 27.10.4 PLAIN – what is really happening inside

1. The name did not travel with the packet. The router never told anyone its name.
2. What happened is that traceroute collected an address, then asked DNS to turn that address back into a name. That is a reverse lookup.
3. The operator publishes those names in DNS voluntarily, one per interface address, usually generated automatically from its own inventory system.

4. That is why the interface appears in the name at all. There is one name per interface address, not one name per router.
5. So a single router shows up under different names at different hops, depending on which door your probe came in through.
6. It also means the names can be stale. An automatic system that is not run after a change leaves an old name in place for months.
7. And it means the names can be absent, which is what happened on the reader's hops 1 to 6.

■ 27.10.5 TECHNICAL – the engineer's version

1. City codes in `ntwk.msn.net` follow IATA airport codes, the three-letter codes used on luggage tags. The three in the reader's trace, verified:

Code	City	Airport
del	New Delhi	DEL, Indira Gandhi
bom	Mumbai	BOM, Chhatrapati Shivaji
pnq	Pune	PNQ, Pune Airport

2. The trailing digits are facility numbers within the metro, so `pnq20` and `pnq21` are two distinct Pune sites, not two routers in one building.
3. Role codes, as used by Microsoft. These are a **convention** of one operator, not an industry standard, and other carriers use entirely different letters:

Code	Conventional meaning
ibr	inter-region backbone router
rwa	regional wide-area router
owr	outer wide-area router

4. Reading the roles as a shape: `ibr` devices carry traffic between regions over long-haul links, `rwa` devices aggregate within a region, and `owr` devices sit further out toward peers and customers.
5. Interface prefixes all mean the same underlying thing, several physical links bonded into one logical link, but the word differs by vendor:

Prefix	Full name	Vendor usage
ae	aggregated Ethernet	Juniper Junos
be	bundle-Ether	Cisco IOS-XR
po	port-channel	Cisco IOS, NX-OS

6. Seeing `ae`, `be` and `po` in one trace is normal and suggests a multi-vendor network, which every large operator runs deliberately.
7. The `-0` in `ae66-0` and `ae104-0` is the logical unit or subinterface number, Junos style. `be23` and `po22` carry no unit, which is the Cisco style of writing the same idea.
8. Full decode of every hostname in the reader's trace. All nine end in `.ntwk.msn.net`, which is omitted here for width.

Hostname	Interface	Site and role
ae66-0.del01-96cbe-1b	ae 66, unit 0	Delhi 01, edge device
be23.rwa02.bom01	bundle 23	Mumbai 01, rwa 02
po22.rwa04.bom01	port-channel 22	Mumbai 01, rwa 04
be1010.owr02.bom01	bundle 1010	Mumbai 01, owr 02
be1013.owr01.bom01	bundle 1013	Mumbai 01, owr 01
be5.ibr02.pnq21	bundle 5	Pune 21, ibr 02
ae104-0.rwa03.pnq21	ae 104, unit 0	Pune 21, rwa 03
be1.ibr01.pnq20	bundle 1	Pune 20, ibr 01
ae106-0.rwa04.pnq20	ae 106, unit 0	Pune 20, rwa 04

- Hop 7 is the odd one out: it has no three-letter role code, only del01-96cbe-1b. **Inferred**, not proven: this is an edge or peering device named from an inventory identifier rather than a backbone role code, which is consistent with it being the handover point from the reader's ISP.
- Honest limits of name reading, in order of importance: names are published voluntarily and may be absent; they may be stale after a change; the codes are one operator's private scheme; and geolocation databases built on such names inherit every one of those errors.
- Cross-check a name before relying on it. Compare the round-trip time with the distance the name implies. A hop named Mumbai answering in 2 ms from Delhi is a stale or wrong name, not a physics breakthrough.

■ 27.10.6 WORDS – remember these

- IATA code — the three-letter airport code — the International Air Transport Association location identifier, widely reused for network site names.
- Site code — which building the router is in — the operator's facility identifier, usually a city code plus a number.
- Role code — what job the router does — a naming convention indicating the device's place in the network hierarchy, private to each operator.
- Bundle interface — several cables acting as one — a link aggregation group, called ae on Junos, Bundle-Ether on IOS-XR, Port-channel on IOS.
- Subinterface — a logical slice of one interface — a numbered unit on a physical or bundle interface, often carrying one VLAN.

27.11 The trace as a story

■ 27.11.1 PLAIN – in simple words

- Read the whole thing as one journey and it becomes memorable.
- A packet leaves a laptop in a flat and reaches a plastic box on a shelf.
- The box passes it to the provider's access network, the equipment that serves the street and the neighbourhood.
- It crosses the provider's own core, several machines using addresses that only mean something inside that company.
- It reaches the provider's public edge, the point where that company touches the rest of the internet.
- It is handed to Microsoft in Delhi.
- It travels down the country to Mumbai, then inland to Pune.
- And then nothing more is heard from it.

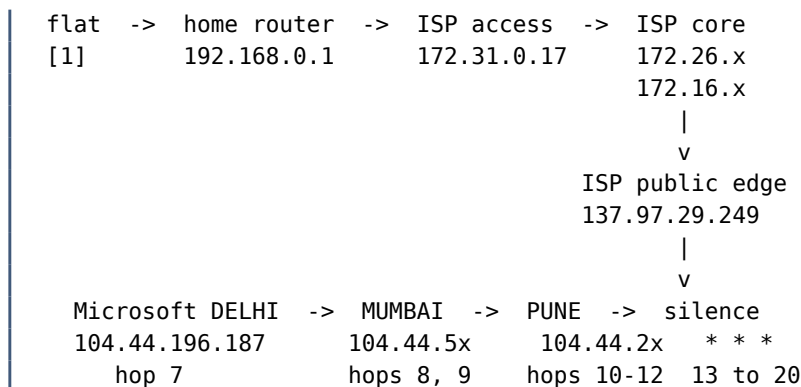
■ 27.11.2 PLAIN – a picture in your head

1. Picture a parcel leaving a house and reaching the corner post box.
2. A van takes it to the local depot, then a bigger van to the regional hub.
3. At the hub it is handed to a different company entirely, with different vans and different paperwork.
4. That company flies it from Delhi to Mumbai, then drives it to Pune.
5. At Pune the tracking page stops. The parcel is somewhere in that building or somewhere beyond it, and nobody will say which.

Where this comparison breaks: a parcel is one object, and this is not. Twenty probes went out and twelve zones answered. Also, no van driver in this story was ever obliged to scan anything for you, and most of them did it as a courtesy that they can withdraw at any time.

■ 27.11.3 PLAIN – a worked example

1. The journey in one diagram.



2. Now the distances. These are straight-line figures and are approximate.

Leg	Straight km	Fastest possible RTT
Flat to ISP edge	under 50	under 1 ms
Delhi to Mumbai	about 1,150	about 11.5 ms
Mumbai to Pune	about 120	about 1.2 ms

3. Where the fastest possible figure comes from: light in glass fibre travels at roughly 200,000 km per second, which is about 5 microseconds per kilometre one way, so 10 microseconds per kilometre there and back.
4. Real cable never runs straight, and real routers add delay, so expect real figures well above the floor.

Leg	Floor	Typical real RTT
Home to ISP edge	under 1 ms	2 to 15 ms
Delhi to Mumbai	11.5 ms	25 to 40 ms
Mumbai to Pune	1.2 ms	3 to 8 ms

5. The typical figures are approximate working numbers for Indian domestic routes, not measurements from this session.

■ 27.11.4 PLAIN – what is really happening inside

1. Notice what the story reveals about the reader's provider: it did not hand the traffic to Microsoft locally. It carried it to Delhi first.
2. Notice also that Microsoft carried it a long way inside its own network, from Delhi to Pune, roughly 1,180 km in a straight line.
3. That is normal for a large content network. Once traffic is inside, moving it on private long-haul fibre is cheap and fully controlled.
4. **Inferred, not proven:** that the reader is physically near Delhi. The Delhi handover is equally consistent with a provider that backhauls traffic from elsewhere to a Delhi interconnection point.
5. What is **proven** is only the sequence of sites: the handover happened at a Delhi-named Microsoft device, and the last reply came from a Pune-named one.
6. The destination address itself, 20.207.73.82, is also a Microsoft address, so the trace was heading toward the same network the whole time from hop 7 onward.

■ 27.11.5 TECHNICAL – the engineer's version

1. The handoff at hop 7 is an interconnection: either a private interconnect or a public internet exchange port, between the reader's ISP and AS8075.
2. A traceroute cannot distinguish those two. Both appear as one hop with the far side's address.
3. Once inside AS8075, forwarding is Microsoft's own decision. The Delhi to Mumbai to Pune ordering reflects the internal topology, and the `ibr` role code at hops 10 and 11 is consistent with inter-region long-haul devices.
4. Rough latency budget for the whole visible path, using floor values plus realistic equipment and routing overhead:

Segment	Expected RTT
Hops 1 to 3	2 to 15 ms
Hop 3 to hop 7	5 to 20 ms
Delhi to Mumbai	25 to 40 ms
Mumbai to Pune	3 to 8 ms

5. Total expected round trip at hop 12 is therefore roughly 40 to 80 ms, which is unremarkable for a domestic Indian path to a cloud edge.
6. This estimate cannot be checked against the reader's capture, because that capture contains no timing data. Note that as a gap in the evidence rather than filling it in.

■ 27.11.6 WORDS – remember these

1. Access network — the equipment serving your street — the last-mile aggregation between customer premises and the provider core.
2. Interconnection — where two networks touch — a private interconnect or internet exchange port between two autonomous systems.
3. Backhaul — carrying traffic to a bigger site before handing it off — long transport from access aggregation to a core or peering location.
4. Long-haul — a link between distant cities — high-capacity inter-city fibre, typically terrestrial DWDM within a country.
5. Latency floor — the fastest the distance allows — propagation delay at roughly 200,000 km per second in fibre, about 5 microseconds per kilometre.

27.12 What the silence after hop 12 means

■ 27.12.1 PLAIN – in simple words

1. Eight hop numbers, twenty-four probes, not one answer.
2. There are five plausible explanations, and they are not equally likely.
3. One. Microsoft's inner network does not send these diagnostic replies to outsiders, by policy. Very likely.
4. Two. Those routers do send them, but only a few per second, and traceroute asked too fast. Very likely.
5. Three. The destination itself never answers this kind of probe. Very likely, and true of almost every large web service.
6. Four. Something on the way discarded the reader's packets silently. Possible.
7. Five. Something discarded the replies on the way back. Possible.
8. The first three are ordinary behaviour that would look exactly the same on a perfectly working connection.
9. So the honest conclusion from the trace alone is: no conclusion.

■ 27.12.2 PLAIN – a picture in your head

1. You knock on twenty doors down a corridor. Twelve open. Eight do not.
2. Behind those eight doors could be an empty room, a person who ignores knocks, a person already busy answering someone else, or a wall.
3. You will not learn which by knocking harder.
4. But you have learned something real: the corridor definitely continues to door twelve, and door twelve is in Pune.

Where this comparison breaks: knocking on a real door has one meaning. A traceroute probe carries a hop counter, and a device can forward it perfectly while refusing to answer, so a silent door here may be one your packets are sailing straight past.

■ 27.12.3 PLAIN – a worked example

1. Ranked honestly, most likely first, for a trace toward a large cloud provider's inner network.

Rank	Explanation	Fault?
1	Policy: no ICMP errors	no
2	ICMP rate limiting	no
3	Destination stays silent	no
4	Return replies filtered	no
5	MPLS hides interior hops	no
6	Genuine silent discard	yes

2. Five of the six most likely explanations are not faults.
3. And note that explanation 6, the only real fault, is indistinguishable from the other five using this tool.
4. What the trace did contribute: it proved the traffic reached Microsoft's network and travelled inside it as far as a Pune device.
5. That narrows the region enormously. It rules out the reader's own router, the reader's provider, and any question of a wrong address.
6. It does not identify the cause. Not even slightly.

■ 27.12.4 PLAIN – what is really happening inside

1. The reason silence is uninformative is that all six causes produce the same observation at the reader's end: no packet arrives.
2. To distinguish them you need an observation that cannot be produced by politeness, only by a working or a broken path.
3. That is what the curl test provided. It did not ask for a courtesy reply. It attempted a real connection to a real service on port 443.
4. A working path produces an answer to that within milliseconds. A refusing server produces a refusal. A missing route produces an error message.
5. The reader got none of those. Fifteen seconds, and nothing of any kind.
6. Then the same request, to the same address, over mobile data, succeeded immediately.
7. That pair of results is what settled it, and Chapter 28 works through exactly why.
8. Keep the two roles separate in your head: the traceroute describes the region, and the connection test establishes the fault.

■ 27.12.5 TECHNICAL – the engineer's version

1. **Proven.** No ICMP Time Exceeded arrived for TTL 13 through 20 within the configured wait, across 24 probes.
2. **Not proven, and not supported by this data alone.** That forwarding stops at hop 13; that any specific party filtered anything; that the destination is unreachable.
3. Discriminating experiments, from cheapest to most conclusive:

Test	What it separates
Same trace, TCP 443	Filtering by probe type
Trace to a live control	Local versus path-specific
mttr for 100 cycles	Rate limiting from real loss
curl to port 443	Politeness from real failure
Reverse trace inbound	Which direction fails

4. The rate-limiting hypothesis is testable directly: rerun with -q 1 and a pause between probes. If hops appear that were stars at -q 3, the cause was rate limiting, not filtering.
5. The MPLS hypothesis is testable by looking for a discontinuity: a sudden large RTT step across one hop boundary, or ICMP extensions carrying label information as specified in RFC 4950.
6. The direction question cannot be answered from one end at all. It requires a probe originated inside or near the destination network, which is what a looking glass or a remote measurement platform provides.
7. Final honest reading of the reader's evidence, in one sentence: the traffic entered Microsoft's network in Delhi and was last seen alive at a Pune device, after which it was discarded silently somewhere, by something, in a direction that this trace cannot determine.

■ 27.12.6 WORDS – remember these

1. Discriminating test — an experiment whose outcomes differ by cause — a test designed so that competing hypotheses predict different observations.
2. Silent discard — dropping without notification — a filter or forwarding action producing no ICMP error, common on security policy denies.
3. Narrowing — reducing the region under suspicion — eliminating candidate causes without identifying the actual one.

4. Control test — the same test against a known-good target — a baseline run that separates environment-specific from target-specific behaviour.
5. Bidirectional visibility — seeing both directions — measurement from both ends, required to determine which direction of a path fails.

27.13 How to run a much better traceroute

■ 27.13.1 PLAIN – in simple words

1. The default traceroute is a 1987 tool with 1987 defaults, and it is the wrong tool for a modern web problem.
2. Four changes make it far better, and all four are one-letter flags.
3. Change the probe type to match the traffic that is actually failing. For a website, that means a real web connection to port 443.
4. Turn off name lookups. During a network fault, name lookups are slow or broken and they make the tool look stuck when it is not.
5. Shorten the waiting time, so a silent path finishes in seconds instead of minutes.
6. Then run the same command again against something known to work, so you have a comparison.
7. After that, run a tool that keeps probing, because one snapshot cannot show a fault that comes and goes.
8. Finally, get somebody else to trace back toward you, because you cannot see the return path from where you are standing.

■ 27.13.2 PLAIN – a picture in your head

1. A doctor who takes one temperature reading, of one patient, with a broken thermometer, has learned very little.
2. Take the reading with the right instrument. Take it again on a healthy person to check the instrument. Take it repeatedly over time. And ask a colleague at the other hospital what they see.
3. Those four steps are exactly the four commands below.

Where this comparison breaks: a patient stays still. A network path changes between measurements, so two runs a minute apart can legitimately differ, and that difference is itself information rather than an error.

■ 27.13.3 PLAIN – a worked example

1. The commands, for macOS, in the order to run them.

```
# 1. The real test: TCP to the port that is failing.
sudo traceroute -P TCP -p 443 -n -q 1 -w 2 -m 30 20.207.73.82

# 2. The control: identical flags, a target known to answer.
sudo traceroute -P TCP -p 443 -n -q 1 -w 2 -m 30 1.1.1.1

# 3. For comparison, the ICMP method Windows uses by default.
sudo traceroute -I -n -q 3 -w 2 -m 30 20.207.73.82

# 4. The classic UDP method, which needs no special permission.
traceroute -n -q 3 -w 2 -m 30 20.207.73.82

# 5. Continuous, with per-hop loss, over TCP port 443.
sudo mtr -T -P 443 -n --report --report-cycles 100 20.207.73.82
```

2. What each flag does.

Flag	Meaning
-P TCP	Send TCP probes, not UDP
-p 443	Aim at the HTTPS port
-I	Send ICMP echo instead
-n	Do not look up names
-q 1	One probe per hop
-w 2	Wait 2 seconds, not 5
-m 30	Stop after 30 hops

3. Why -q 1 with TCP: each TCP probe leaves a half-open connection at the target until it times out. Sending fewer is both faster and politer.
4. Why -n matters most during a fault: without it, every hop triggers a reverse name lookup, and if DNS is affected the tool appears to hang.
5. Worst case with these flags is 30 hops times 1 probe times 2 seconds, which is 60 seconds. With the defaults it is 450 seconds.

■ 27.13.4 PLAIN – what is really happening inside

1. A **looking glass** is a web page or login service run by a network operator that lets outsiders run a few read-only commands on that operator's own routers.
2. Typically you can run ping, traceroute, and queries about which routes that operator knows.
3. Operators provide them so that other engineers can diagnose problems involving their network without having to phone anybody.
4. Why it matters here: a looking glass inside or near the destination network can trace back toward the reader's address.
5. That gives you the return path, which is the one thing a traceroute from your own machine can never show you.
6. Where to find them: most large operators publish one, and there are public directories of them maintained by the peering community. Measurement platforms such as RIPE Atlas do the same job from thousands of probes.
7. A cheaper version of the same idea, which the reader already had available: run the test from a phone on mobile data. That is a second viewpoint on a completely different path.

■ 27.13.5 TECHNICAL – the engineer's version

1. Flag reference, quoting the macOS manual page:

Flag	Documented meaning
-P proto	UDP, TCP, GRE and ICMP
-I	ICMP ECHO, synonym for -P icmp
-p port	Base port, default 33434
-q n	Probes per ttl, default 3
-w s	Wait, default 5 sec
-m n	Max ttl, default from sysctl
-n	Numeric addresses only
-f n	Set the first ttl used

2. -P TCP and -I require raw sockets, so run them with sudo.

3. `mtr` flags used above: `-T` selects TCP probes, `-P 443` sets the port, `-n` disables name lookup, `--report` prints a summary rather than a live display, `--report-cycles 100` sets how many rounds to run.
4. Reading `mtr` output correctly is the single highest-value skill here: loss at an intermediate hop that does not persist to the final hop is ICMP rate limiting, not packet loss. Only loss that continues to the last hop is real.
5. For flow-stable path discovery use `paris-traceroute`, or `scamper` from CAIDA in its multipath detection mode, or `dublin-traceroute`. These map the parallel paths instead of blending them.
6. For a reproducible record, save output with a timestamp, for example by redirecting to a file named with the date, and keep the control run alongside it. A trace without a control run is much weaker evidence.
7. Recommended minimum evidence set for reporting a path fault to a provider: the TCP traceroute to the failing target, the identical traceroute to a working control, an `mtr` report of at least 100 cycles, the `curl -v` output showing the failure, and the same `curl` from a second network.

■ 27.13.6 WORDS – remember these

1. Looking glass — a public window onto an operator's routers — a read-only interface offering ping, traceroute and BGP route queries.
2. Control run — the same test on something known to work — a baseline measurement isolating target-specific behaviour from environmental noise.
3. Reverse traceroute — a trace run toward you from elsewhere — a measurement originated near the destination, revealing the return path.
4. RIPE Atlas — a global network of small measurement probes — a distributed active measurement platform operated by the RIPE NCC.
5. Half-open connection — a connection request never completed — a TCP session left in SYN-RECEIVED at the target, consuming state until it times out.

27.98 Common wrong ideas

1. Wrong: traceroute asks the network to report the path. Right: there is no such request. It provokes error messages by sending packets designed to die.
2. Wrong: the trace shows the route your traffic takes. Right: it shows a blend of routes taken by dozens of separate probes, each of which may be hashed onto a different link.
3. Wrong: TTL means time, so a slow link uses more of it. Right: it counts hops. One router costs exactly one, whether it holds the packet for a microsecond or a second.
4. Wrong: stars mean the packet stopped there. Right: stars mean no reply arrived. The packet may have sailed straight through a router that simply does not answer.
5. Wrong: several addresses on one hop is a bug. Right: it is load balancing across parallel links, and it is a sign of a well-built network.
6. Wrong: subtract two hops' times to get the latency of the link between them. Right: each time is a full round trip on a different return path, with different reply-generation delays. The subtraction is meaningless.
7. Wrong: a slow hop in the middle is causing your problem. Right: if the hops after it are fast, that router is merely slow at generating replies, and its forwarding is fine.
8. Wrong: the last responding hop is where the fault is. Right: it is the last hop that was willing and able to answer, which says nothing about what lies beyond it.
9. Wrong: the reader's traceroute proved a fault. Right: it proved nothing of the kind. Every line of it is compatible with a healthy network. The `curl` timeout plus success on mobile data is what

established the fault.

10. Wrong: router hostnames are a standard you can rely on. Right: they are each operator's private convention, published voluntarily, sometimes stale, and always to be cross-checked against measured times.

27.99 Chapter summary in 20 lines

1. IP has no field that records a path, so there is no way to ask for one.
2. Traceroute works by abusing the TTL field, a safety counter meant only to stop packets looping forever.
3. TTL is 8 bits, counts hops not seconds, and every forwarding router must decrement it by exactly one, per RFC 1812 section 5.3.1.
4. Common starting values are 64 on Linux and macOS, 128 on Windows and 255 on much network gear, which lets you guess a sender's system, weakly.
5. Send TTL 1 and the first router discards the packet and returns ICMP Time Exceeded, type 11 code 0, sourced from its own interface address.
6. Repeat with TTL 2, 3, 4 and so on. Three probes per hop is the default, and each round-trip time is measured per probe.
7. Van Jacobson wrote traceroute in 1987 at Lawrence Berkeley Laboratory, from a suggestion by Steve Deering.
8. The classic method sends UDP to ports from 33434 upward; Windows `tracert` sends ICMP echo; TCP traceroute sends a SYN to a real port such as 443.
9. TCP to 443 gets furthest through modern filtering because it looks exactly like the traffic the path is built to carry. It is the right tool here.
10. Multiple addresses on one hop are equal-cost multi-path load balancing. The reader's hops 5, 6, 8, 9 and 11 all show it, and all are normal.
11. Classic traceroute changes its destination port per probe, which changes the load-balancing hash. Paris traceroute, from 2006, fixes exactly that.
12. Stars mean only that no reply arrived. Causes include policy, rate limiting, filtering, MPLS hiding hops, and a silent destination.
13. Linux rate limits ICMP Time Exceeded to one per second per target by default, which alone explains most partial star lines.
14. Stars at the end of a trace are extremely common and prove nothing. The reader's hops 13 to 20 are consistent with six different causes.
15. Round-trip times can go down as you go further out, because they include reply-generation time on a slow control plane and an unseen return path.
16. You cannot subtract two hops' times to get a link latency.
17. The reader's trace reads: flat, home router, ISP access, ISP core with one public address at hop 3, Microsoft's Delhi edge, Mumbai, Pune, silence.
18. Names decode as `ntwk.msn.net` for Microsoft's backbone, IATA city codes `del`, `bom` and `pnq` for Delhi, Mumbai and Pune, role codes `ibr`, `rwa` and `owr`, and interface prefixes `ae`, `be` and `po` all meaning bonded link bundles.
19. Say it plainly: the reader's traceroute alone would not have proved a fault. It narrowed the region and nothing more.
20. What settled it was a real connection attempt that received no response of any kind, and the same request succeeding over mobile data.

TCP, Ports, and Why a Timeout Is Evidence

28.0 What this chapter gives you

1. You will be able to say exactly what the internet's basic delivery service promises, and the much longer list of things it does not promise.
2. You will be able to explain TCP and UDP as two answers to the same problem, and say why UDP is not a worse TCP but a different tool.
3. You will be able to explain a port as a 16-bit number, and a socket as the four numbers that make one conversation unique on the whole planet.
4. You will be able to draw the three-way handshake with real sequence numbers, walk the eleven states of a connection, and explain the four-way close.
5. You will be able to explain how TCP turns an unreliable service into a reliable one, and what it costs in round trips.
6. You will be able to separate flow control from congestion control, and say why a link that loses one packet in a hundred destroys throughput.
7. You will be able to name the three possible endings of a connection attempt, and tell them apart from the failure message alone.
8. You will be able to state precisely what the reader's fifteen-second silence proves, and be equally precise about what it does not prove.
9. You will be able to run `curl -v`, `nc -vz`, `nmap` and `tcpdump` and say what each one adds to the evidence.

28.1 What IP alone gives you, and what it does not

■ 28.1.1 PLAIN – in simple words

1. The internet's basic delivery service is called IP, short for Internet Protocol. It moves one small parcel of data from one address to another.
2. That parcel is called a packet. It carries a from-address, a to-address, and some data.
3. IP makes exactly one promise, and it is a weak one: it will try. The official phrase for it is **best effort**.
4. IP does not promise the packet will arrive, nor that ten packets sent in order will arrive in order, nor that a packet arrives only once.
5. Worst of all, IP does not promise to tell you when a packet is lost. Nobody rings a bell. The packet just stops existing.
6. So something had to be built on top of IP that turns "try" into "it arrived, complete, in order, exactly once". That something is TCP.

■ 28.1.2 PLAIN – a picture in your head

1. Imagine a postal system with no tracking, no receipts and no complaints desk.
2. Most postcards arrive. Some do not. Nobody tells you which.
3. Now send a 300-page book through this system, one page per postcard. The receiver gets a pile with pages missing, out of order, maybe one twice.
4. To fix it you would number every page, and the receiver would write back saying which numbers arrived, and you would re-post the gaps.
5. That numbering-and-writing-back scheme, done automatically, is TCP.

Where this comparison breaks: real post loses letters mostly by accident. IP loses packets mostly on purpose. A router with a full queue drops the newest packet deliberately, because dropping is the only signal it has to tell senders to slow down. Loss is not a malfunction in IP. Loss is the brake pedal.

■ 28.1.3 PLAIN – a worked example

1. Suppose you send ten packets numbered 1 to 10 across the internet.
2. A router in the middle has a full queue for a few milliseconds and drops two.
3. Two others take a different path because the network is load balancing, the same behaviour the reader saw at hops 5, 6, 8, 9 and 11 of their traceroute.

Sent	Arrived	Order seen	Note
1 and 2	yes	1st, 2nd	fine
3	no	–	dropped in queue
4	yes	4th	out of order
5	yes	3rd	took faster path
6	no	–	dropped in queue
7 to 10	yes	5th to 8th	fine

4. The receiver has eight packets, two gaps, and one pair swapped. IP has done its job correctly. Nothing is broken.
5. Nobody sent any message saying “3 and 6 are gone”. The sender must work it out by noticing that 3 and 6 were never confirmed.
6. That is precisely what TCP does, and it is the whole reason TCP exists.

■ 28.1.4 PLAIN – what is really happening inside

1. Each router looks only at the destination address in the packet header, decides the next hop, and forwards it.
2. Routers hold packets in a fixed-size queue while the outgoing link is busy. When the queue is full, an arriving packet is discarded. That rule is called tail drop, and no record of the packet is kept anywhere.
3. Sometimes the router sends back a complaint using ICMP, the Internet Control Message Protocol, the mechanism behind “host unreachable” and traceroute.
4. But ICMP is carried in IP packets, so complaints can be lost too, and many networks throttle or block ICMP on purpose.
5. The reader saw exactly that: hops 13 to 20 of their traceroute returned * * *, which usually means ICMP replies were suppressed, not a broken path.

■ 28.1.5 TECHNICAL – the engineer’s version

1. IPv4 is RFC 791, published September 1981 by Jon Postel at the Information Sciences Institute for DARPA. IPv6 is RFC 8200, July 2017.
2. The IPv4 service model is unreliable, connectionless, best-effort datagram delivery. There is no connection state in any router.
3. The idea comes from Vint Cerf and Bob Kahn’s paper “A Protocol for Packet Network Intercommunication”, IEEE Transactions on Communications, May 1974.
4. In 1978 the design was deliberately split into IP for delivery and TCP for reliability, by Cerf, Postel and Danny Cohen, so packet voice could skip the reliability part. That split is why UDP exists. ARPANET switched from NCP to TCP/IP on 1 January 1983, a cutover still remembered as “flag day”.

Property	Provided by IP	Provided by TCP
Addressing of hosts	yes	no
Delivery guarantee	no	yes
Ordering	no	yes
Duplicate removal	no	yes
Data integrity check	header only	full segment
Identify the program	no	yes, via ports

5. Observe raw IP behaviour with `ping -s` for size effects, `tracert` for the path, and `tcpdump -n -v ip` to read headers on the wire.

■ 28.1.6 WORDS – remember these

1. Packet — one small parcel of data with an address on it — an IP datagram carrying a header and a payload.
2. Best effort — we will try, we promise nothing — a service model with no delivery, ordering or duplicate-suppression guarantees.
3. ICMP — the network’s complaint messages — the Internet Control Message Protocol, RFC 792, advisory and frequently filtered.

28.2 TCP and UDP: two ways to use IP

■ 28.2.1 PLAIN – in simple words

1. Sitting directly on top of IP there are two main choices, and almost everything you use picks one of them.
2. **TCP**, the Transmission Control Protocol, is the careful one. It sets up a connection first, numbers everything, checks everything arrived, and puts it back in order.
3. **UDP**, the User Datagram Protocol, is the plain one. It adds almost nothing. It sends your parcel and forgets about it.
4. It is tempting to say TCP is good and UDP is bad. That is wrong. TCP’s care costs time, and repairing a missing piece can be worse than living without it.
5. In a live voice call, a lost slice of sound from 300 milliseconds ago is worthless, so a voice call uses UDP and carries on. A file download uses TCP, because a file with a hole in it is not a file.
6. TCP trades time for completeness. UDP trades completeness for time.
7. A third name you will hear is QUIC. QUIC runs on top of UDP and rebuilds reliability in a smarter way. Chapter 32 covers it properly.

■ 28.2.2 PLAIN – a picture in your head

1. TCP is a registered parcel service with signatures. Before sending, the courier phones ahead and confirms someone is home.
2. Every parcel is numbered and the receiver signs for each one. If a signature does not come back, the courier sends that parcel again.
3. UDP is a person throwing newspapers from a bicycle. No phone call, no signature, no numbering, no second attempt. Fast, cheap, mostly fine.
4. For a newspaper, mostly fine is fine. For a contract it is a disaster.

Where this comparison breaks: the courier does not really phone ahead to a human. The three-way handshake is between two pieces of software, and it exists mainly to agree on starting numbers, not to check that anyone is home. Also, the newspaper thrower does at least check the paper is not torn: UDP has a checksum, optional over IPv4 and mandatory over IPv6.

■ 28.2.3 PLAIN – a worked example

1. Here are the two headers side by side, drawn to the same scale. Each row is 32 bits, which is 4 bytes.

TCP header, 20 bytes minimum, up to 60 with options

0										1										2										3														
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1													
+										+										+										+														
										Source Port																				Destination Port														
+										+										+										+														
										Sequence Number																																		
+										+										+										+														
										Acknowledgement Number																																		
+										+										+										+														
					Offset										Reserved										Flags										Window									
+										+										+										+														
										Checksum																				Urgent Pointer														
+										+										+										+														
										Options, 0 to 40 bytes																																		
+										+										+										+														

UDP header, exactly 8 bytes, always

Source Port																Destination Port													
Length																Checksum													

2. UDP then stops. TCP keeps going for another 12 bytes minimum, and those 12 bytes are the entire machinery of reliability.
3. Sending 100 bytes by UDP costs 8 bytes of overhead. By TCP it costs at least 20 bytes plus a three-packet handshake: a bad deal for one short message, invisible for a 50 MB download.

■ 28.2.4 PLAIN – what is really happening inside

1. When a program sends by UDP, the operating system writes an 8-byte header, hands the result to IP, and the call returns. Nothing is remembered, and if the message is lost, no software anywhere notices.

2. When a program sends by TCP, the operating system copies the data into a send buffer belonging to that connection, and it sits there until confirmed.
3. The kernel keeps a timer per connection. If a confirmation does not arrive in time, it sends the data again from the buffer. That is why TCP needs a connection: it is the place to keep the buffer, the timers and the numbering.
4. And TCP does not preserve your write boundaries. Three writes of 100 bytes may arrive as one read of 300. TCP is a stream of bytes, not messages.
5. UDP does preserve boundaries. One send equals one receive, or nothing at all. That single difference decides many design arguments.

■ 28.2.5 TECHNICAL – the engineer’s version

1. TCP is now specified in RFC 9293, “Transmission Control Protocol (TCP)”, STD 7, published August 2022, edited by Wesley Eddy.
2. RFC 9293 obsoletes the original RFC 793 of September 1981 and folds in RFC 879, RFC 2873, RFC 6093, RFC 6429, RFC 6528 and RFC 6691. If you cite one document for TCP today, cite RFC 9293.
3. UDP is RFC 768, published 28 August 1980 by Jon Postel. It is three pages long. That is the entire specification and it has never needed replacing.
4. QUIC is RFC 9000, May 2021. It runs over UDP and implements its own loss recovery, congestion control and encryption. HTTP/3 over QUIC is RFC 9114, June 2022. Chapter 32 covers this in full.

Application	Transport	Why
HTTP/1.1 and HTTP/2	TCP	needs every byte
HTTP/3	QUIC over UDP	faster setup
SSH, git over SSH	TCP	needs every byte
Email, SMTP, IMAP	TCP	needs every byte
DNS query	UDP mostly	one small round trip
DNS over TLS	TCP	size and privacy
Voice and video call	UDP	late data is useless
Live game state	UDP	newest state wins
NTP time sync	UDP	tiny, retry is cheap
DHCP address lease	UDP	no address yet
Streaming video	TCP or QUIC	buffered, needs all bytes

5. Note also that streaming video is not a UDP application. Netflix and YouTube send video over TCP or QUIC, because a video file with holes will not decode. It is live conversation, not video, that needs UDP.
6. Observe both with `ss -tan` and `ss -uan` on Linux, or `netstat -an -p tcp` on macOS.

■ 28.2.6 WORDS – remember these

1. TCP — the careful delivery service — a connection-oriented, reliable, ordered byte-stream transport protocol, RFC 9293.
2. UDP — the plain delivery service — a connectionless, unreliable, message-oriented transport protocol, RFC 768.
3. QUIC — a modern replacement — a UDP-based transport with integrated TLS 1.3 and per-stream loss recovery, RFC 9000.

28.3 Ports, sockets, and the four-tuple

■ 28.3.1 PLAIN – in simple words

1. An IP address gets a packet to the right machine. It does not say which program on that machine should receive it, and one laptop may run a browser, a mail client and a chat app all talking to the network at once.
2. So each packet also carries a number saying which program it is for. That number is called a **port**.
3. A port is a 16-bit number. Sixteen bits gives 65,536 values, 0 to 65,535.
4. Every TCP packet carries two: a source port, saying which program sent it, and a destination port, saying which program should get it.
5. Servers use fixed, agreed numbers so clients know where to knock. Web servers using TLS listen on port 443. Everyone knows this.
6. Clients need no fixed number. When your browser opens a connection, the operating system hands it a spare high-numbered port, used once and released.

■ 28.3.2 PLAIN – a picture in your head

1. Think of a large office block with one street address, which is the IP address. It gets the post van to the building.
2. Inside there are 65,536 numbered pigeonholes. A pigeonhole number is a port.
3. Reception sorts every letter into a pigeonhole by the number on it. Some pigeonholes have a standing arrangement: 443 is always the web desk, 22 is always the remote-access desk, and anyone can look this up.

Where this comparison breaks: a real pigeonhole holds one item at a time. Port 443 on a busy server holds hundreds of thousands of simultaneous conversations, because a conversation is identified by four numbers together, not by the port alone. The next block fixes that picture properly.

■ 28.3.3 PLAIN – a worked example

1. When the reader ran `curl -v https://github.com`, the operating system picked a spare local port. On macOS these come from the range 49152 to 65535.
2. Say it picked 51514. The connection is then described by four numbers.

source IP	192.168.0.101	the laptop, on the home LAN
source port	51514	picked by macOS, used once
destination IP	20.207.73.82	what DNS returned for github.com
destination port	443	the agreed port for HTTPS

3. Those four values together are the **four-tuple**, and they identify this one conversation and no other.
4. The laptop address here is an example inside the reader's home range: the session record kept the router address 192.168.0.1 but not the laptop's own. Everything else is exactly as recorded.

Field	Tab 1	Tab 2	Someone else
Source IP	192.168.0.101	192.168.0.101	203.0.113.9
Source port	51514	51515	51514
Dest IP	20.207.73.82	20.207.73.82	20.207.73.82
Dest port	443	443	443

5. Look at column two and column four. Same source port, 51514, and no conflict, because the source IP differs. The tuple as a whole must be unique. That is how one server port holds hundreds of thousands of conversations at once.

■ 28.3.4 PLAIN – what is really happening inside

1. Inside the operating system there are two kinds of socket, and confusing them causes real bugs.
2. A **listening socket** is a server saying “send me anything arriving on port 443”. It is identified by only two things: a local address and a local port. It never carries data; its only job is to catch new connection attempts.
3. A **connected socket** is one live conversation, identified by all four values: local IP, local port, remote IP, remote port.
4. When a connection request arrives, the kernel finds the listening socket, creates a brand new connected socket for that one client, and queues it.
5. The server program calls `accept`, which takes one connected socket off the queue and hands it over as a new file descriptor.
6. When a packet arrives, the kernel first matches all four values against connected sockets. Only if nothing matches does it try listening sockets.
7. If nothing matches at all, the kernel replies with a TCP RST, which produces the message “Connection refused”. Remember that reply. It is the centre of this whole chapter.

■ 28.3.5 TECHNICAL – the engineer’s version

1. Port numbers are 16-bit unsigned integers. IANA divides the space by RFC 6335, published August 2011.
2. System Ports are 0 to 1023 and on Unix require root or a capability to bind. User Ports are 1024 to 49151. Dynamic or Private Ports are 49152 to 65535.
3. Ephemeral port ranges are an implementation detail, not a standard. Linux defaults to 32768 to 60999 in `net.ipv4.ip_local_port_range`. macOS and the BSDs default to 49152 to 65535 via `net.inet.ip.portrange.first`.

Port	Proto	Service	What it is for
20	TCP	FTP data	file bytes, old
21	TCP	FTP control	commands only
22	TCP	SSH	shell, scp, git over SSH
23	TCP	Telnet	plain text, obsolete
25	TCP	SMTP relay	server to server mail
53	UDP+TCP	DNS	name to address
67	UDP	DHCP server	hands out leases
68	UDP	DHCP client	receives the lease
80	TCP	HTTP	web, no encryption
110	TCP	POP3	fetch and delete mail
123	UDP	NTP	clock synchronization
143	TCP	IMAP	mail kept on server
443	TCP+UDP	HTTPS	web over TLS, QUIC on UDP

Port	Proto	Service	What it is for
465	TCP	SMTPS submission	mail send, implicit TLS

Port	Proto	Service	What it is for
587	TCP	SMTP submission	mail send, STARTTLS
853	TCP	DNS over TLS	private DNS queries
993	TCP	IMAPS	IMAP with implicit TLS
3306	TCP	MySQL, MariaDB	database
3389	TCP	RDP	Windows remote desktop
5432	TCP	PostgreSQL	database
6379	TCP	Redis	in-memory data store
8080	TCP	HTTP alternate	dev servers, proxies

- Port 443 carries HTTP over TLS, defined for that use in RFC 2818, May 2000. Port 853 for DNS over TLS is RFC 7858, May 2016. Port 465 was revoked and then restored for submission with implicit TLS by RFC 8314, January 2018; port 587 for submission is RFC 6409.
- Port 443 became the universal port because firewalls and captive networks nearly always permit outbound 443, and because the traffic is encrypted, so middleboxes cannot easily inspect or rewrite it.
- So HTTPS is now a general-purpose tunnel. Git, container registries, remote desktops, VPNs and messaging protocols all offer a 443 mode. That is a convention driven by firewall reality, not a design goal.
- It is directly relevant to the reader's session: git over SSH uses port 22, git over HTTPS uses port 443, and in the reader's outage those two transports did not behave the same way.
- Inspect with `ss -ltn` on Linux, `lsof -i -P -n` on macOS, and `ss -tn state established` for live connections.

■ 28.3.6 WORDS – remember these

- Port — a number saying which program a packet is for — a 16-bit unsigned field in the TCP or UDP header.
- Four-tuple — the four numbers that make one conversation unique — source IP, source port, destination IP, destination port.
- Ephemeral port — a temporary number for outgoing calls — a dynamically assigned local port, 49152 to 65535 on macOS, 32768 to 60999 on Linux.

28.4 The three-way handshake

■ 28.4.1 PLAIN – in simple words

- Before TCP carries a single byte of your data, the two sides exchange three small packets. This is the **three-way handshake**.
- Packet one, client to server: I want to open a connection, and my counting starts at this number.
- Packet two, back: agreed, I heard your number, and my counting starts at this other number. Packet three, forward: agreed, I heard your number too.
- After those three, both sides know the other is there, both know where the other's numbering starts, and data can flow.
- The names come from flags in the header. The first packet has the SYN flag set, for synchronize. The second has SYN and ACK. The third has ACK.
- Notice the cost. The client cannot send data until packet three, and packet three needs packet two first. That is one complete round trip of pure waiting before any useful byte moves.

7. This handshake is also exactly where the reader's problem lived. Their packet one went out and packet two never came back.

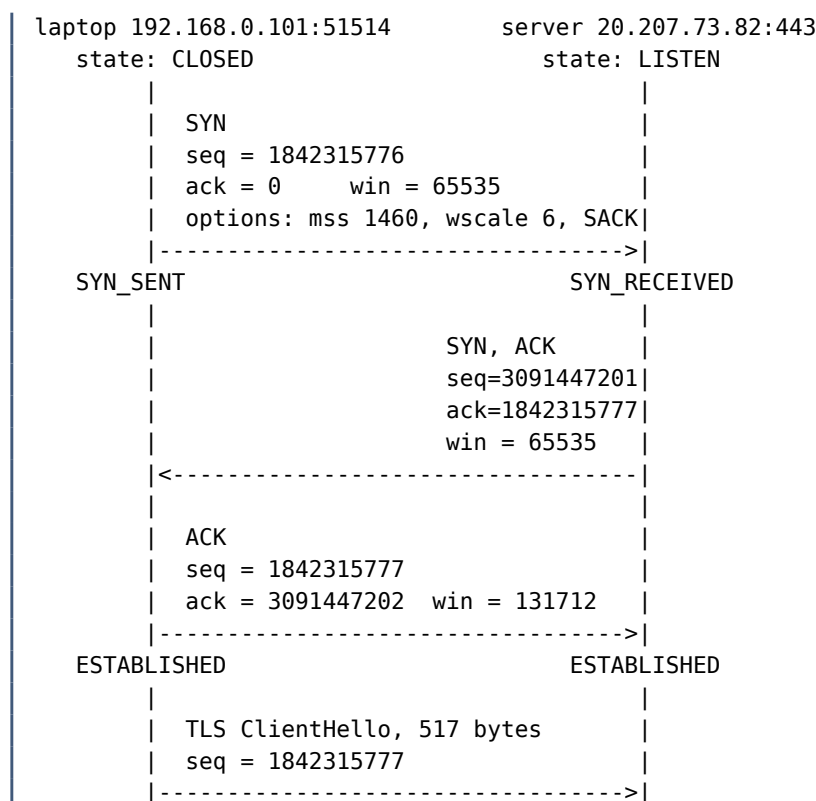
■ 28.4.2 PLAIN – a picture in your head

1. Two people want to talk on a bad radio link, and must be sure both directions work before they start.
2. Aisha says: "Aisha here, my messages will be numbered starting at 481."
3. Ben replies: "Ben here, I heard 481, so I expect 482 next. Mine start at 907."
4. After Ben's reply, Aisha knows the link works both ways, because she got an answer. But Ben knows his link works only one way until Aisha's third message arrives. That is why three messages are needed and not two.

Where this comparison breaks: real people can repeat themselves and improvise. TCP cannot. If Ben's reply is lost, Aisha has no way to tell whether Ben never heard her or Ben answered and the answer vanished. She can only send her opening line again and wait. Section 28.8 is about what she may conclude from silence.

■ 28.4.3 PLAIN – a worked example

1. Here is the reader's own connection, packet by packet, with realistic numbers filled in. The starting numbers are chosen unpredictably by each side.



2. Read the acknowledgement numbers carefully. The server's ack is 1842315777, the client's start plus one.
3. That plus one is not a data byte. The SYN flag itself consumes one number in the sequence space, so the handshake can be acknowledged reliably. The client's ack of 3091447202 does the same to the server's start.

- The window value 65535 is a raw 16-bit field. The `wscale 6` option means “multiply my future windows by 2 to the power 6”, which is 64. So the later window of 131712 means about 8.4 megabytes of buffer space.

■ 28.4.4 PLAIN – what is really happening inside

- The application calls `connect`. The kernel allocates a connection control block, picks a source port, picks an initial sequence number, moves the socket to state `SYN_SENT`, and sends the SYN.
- A retransmission timer starts, initially one second.
- At the server, the SYN arrives and the kernel checks whether any socket is listening on port 443. If yes, it creates a small half-open entry, replies with SYN-ACK, and enters `SYN_RECEIVED`.
- When the client’s ACK arrives, the server moves the entry into the accept queue, marks it `ESTABLISHED`, and wakes any program waiting in `accept`.
- On the client, the SYN-ACK cancels the retransmission timer, moves the socket to `ESTABLISHED`, and unblocks the `connect` call.
- Only now does the application start writing. For HTTPS the first thing written is a TLS ClientHello, not the web request.
- If the SYN-ACK never arrives, the timer fires and the SYN is sent again with the wait doubled. That behaviour is the whole of section 28.8.

■ 28.4.5 TECHNICAL – the engineer’s version

- RFC 9293 section 3.5 defines the handshake. The three segments carry these exact control bits.

Segment	Flags set	Seq	Ack
1, client to server	SYN	ISN_c	not valid
2, server to client	SYN, ACK	ISN_s	ISN_c + 1
3, client to server	ACK	ISN_c + 1	ISN_s + 1

- Initial sequence numbers must not be predictable. RFC 9293 section 3.4.1, folding in RFC 6528 by Fernando Gont and Steven Bellovin, February 2012, recommends this construction.

```
ISN = M + F(localip, localport, remoteip, remoteport, secretkey)
```

```
M = a timer that ticks about every 4 microseconds
```

```
F = a cryptographic hash, keyed with a host secret
```

- The reason is security. If an attacker can guess your next sequence number, they can inject data or forge a connection without ever seeing your packets. That is a blind in-window attack.
- The famous demonstration was the attack on Tsutomu Shimomura’s machines on 25 December 1994, attributed to Kevin Mitnick, which used sequence-number prediction with IP source spoofing.
- Steven Bellovin had described the weakness in his 1989 paper “Security Problems in the TCP/IP Protocol Suite”, and later wrote RFC 1448, May 1996, the first randomization recommendation. RFC 5961, August 2010, adds hardening against blind RST and SYN injection.
- The SYN and FIN flags each consume one sequence number, so they can be acknowledged and retransmitted with the same machinery as data.
- Handshake cost is exactly one round-trip time before the first application byte. TLS 1.3 adds one more round trip and TLS 1.2 adds two, as Chapter 29 covers.
- Observe with `tcpdump -n 'tcp[tcpflags] & tcp-syn != 0'`.

■ 28.4.6 WORDS - remember these

1. Handshake — the short exchange that opens a connection — the SYN, SYN-ACK, ACK sequence defined in RFC 9293 section 3.5.
2. SYN — the flag meaning “let us start counting” — the synchronize control bit, which consumes one sequence number.
3. ACK — the flag meaning “I received up to here” — the acknowledgement control bit, making the acknowledgement number field valid.
4. Initial sequence number — the random starting count — the ISN, chosen per RFC 6528 as a timer plus a keyed hash of the four-tuple.

28.5 The TCP state machine and the four-way close

■ 28.5.1 PLAIN - in simple words

1. A TCP connection is not just a pipe. It is a small machine with a memory of where it currently is. That memory is called its **state**.
2. There are eleven states worth knowing, and a connection moves between them only when a specific packet arrives or the program does something.
3. Opening is fast: three packets and you are in ESTABLISHED, the state where data flows.
4. Closing is slower, because a TCP connection is really two one-way streams, and each direction must be closed separately.
5. So closing takes four packets, not three. One side says “I am done sending”. The other says “understood”. Then the same happens in the other direction.
6. After the last packet, the side that started the close sits in a state called TIME_WAIT for a while, doing nothing. That looks like waste. It is not: it catches late packets from the connection that just ended.

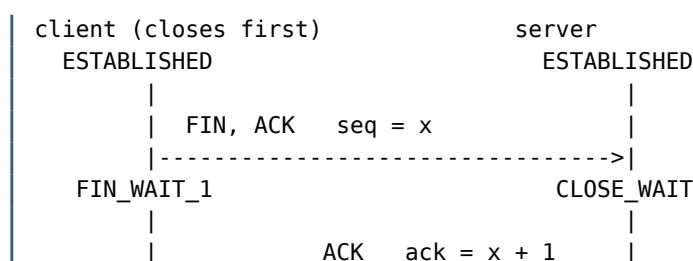
■ 28.5.2 PLAIN - a picture in your head

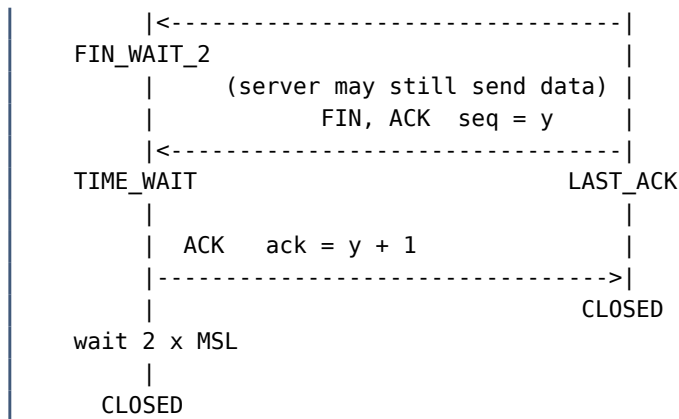
1. Think of a phone call between two polite people who both hate hanging up first.
2. Aisha says “that is all from me”. Ben says “understood, noted”. Ben may still talk, and Aisha listens. The call is half-closed.
3. But Aisha does not put the phone down immediately. She holds the line for a couple of minutes, because if her final “understood” was not heard, Ben will repeat his goodbye, and somebody has to answer it.

Where this comparison breaks: real people can tell a repeated goodbye from a new call. A computer reusing the same four numbers cannot, which is the real reason for the wait. It is not politeness, it is identity. The next connection using the same four-tuple must not receive a stray packet meant for the last one.

■ 28.5.3 PLAIN - a worked example

1. Here is the four-way close. The client closes first, so the client ends up waiting.

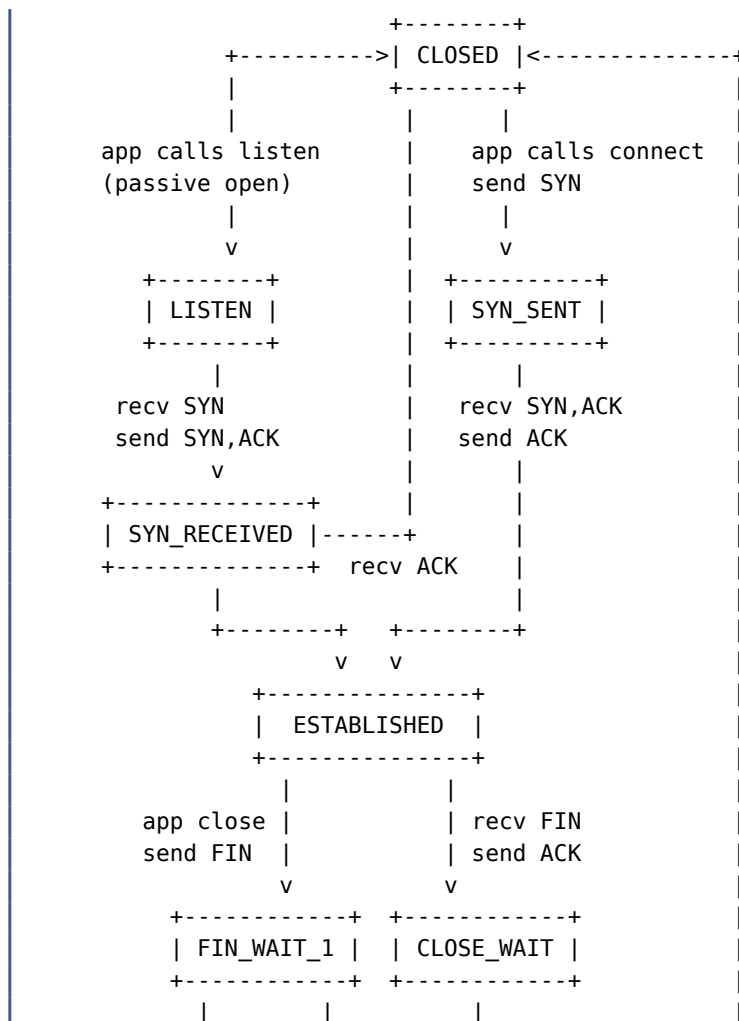


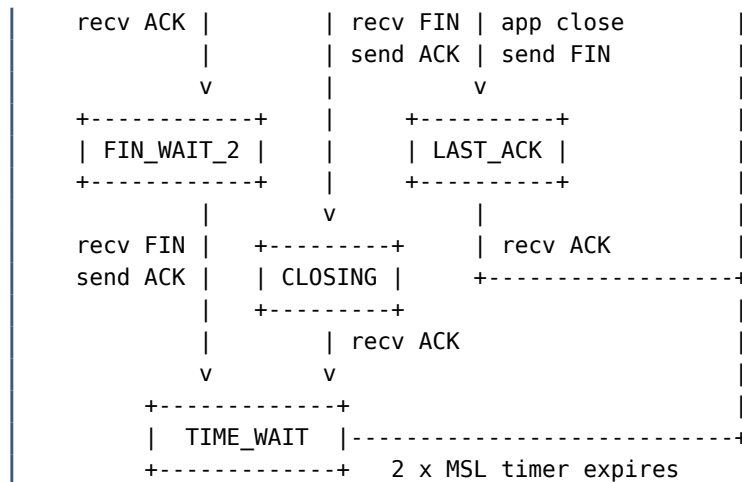


2. Count the packets: FIN, ACK, FIN, ACK. Four. Often the middle two are combined into one FIN-ACK, when the server has nothing left to send.
3. $2 \times \text{MSL}$ means twice the maximum segment lifetime. RFC 9293 sets MSL at 2 minutes, so the specified wait is 4 minutes.
4. Linux uses a fixed 60 seconds instead, compiled in as `TCP_TIMEWAIT_LEN`. That is an implementation detail, not a standard.

■ 28.5.4 PLAIN – what is really happening inside

1. Here is the full state machine. Read it as: you are in a box, an event happens, you move along an arrow.





2. SYN_SENT is a client that has sent its SYN and is waiting. This is exactly where the reader's connection was stuck.
3. SYN_RECEIVED is the server's half-open state between SYN-ACK and the final ACK. ESTABLISHED is the working state where everything useful happens.
4. FIN_WAIT_1 and FIN_WAIT_2 belong to the side that closed first, before and after its FIN is acknowledged. CLOSE_WAIT belongs to the side that was told first, and thousands of CLOSE_WAIT sockets is nearly always an application bug: the program never called close itself.

■ 28.5.5 TECHNICAL - the engineer's version

1. The eleven states are normative in RFC 9293 section 3.3.2. The names are the same strings printed by netstat and ss on every platform.
2. TIME_WAIT exists for two distinct reasons, and people usually remember only the first.
3. Reason one: to let any duplicate or delayed segment from this incarnation of the connection expire before the same four-tuple can be reused.
4. Reason two: to be able to retransmit the final ACK. If that ACK is lost, the peer in LAST_ACK resends its FIN and somebody must answer. Without TIME_WAIT the answer would be an RST, which the peer logs as an error.
5. Only the side performing the active close enters TIME_WAIT. Whoever calls close first pays the cost.
6. This is why servers accumulate TIME_WAIT. An HTTP server that closes each connection after responding, or a proxy that opens and closes short-lived back-end connections, is the active closer every time.

Item	Specified	Linux	macOS
MSL	2 minutes	not used directly	net.inet.tcp.msl
TIME_WAIT length	2 x MSL = 240 s	fixed 60 s	2 x msl
FIN_WAIT_2 timeout	none	60 s, tcp_fin_timeout	sysctl controlled

7. At 60 seconds of TIME_WAIT that caps sustained new connections at roughly 470 per second to a single destination. The fix is connection reuse, not tuning.
8. net.ipv4.tcp_tw_recycle was different, was unsafe behind NAT, and was removed from Linux in version 4.12, July 2017. Advice telling you to enable it is out of date and should be ignored.
9. Inspect with ss -tan state time-wait | wc -l on Linux, or netstat -an -p tcp | grep -c TIME_WAIT on macOS.

■ 28.5.6 WORDS – remember these

1. State — where a connection currently is — one of the eleven values in the RFC 9293 state machine.
2. ESTABLISHED — the working state — both directions open, data may flow.
3. TIME_WAIT — the pause after hanging up — the $2 \times \text{MSL}$ wait held by the active closer to absorb stray segments and resend the final ACK.
4. CLOSE_WAIT — waiting for your own program to hang up — the state after receiving a FIN, exited only when the application calls `close`.

28.6 Making TCP reliable and fast

■ 28.6.1 PLAIN – in simple words

1. TCP's reliability rests on one habit: number everything, and confirm what you received.
2. Every byte in the stream has a number. Not every packet, every byte.
3. The receiver keeps saying "I have everything up to number N". That message is the acknowledgement, or ACK.
4. The sender keeps a copy of everything sent but not yet confirmed, and if a confirmation does not arrive in time, sends it again.
5. Doing that one piece at a time would be safe but slow, so the sender is allowed to send many pieces before waiting for any confirmation. The amount allowed in flight at once is called the **window**, and as confirmations come in it slides forward, letting new data out.
6. The receiver controls the window size, so a fast sender cannot drown a slow receiver. That is **flow control**.

■ 28.6.2 PLAIN – a picture in your head

1. Think of a conveyor belt between a warehouse and a shop.
2. The shop tells the warehouse how many crates it has room for. Say twelve. The warehouse may put up to twelve crates on the belt without hearing back.
3. Each crate is numbered. The shop calls out "received everything up to 7", which frees the warehouse to put crates 13 to 19 on the belt.
4. If crate 5 falls off, the shop keeps calling "everything up to 4", no matter how many later crates arrive, and the warehouse works out that 5 is missing.

Where this comparison breaks: the numbering in TCP counts bytes, not crates, so a crate's number is the number of the first item inside it. Also the belt is not a single fixed-speed belt: the sender is guessing how fast the road can take traffic, which is section 28.7, and that guess is separate from the storeroom.

■ 28.6.3 PLAIN – a worked example

1. Suppose the sender's window is 5 segments of 1,460 bytes, starting at sequence number 1000, and segment 3 is lost.

sent:	seq 1000	1460 bytes	arrives
	seq 2460	1460 bytes	arrives
	seq 3920	1460 bytes	LOST
	seq 5380	1460 bytes	arrives
	seq 6840	1460 bytes	arrives
acks:	ack 2460		after first
	ack 3920		after second
	ack 3920	(duplicate)	after fourth

```

ack 3920    (duplicate)  after fifth
ack 3920    (duplicate)  after a sixth segment

```

- The receiver cannot say “I have 5380”, because that would claim everything below it, and 3920 is missing. So it repeats 3920. This is a **cumulative acknowledgement**: all bytes below this number are safely here.
- The sender sees the same acknowledgement three times over. Three duplicates is the agreed signal.
- It immediately re-sends just the segment at 3920, without waiting for any timer. That is **fast re-transmit**.
- The extra mechanism is **selective acknowledgement**, SACK. The receiver adds a note saying “I also have the block from 5380 to 8300”.
- With SACK the sender re-sends exactly one segment. Without it, up to three.

■ 28.6.4 PLAIN - what is really happening inside

- The sender keeps three markers over its send buffer: the oldest unconfirmed byte, the next byte to send, and the end of the window.
- The waiting time before declaring a loss is measured, not fixed. The sender keeps a smoothed average of the round-trip time and a measure of how much that time jumps around, then sets the timeout to average plus four times the jumpiness.
- On the receiving side, out-of-order segments are held in a reorder queue and handed to the application only once the gap is filled.
- If the application is slow to read, that queue fills and the advertised window shrinks. At zero the sender stops sending data entirely.
- The sender then sends a tiny **window probe** every so often, because if it simply waited for an update and that update were lost, both sides would wait forever.

■ 28.6.5 TECHNICAL - the engineer's version

- The retransmission timeout is computed per RFC 6298, June 2011, by Paxson, Allman, Chu and Sargent, with $\alpha = 1/8$, $\beta = 1/4$ and $K = 4$.

```

first measurement R:
  SRTT  = R
  RTTVAR = R / 2

later measurements R':
  RTTVAR = (1 - beta) * RTTVAR + beta * |SRTT - R'|
  SRTT   = (1 - alpha) * SRTT  + alpha * R'
  RT0    = SRTT + max(G, K * RTTVAR)

then clamp: RT0 = max(RT0, 1 second)

```

- Here is that running on a path of about 30 ms with one spike.

Sample	RTT ms	SRTT ms	RTO used
1	30.0	30.00	1000 ms, floored
2	32.0	30.25	1000 ms, floored
3	28.0	29.97	1000 ms, floored
4	95.0	38.10	1000 ms, floored
5	31.0	37.21	1000 ms, floored

3. Every computed value is well under one second, so the RFC 6298 minimum of 1 second governs. On short paths the timer is almost always the floor, which is why fast retransmit matters: waiting for the RTO would cost thirty round trips.
4. Fast retransmit and fast recovery are in RFC 5681, September 2009. The trigger is three duplicate ACKs, chosen because one or two duplicates are commonly caused by simple reordering.
5. Selective acknowledgement is RFC 2018, October 1996, by Mathis, Mahdavi, Floyd and Romanow. It is negotiated in the SYN and carries three or four blocks per segment because of the 40-byte option limit.
6. The receive window field is 16 bits, capping the raw value at 65,535 bytes. Window scaling, RFC 7323, September 2014, obsoleting RFC 1323, adds a shift count of up to 14, giving a maximum window just under 1 GiB. The scale factor is exchanged only in the SYN and SYN-ACK; if either side omits it, no scaling is used for the whole connection.
7. Zero window handling: the sender must send a one-octet probe on the persist timer, backing off exponentially, so a lost window update cannot deadlock the connection.
8. Nagle's algorithm is RFC 896, "Congestion Control in IP/TCP Internetworks", John Nagle, 6 January 1984. Rule: if unacknowledged data is outstanding, buffer small writes until an ACK arrives or a full segment can be sent.
9. Delayed acknowledgement is in RFC 1122, October 1989. An ACK may be delayed by less than 500 ms, and there must be an ACK for at least every second full-sized segment.
10. The two together create a well-known stall. The sender holds a small final piece under Nagle. The receiver holds its ACK under delayed ACK. Neither moves until the delayed-ACK timer fires.
11. On Linux that timer is adaptive between about 40 ms and 200 ms, from `TCP_DELACK_MIN` and `TCP_DELACK_MAX`. A protocol that writes a header and a body in two separate calls can add 40 ms per exchange for no reason.
12. The fixes, in order of preference: write the whole message in one call or use `writew`; or set `TCP_NODELAY` to disable Nagle. Disabling Nagle is normal for request-response traffic and is the default in most RPC libraries.
13. Inspect with `ss -tin` on Linux, which prints `rtt`, `rttvar`, `cwnd`, `ssthresh`, `retrans` and the negotiated window scale for every socket.

■ 28.6.6 WORDS – remember these

1. Sequence number — the count of bytes sent so far — a 32-bit field naming the first octet of this segment's payload.
2. Acknowledgement — a receipt for what arrived — a cumulative ACK naming the next octet expected, implying everything below it arrived.
3. SACK — a note saying which pieces beyond the gap arrived — the selective acknowledgement option of RFC 2018.
4. Fast retransmit — resending early because of repeated receipts — resending the missing segment after three duplicate ACKs, per RFC 5681.
5. RTO — how long to wait before assuming loss — the retransmission timeout of RFC 6298, `SRTT` plus four times `RTTVAR`, floored at 1 second.

28.7 Congestion control in plain words

■ 28.7.1 PLAIN – in simple words

1. Flow control stops you drowning the person you are talking to. Congestion control stops you drowning the road between you. Mixing them up is one of the most common mistakes in this subject.
2. The receiver tells you its buffer size, so flow control is easy. Nobody tells you the capacity of the network, so the sender has to guess and keep re-guessing.
3. The guess is called the **congestion window**, a second limit on how much may be in flight, kept alongside the receiver's window.
4. The strategy is simple: start small, increase while things go well, cut hard when a packet is lost.
5. Loss is the signal, because on a plain IP network loss is the only message the network sends about being full.
6. That works beautifully when loss means congestion, and terribly when loss means a bad radio link, because then TCP slows for no reason.

■ 28.7.2 PLAIN – a picture in your head

1. Imagine driving into an unfamiliar city at night with no speed signs.
2. You start slowly. Nothing bad happens, so you go faster. At some point you scrape a kerb. You brake hard, then creep back up.
3. That is congestion control: millions of senders, no central authority, all probing for a limit and backing off when they find it.
4. Now imagine your car reports a scrape whenever you drive over a bump. You crawl through an empty city, convinced it is full. That is TCP on a lossy wireless link.

Where this comparison breaks: a driver can see the road ahead. TCP cannot see anything. It learns about the road only by driving into it and noticing what happened one round trip later. Every decision is made with information that is already old.

■ 28.7.3 PLAIN – a worked example

1. Slow start is badly named. It is not slow; it is a doubling. Start with a congestion window of 10 segments, the modern default.

Round trip	Window, segments	Bytes in flight
1	10	14,600
2	20	29,200
3	40	58,400
4	80	116,800
5	160	233,600
6	320	467,200

2. Doubling continues until a loss happens or the window passes the slow start threshold, after which it grows by about one segment per round trip. So the shape is rocket, then crawl, then a cut on loss, then crawl again.
3. Now the bandwidth-delay product: how much data must be in flight to keep a link busy, which is rate multiplied by round-trip time.

Path	RTT	Rate	Must be in flight
Laptop to Pune edge	30 ms	100 Mbit/s	375 KB
India to US East	250 ms	100 Mbit/s	3.1 MB
India to US East	250 ms	1 Gbit/s	31 MB

4. The 30 millisecond figure is the right order for the reader's own path, which ended at Microsoft routers in Pune. The 250 millisecond figure is typical for India to the east coast of the United States. Both are approximate.
5. Without window scaling the maximum window is 65,535 bytes. Divide that by the round-trip time to get the ceiling on speed: about 17 megabits per second at 30 milliseconds, and about 2.1 megabits per second at 250 milliseconds.
6. A gigabit connection from India to a US server, without window scaling, delivers about two megabits per second. The link is not the problem. The window is.

■ 28.7.4 PLAIN – what is really happening inside

1. The sender keeps two numbers: `cwnd`, the congestion window, and `sssthresh`, the slow start threshold.
2. While `cwnd` is below `sssthresh`, the sender is in slow start and `cwnd` grows by one segment per segment acknowledged, which doubles it per round trip.
3. Above `sssthresh`, the sender is in congestion avoidance and `cwnd` grows by about one segment per round trip, however many acknowledgements arrive.
4. When three duplicate acknowledgements arrive, the sender halves `sssthresh`, sets `cwnd` near the new `sssthresh`, retransmits, and continues. That is fast recovery, a mild reaction.
5. When the retransmission timer fires instead, the reaction is severe: `cwnd` drops all the way back to one segment and slow start begins again. That is why fast retransmit was such an important invention.

■ 28.7.5 TECHNICAL – the engineer's version

1. The problem was not theoretical. In October 1986 the link between Lawrence Berkeley Laboratory and the University of California, Berkeley, a distance of about 400 yards, collapsed from 32 kbit/s to 40 bit/s.
2. Van Jacobson diagnosed it and published "Congestion Avoidance and Control" at ACM SIGCOMM in August 1988, introducing slow start, congestion avoidance and the RTT variance estimator.
3. 4.3BSD-Tahoe, 1988, shipped slow start, congestion avoidance and fast retransmit. 4.3BSD-Reno, 1990, added fast recovery. The algorithm names come from the release names.
4. The current standard is RFC 5681, "TCP Congestion Control", September 2009, with NewReno loss recovery in RFC 6582, April 2012. The initial window of 10 segments comes from RFC 6928, April 2013; before that it was 2 to 4.
5. CUBIC is the Linux default since kernel 2.6.19 in November 2006, from the 2008 work of Sangtae Ha, Injong Rhee and Lisong Xu. It is now standards track as RFC 9438, August 2023, obsoleting the informational RFC 8312.
6. CUBIC grows the window as a cubic function of the time since the last reduction, making it far more aggressive than Reno on high bandwidth-delay-product paths and roughly as polite on short ones.

7. BBR, from Google in 2016 by Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh and Van Jacobson, takes a different view: it estimates the bottleneck bandwidth and the minimum round-trip time and paces to that, rather than treating loss as the signal.
8. Established fact: BBR usually outperforms loss-based algorithms on paths with non-congestive loss and deep buffers. Active dispute: how fairly BBR shares capacity with CUBIC in shallow buffers. BBR version 3 was presented in 2023 and is being standardized in the IETF; it is a draft, not an RFC.
9. The Mathis equation, from the 1997 paper by Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi and Teunis Ott, gives the throughput ceiling of a loss-based algorithm.

$$\text{rate} \leq (\text{MSS} / \text{RTT}) * (\text{C} / \sqrt{p})$$

MSS = 1460 bytes, C = 1.22, p = loss probability

Loss rate	RTT 30 ms	RTT 250 ms
0.01 percent	47 Mbit/s	5.7 Mbit/s
0.1 percent	15 Mbit/s	1.8 Mbit/s
1 percent	4.8 Mbit/s	0.57 Mbit/s
5 percent	2.1 Mbit/s	0.26 Mbit/s

10. Read the bottom-right cell. One in twenty packets lost on a long path gives you 260 kbit/s regardless of how fat the pipe is. This is why “the link is 1 Gbit/s” tells you almost nothing on its own.
11. Inspect and change with `sysctl net.ipv4.tcp_congestion_control` on Linux, `ss -tin` to see `cwnd` and `ssthresh` per socket, and `sysctl net.inet.tcp.cc.available` on macOS and FreeBSD.

■ 28.7.6 WORDS – remember these

1. Congestion window — the sender’s guess at what the road can take — `cwnd`, a sender-side limit measured in segments or bytes.
2. Bandwidth-delay product — how much must be in the air to keep a pipe full — rate multiplied by round-trip time, in bytes.
3. CUBIC — today’s usual algorithm — the cubic-growth loss-based controller of RFC 9438, default on Linux since 2006.
4. BBR — the model-based alternative — a controller estimating bottleneck bandwidth and minimum RTT, from Google in 2016, still an IETF draft.

28.8 The three outcomes of a connection attempt

■ 28.8.1 PLAIN – in simple words

1. This section is the heart of the chapter. Everything before it was preparation.
2. When you try to open a TCP connection, exactly three things can happen. Not four. Three.
3. **Success.** The far side answers with SYN-ACK. You are connected, in about one round trip.
4. **Refused.** Something answers, but the answer is no. You get an error almost immediately, well under a second on any normal path.
5. **Dropped.** Nothing answers. Not a yes, not a no, not an error. Silence. Your machine tries again a few times and eventually gives up.
6. Here is the rule to carry with you: **a refusal is a reply. A drop is not.** So the speed of a failure tells you which kind it was, before you know anything else about the network.

7. Fast failure means something on the path is alive, received your packet, and decided against you. Slow silent failure means your packet, or the answer to it, disappeared without anyone saying so.
8. The reader's failure was the slow silent kind. That one fact rules out a whole category of explanations, which is what section 28.9 does.

■ 28.8.2 PLAIN – a picture in your head

1. You knock on a door.
2. Someone opens it and lets you in. That is success.
3. Someone shouts through the door “go away” and you leave immediately. That is refusal. Unpleasant, but fast, and it proves someone is home.
4. Nothing happens. You knock again. Nothing. Eventually you walk away. That is a drop.
5. Notice how much less you learned in case three. You do not know if the house is empty, if the bell is broken, if there is a wall you cannot see, or if someone inside is deliberately ignoring you.

Where this comparison breaks: with a real door you can look at the windows and check whether the lights are on. On a network you often cannot get a second opinion from the same vantage point, which is why testing from another network, as the reader did with mobile data, is worth more than any amount of retrying.

■ 28.8.3 PLAIN – a worked example

You send: SYN -----> ?

(a) SUCCESS

<----- SYN, ACK
result: connected in about one round trip

(b) REFUSED

<----- RST, ACK
or <----- ICMP dest unreachable
result: error returned in well under a second

(c) DROPPED

...nothing...
SYN -----> ...nothing...
SYN -----> ...nothing...
SYN -----> ...nothing...
result: timeout after tens of seconds

Outcome	What comes back	Time	Typical message
Success	SYN-ACK	1 round trip	connection opens
Refused	RST, or ICMP	under 1 second	Connection refused
Dropped	nothing at all	15 to 130 s	Connection timed out

1. Try this on your own machine to feel the difference in speed.

```
# (b) refused: nothing is listening on port 9 locally
nc -vz 127.0.0.1 9
# nc: connect to 127.0.0.1 port 9 failed: Connection refused
# returns instantly

# (c) dropped: a documentation address that goes nowhere
nc -vz -w 30 192.0.2.1 443
# nc: connect to 192.0.2.1 port 443 failed: Operation timed out
# returns only after the full wait
```

- The two commands differ only in the address. The difference in how long they take is the entire diagnostic.

■ 28.8.4 PLAIN – what is really happening inside

- In the refused case, a machine received your SYN, found no socket listening on that port, and replied with a TCP segment carrying the RST flag.
- RST means reset. It says: this connection does not exist and never will.
- Your kernel receives that RST while in SYN_SENT, moves the socket to CLOSED, and returns the error ECONNREFUSED to your program at once.
- A firewall can also produce a refusal, by sending an ICMP destination unreachable message, most often code 3 for port unreachable or code 13 for administratively prohibited.
- In the dropped case, no such packet exists. Your kernel sits in SYN_SENT with a retransmission timer running and nothing to react to.
- The timer fires, the SYN is sent again, and the timer is doubled. Then again, and doubled again. This is exponential backoff.
- After a fixed number of attempts, or a fixed total time, the kernel gives up and returns ETIMEDOUT.

■ 28.8.5 TECHNICAL – the engineer’s version

- Errors as your program sees them: ECONNREFUSED for a refusal, ETIMEDOUT for a drop, EHOSTUNREACH or ENETUNREACH when ICMP or the local routing table says the destination cannot be reached at all.
- RFC 9293 requires an RST in response to a SYN for a port with no listener, unless local policy says otherwise, with hardening rules in RFC 5961.
- The relevant ICMP type is 3, Destination Unreachable, from RFC 792. Code 3 is port unreachable; codes 9, 10 and 13 are the administratively prohibited family added by RFC 1812, June 1995; code 4 is fragmentation needed.
- Here is the Linux SYN retry schedule with the default `net.ipv4.tcp_syn_retries` of 6 and the RFC 6298 initial RTO of 1 second.

Attempt	Sent at	Then waits
SYN 1	0 s	1 s
SYN 2	1 s	2 s
SYN 3	3 s	4 s
SYN 4	7 s	8 s
SYN 5	15 s	16 s
SYN 6	31 s	32 s
SYN 7	63 s	64 s
give up	127 s	-

- The Linux kernel documentation states this exactly: six retries corresponds to 63 seconds until the last retransmission, and a final timeout at 127 seconds.
- The server side has its own counter, `net.ipv4.tcp_synack_retries`, default 5, giving 31 seconds to the last SYN-ACK retransmission and a final timeout at 63 seconds.
- macOS and the BSDs use a total-time cap instead of a retry count. The `sysctl` is `net.inet.tcp.keepinit`, default 75000 milliseconds, so a connection attempt is abandoned after 75 seconds.

System	Control	Default	Gives up after
Linux	tcp_syn_retries	6	127 s
macOS, BSD	tcp.keepinit	75000 ms	75 s
Windows	retransmit count	2, approx	about 21 s

- The Windows row is approximate, because the retransmission count and the 3 second initial timeout have changed across versions.
- So a 5 second timeout in a log may be the application's limit, not the kernel's, and tells you less than a full kernel-length timeout does.
- The mapping onto scanner vocabulary is exact: success is open, refusal is closed, and a drop is filtered. Section 28.12 returns to this.

■ 28.8.6 WORDS – remember these

- RST — the packet that means “no, and stop” — the TCP reset control bit, aborting a connection immediately.
- Connection refused — someone answered no — the ECONNREFUSED error, caused by an RST or an ICMP port unreachable.
- Connection timed out — nobody answered at all — the ETIMEDOUT error, caused by exhausting SYN retransmissions.
- Exponential backoff — waiting twice as long after each failure — the doubling of the retransmission timer, 1, 2, 4, 8 seconds and onward.
- Silent drop — a packet discarded with no notification — a filtering or forwarding decision that generates no ICMP and no RST.

28.9 What the reader's timeout actually proves

■ 28.9.1 PLAIN – in simple words

- Here is what the reader saw, exactly as recorded. `curl -v https://github.com` printed Trying 20.207.73.82:443... , then stopped for 15 seconds and failed with nothing coming back at all.
- No refusal. No error message from any router. No reset. Silence.
- Now let us extract every single thing that observation proves, and refuse to claim anything more.
- One. DNS worked.** The name `github.com` became the address 20.207.73.82, so the resolver at 1.1.1.1 was reachable and answered.
- Two. Routing worked at least part of the way.** The traceroute reached hop 12, `ae106-0.rwa04.pnq20.ntwk.msn.net`, at 104.44.20.42. Packets from this laptop were leaving the flat, crossing the ISP and entering Microsoft's network in Pune.
- Three. The TCP handshake never completed.** The connection never reached ESTABLISHED. It stayed in SYN_SENT until it was abandoned.
- Four. Nothing actively refused it.** No RST came back. If one had, `curl` would have said “Connection refused” in a fraction of a second.
- Five. No router complained either.** No ICMP destination unreachable arrived. If one had, `curl` would have failed quickly with a different message.
- Six. Therefore something swallowed a packet in silence.** Either the SYN never reached a machine willing to answer, or the SYN-ACK it produced never made it home. One of those two packets died without a word.
- That is the complete list. Everything else people usually say about a timeout is guessing.

■ 28.9.2 PLAIN – a picture in your head

1. You post a letter that needs a signature on delivery. Three weeks later you have no signature and no returned envelope.
2. Compare this with getting the envelope back stamped “no such address”. That stamp would tell you a sorting office read your address and rejected it.
3. But silence is not useless. It definitively rules out every story that would have produced a stamp.

Where this comparison breaks: with post you could telephone the recipient and ask. The network equivalent is testing from a different vantage point, and the reader did exactly that with mobile data. That is why the mobile test was the most valuable single action in the whole session.

■ 28.9.3 PLAIN – a worked example

1. Here is the evidence laid out as a ladder, from most certain to least.

Claim	Status	Basis
DNS resolved the name	proven	address was printed
Some path existed outward	proven	trace reached hop 12
Handshake did not complete	proven	curl timed out
Nothing sent an RST	proven	no refusal message
No ICMP error arrived	proven	no unreachable message
A packet was dropped silently	proven	follows from the above
The drop was after hop 12	not proven	see below
A named party blocked it	not proven	intent is invisible
The server was down	contradicted	mobile data worked

2. Note the line about hop 12. The traceroute used ICMP or UDP probes, not TCP packets to port 443, and networks routinely treat those differently. A device can forward ICMP happily and drop TCP SYNs to port 443, or the reverse.
3. So “the trace reached hop 12” proves that some packets from this laptop reached Pune. It does not prove that this SYN did.
4. The session record notes several utun interfaces on the macOS machine, which are tunnel interfaces created by VPN software.
5. A stale route pointing 20.207.73.82 into a utun interface whose tunnel is not up produces precisely this symptom: a SYN that goes nowhere and a silent timeout.
6. That is not a claim that this happened. It is a candidate the collected evidence cannot rule out, and one command would have settled it.

■ 28.9.4 PLAIN – what is really happening inside

1. Walk the SYN’s possible fates in order, from the laptop outward.
2. It could have been dropped by the local packet filter on macOS, or routed into a tunnel interface that leads nowhere because a VPN left a route behind.
3. It could have crossed the ISP’s access network and been dropped in the private core at hops 2 to 6, which are carrier addresses in the RFC 1918 ranges.
4. Or, and this is the one people forget, it could have arrived perfectly and produced a SYN-ACK that was dropped on the way back. The return path is not necessarily the mirror of the outbound path.
5. From the client’s chair, all of these look identical: silence for 15 seconds. That is the fundamental limitation, and no amount of retrying from the same chair fixes it.

■ 28.9.5 TECHNICAL – the engineer’s version

1. First, an honest note about the 15 seconds. The macOS kernel default is `net.inet.tcp.keepinit` at 75000 milliseconds, so the kernel would have kept trying for 75 seconds.
2. A stop at 15 seconds therefore came from something above the kernel: a `--connect-timeout` or `--max-time` flag, a timeout wrapper, or a tool default. It does not change the conclusion, but the number should be attributed correctly.
3. Second, the candidate causes, with the evidence that would separate them.

Candidate	Distinguishing test
Local filter or bad route	<code>tcpdump</code> on the wire interface
Stale VPN route via <code>utun</code>	<code>route get 20.207.73.82</code>
Port-based filtering	try 80 and 22 to same address
Address or prefix block	try another 20.x address
Routing black hole	TCP traceroute to port 443
Asymmetric return drop	test from a second network
MTU black hole	connects, then stalls later
Middlebox state exhaustion	intermittent, load dependent

4. Take the MTU case seriously, because it is the one that does not fit. A path MTU black hole occurs when a router must fragment, sends ICMP type 3 code 4 “fragmentation needed”, and that ICMP is filtered. Path MTU Discovery is RFC 1191, November 1990; the probing alternative is RFC 4821, March 2007.
5. But a SYN is about 60 bytes. It fits through any MTU on earth. An MTU black hole therefore produces a connection that completes and then hangs, usually at the TLS ClientHello or the first large response.
6. The reader’s symptom was the opposite: the connection never opened at all. So MTU is a poor fit here and should be ranked low. Naming it and then ruling it out is better practice than never mentioning it.
7. The single most informative missing command is a capture, which answers “did the SYN leave, and on which interface”.

```
sudo tcpdump -n -i any \
    'host 20.207.73.82 and tcp port 443'

# expected on a drop, repeated with growing gaps:
# 12:00:00.000 192.168.0.101.51514 > 20.207.73.82.443: [S]
# 12:00:01.001 192.168.0.101.51514 > 20.207.73.82.443: [S]
# 12:00:03.005 192.168.0.101.51514 > 20.207.73.82.443: [S]
# and nothing at all coming back
```

8. The second most informative is a TCP-based traceroute, because it probes with the same protocol and port that is failing.

```
sudo traceroute -T -p 443 20.207.73.82
sudo nmap -sS -Pn -p 443 --traceroute 20.207.73.82
```

9. Now the rigour about what is not proven, stated flatly.
10. The evidence does not identify where the drop happened. A silent drop leaves no marker anywhere in the client’s view.

11. The evidence does not prove intent. A drop can come from a deliberate policy, a misconfiguration, a failed line card, a full state table in a carrier-grade NAT, or a route withdrawn during a reconvergence.
12. The evidence does not name a responsible party, and this book will not name one. It is the reader's ISP and it is Microsoft's edge network, and neither fact establishes who dropped anything.
13. Nor does it establish that the condition was permanent. A single 15 second sample says nothing about five minutes later.
14. What the evidence does support, and supports strongly: a silent discard somewhere on that path in that moment, and not a server that was down.

■ 28.9.6 WORDS – remember these

1. Black hole — a place packets go and never return — a forwarding state that discards traffic without generating ICMP.
2. Asymmetric routing — the way back is not the way there — a path where forward and reverse traffic traverse different links.
3. MTU black hole — big packets vanish, small ones do not — a path where fragmentation-needed ICMP is filtered, breaking Path MTU Discovery.
4. Vantage point — where you are standing when you measure — the network location from which an observation was taken, which bounds what it can prove.

28.10 Why “the server is down” was ruled out

■ 28.10.1 PLAIN – in simple words

1. The first thing most people say when a site does not load is “the site must be down”. Here that explanation is wrong, on two independent grounds.
2. **Ground one: the failure had the wrong shape.** A server that is down usually still answers something. It answers badly, but it answers.
3. If the web program has crashed but the machine is running, the machine's network software receives your SYN, finds no listener, and sends back a reset. You get “Connection refused” in milliseconds.
4. If the machine itself is off, the last router on that network usually cannot find it and sends back a “host unreachable” message. Again, fast.
5. If the site is behind a load balancer whose machines are all unwell, the load balancer answers you. It completes the connection and sends an error page.
6. None of those look like fifteen seconds of nothing.
7. **Ground two: it worked elsewhere at the same time.** The same site loaded instantly on the reader's phone over mobile data. Same destination, same moment, different path.
8. If the destination were down, the phone would have failed too. So the problem was not at the destination. It was on one particular path.

■ 28.10.2 PLAIN – a picture in your head

1. You phone a shop and hear nothing at all. Not a ring, not an engaged tone, not a recorded message. Dead air for fifteen seconds.
2. A shop closed for the day still makes your phone ring. A shop out of business gives you “this number is not in service”, promptly.
3. Dead air is not a property of the shop. Dead air is a property of the line.
4. Then your friend, standing beside you, phones the same number on a different network and speaks to an assistant. Now you know. The shop is fine.

Where this comparison breaks: phone networks are far more consistent about generating tones than IP networks are about generating errors. Much of why network faults are hard is that the internet's equivalent of the engaged tone is optional, and is frequently switched off.

■ 28.10.3 PLAIN – a worked example

1. Here is the same user-visible failure under five different causes.

Cause	What comes back	How long
Web process crashed	TCP RST	milliseconds
Whole host powered off	ICMP host unreachable	under a second
Backends unhealthy	HTTP 502 or 503	fast, full reply
Overloaded, queueing	slow but complete reply	seconds, then data
Silent drop on path	nothing at all	until you give up

2. Only the last row matches what the reader saw.
3. A service the size of GitHub is fronted by many machines behind anycast addressing and load balancers. For every one of them to go silent at once, while a phone two feet away reached it instantly, is not a coherent story.

■ 28.10.4 PLAIN – what is really happening inside

1. Consider a server whose web software has died. The kernel is still running, still owns the IP address, and still runs the TCP state machine.
2. Your SYN arrives, the kernel looks for a socket in LISTEN on port 443, finds none, and generates an RST. RFC 9293 requires that.
3. Now consider a load balancer with no healthy backends. It accepted your TCP connection before checking anything, completes TLS, reads your request, finds nowhere to send it, and returns 502 Bad Gateway or 503 Service Unavailable.
4. In every case some device took ownership of your packet and produced a response. Silence means no device took ownership, which is a different category of fault, living in the network rather than in the service.

■ 28.10.5 TECHNICAL – the engineer's version

1. The general principle: **a service failure produces a response with the wrong content; a path failure produces no response.** Distinguishing them is the first branch of any network diagnosis.
2. Independent-vantage testing is the strongest cheap evidence available. The mobile-data test is a control experiment: it changes exactly one variable, the network path, and holds the destination constant.
3. The honest version: the mobile test is slightly weaker than it looks. Mobile data may resolve `github.com` to a different address entirely, since large services return different answers to different resolvers and regions.
4. To make it airtight, pin the same address from the other network.

```
curl -v --resolve github.com:443:20.207.73.82 https://github.com
nc -vz -w 10 20.207.73.82 443
```

5. There is one real exception, where a healthy service does go silent. A large provider under a volumetric attack may drop SYNs from an entire source prefix rather than answer them, which to the client looks like a broken path.

6. So the correct conclusion is not “the network operator is at fault”. It is “the destination was serving, and packets on this path were being discarded without notification”. Only the second claim is supported.

■ 28.10.6 WORDS – remember these

1. Service failure — the thing you wanted is broken but reachable — an error response generated by a machine that received your request.
2. Path failure — you cannot reach the thing at all — loss or misdelivery in the network, typically with no error generated.
3. Control experiment — change one thing and hold the rest fixed — testing the same destination and port from a second network path.

28.11 Reading curl -v line by line

■ 28.11.1 PLAIN – in simple words

1. curl fetches a URL. The -v flag makes it narrate what it is doing, naming each stage as it starts and finishes.
2. There are three kinds of line. A line beginning with * is curl talking about itself. Nothing on the network looks like that.
3. A line beginning with > is something curl sent to the server. A line beginning with < is something the server sent back, so the first < line is the first proof that the far side is really talking to you.
4. The reader’s run stopped after Trying 20.207.73.82:443..., the line curl prints just before it starts the TCP handshake. So the name lookup finished and the handshake never did. The output pinpoints the stage.

■ 28.11.2 PLAIN – a picture in your head

1. Think of curl’s verbose output as a running commentary from a courier.
2. “Looking up the address.” “Driving there.” “Arrived at the gate.” “Showing my papers.” “Papers accepted.” “Handing over the request.” “Receiving a reply.”
3. If the commentary stops after “driving there”, the courier never arrived, and you stop worrying about the papers. If it stops after “papers accepted”, the route and the gate were fine and the problem is later.

Where this comparison breaks: a courier reports its own progress honestly. curl only knows what the operating system tells it, and the operating system only knows what came back. Neither can report on a packet that vanished outbound. The commentary stopping is an absence of news, not news of a failure.

■ 28.11.3 PLAIN – a worked example

1. A successful run, trimmed, with the meaning of each line beside it.

* Trying 20.207.73.82:443...	starting TCP handshake
* Connected to github.com port 443	handshake done, ESTABLISHED
* ALPN: curl offers h2,http/1.1	offering HTTP versions in TLS
* TLSv1.3 (OUT), Client hello (1)	first TLS message sent
* TLSv1.3 (IN), Server hello (2)	first TLS message received
* subject: CN=github.com	the name on the certificate
* issuer: C=US; O=DigiCert Inc	who signed it
> GET / HTTP/2	request line sent
> Host: github.com	request header sent

```
< HTTP/2 200          first byte back from server
< content-type: text/html response header received
```

2. Now the reader's failing run, which is much shorter.

```
* Trying 20.207.73.82:443...
* connect to 20.207.73.82 port 443 failed: Operation timed out
* Failed to connect to github.com port 443 after 15002 ms
* Closing connection 0
curl: (28) Failed to connect to github.com port 443
```

3. Two facts jump out. There is no `Connected` to line, so the handshake never finished. And there is no `<` line anywhere, so nothing came back at all.

Code	Name	Means
6	Could not resolve host	DNS failed
7	Failed to connect	usually refused
28	Operation timed out	silence
35	SSL connect error	TLS stage failed
60	Certificate problem	trust or name failed

4. Code 7 and code 28 are the refusal and the drop from section 28.8, wearing curl's clothing.

■ 28.11.4 PLAIN - what is really happening inside

1. `curl -v` does not measure time. For numbers you use `-w`, the write-out option, which prints variables after the transfer.

```
curl -sS -o /dev/null -w '
  dns      %{time_namelookup}
  tcp      %{time_connect}
  tls      %{time_appconnect}
  ttfb     %{time_starttransfer}
  total    %{time_total}
  address  %{remote_ip}:%{remote_port}
' https://github.com
```

2. Every time value is measured from the start of the request, so they are cumulative, not durations. You subtract to get each stage.

Field	Example	Stage duration
<code>time_namelookup</code>	0.021 s	DNS took 21 ms
<code>time_connect</code>	0.055 s	TCP took 34 ms
<code>time_appconnect</code>	0.128 s	TLS took 73 ms
<code>time_starttransfer</code>	0.291 s	server took 163 ms
<code>time_total</code>	0.354 s	body took 63 ms

3. In the reader's case `time_namelookup` would be small and every later field would read zero. A resolved address with a `time_connect` of zero is the machine-readable signature of the drop case.

■ 28.11.5 TECHNICAL – the engineer’s version

1. `--connect-timeout` limits only the setup phase, which curl defines as DNS plus the TCP, TLS or QUIC handshakes. The libcurl default for `CURLOPT_CONNECTTIMEOUT` is 300 seconds.
2. `--max-time` limits the whole operation including the body. Use both: a short connect timeout and a longer overall one.
3. `-4` forces IPv4 and `-6` forces IPv6. This matters constantly on dual-stack networks, where a broken IPv6 path produces long stalls. The reader’s machine reported IPv6: (none), so `-4` would have changed nothing here.
4. `--resolve host:port:address` overrides DNS for one name and port, letting you test one specific server without changing any system configuration.
5. Here are the exact commands the reader should have run.

```
# 1. is it TCP, or is it TLS or HTTP above it
nc -vz -w 5 20.207.73.82 443

# 2. is it this port, or this address
nc -vz -w 5 20.207.73.82 80
nc -vz -w 5 140.82.121.4 443

# 3. is a name problem hiding as a network problem
dig +short github.com @1.1.1.1
curl -v --resolve github.com:443:20.207.73.82 https://github.com

# 4. did the SYN leave the laptop, and by which interface
sudo tcpdump -n -i en0 'host 20.207.73.82 and tcp port 443'
route get 20.207.73.82

# 5. how far does a TCP SYN to 443 actually get
sudo nmap -sS -Pn -p 443 --reason --traceroute 20.207.73.82

# 6. timing signature, in one line
curl -sS -o /dev/null -m 20 \
  -w 'dns %{time_namelookup} tcp %{time_connect}\n' \
  https://github.com
```

6. Command 1 separates the transport from everything above it: if nc also hangs, the fault is at or below TCP. Command 2 distinguishes a port-specific filter from an address-specific one, for ten seconds of effort.
7. Command 4 is the only one that can prove the packet left the machine, which is the assumption every other test quietly makes. Command 5 probes with a SYN to port 443, so unlike an ICMP traceroute it measures the failing path.

■ 28.11.6 WORDS – remember these

1. Write-out — print measured values after the transfer — curl’s `-w` option with variables such as `time_connect`.
2. Resolve override — pin a name to an address for one command — curl’s `--resolve host:port:address`.
3. Exit code — the number the command returns — curl’s documented failure codes, 7 for connect failure and 28 for timeout.

28.12 The other tools that answer this question

■ 28.12.1 PLAIN – in simple words

1. `curl` tests a whole stack at once: DNS, TCP, TLS and HTTP. When it fails you still have to work out which layer failed.
2. `nc`, usually called netcat, opens a bare TCP connection and reports whether it opened. Nothing above TCP is involved.
3. `telnet` did the same job for decades, and old documentation still recommends it. It works, but modern systems are removing it.
4. `nmap` tries many ports at once and classifies each. Its vocabulary maps exactly onto the three outcomes of section 28.8.
5. `tcpdump` shows the actual packets on the wire, and is the only tool here that can prove a packet left your machine.

■ 28.12.2 PLAIN – a picture in your head

1. Think of a doctor with four instruments.
2. `curl` is asking the patient how they feel: rich information, hard to localize. `nc` is a single specific test: does this one joint bend.
3. `nmap` is a broad screen: check twenty things quickly and flag the odd ones.
4. `tcpdump` is the X-ray: it shows what is actually there rather than what anyone reports. You would not start with it, and you would not finish without it when the reports disagree.

Where this comparison breaks: an X-ray sees inside the patient. `tcpdump` only sees packets passing your own machine's interfaces. It can prove your SYN left; it can never show you what a router 2,000 kilometres away did with it.

■ 28.12.3 PLAIN – a worked example

```
# netcat, bare TCP connect, 5 second limit
nc -vz -w 5 20.207.73.82 443
# open:      Connection to 20.207.73.82 port 443 succeeded
# closed:    nc: connect to ... failed: Connection refused
# filtered:  nc: connect to ... failed: Operation timed out

# nmap, with the reason printed
sudo nmap -sS -Pn -p 443 --reason 20.207.73.82
# PORT      STATE      SERVICE  REASON
# 443/tcp    open       https     syn-ack ttl 52
# 443/tcp    closed     https     reset ttl 52
# 443/tcp    filtered   https     no-response
```

1. `syn-ack` means a handshake reply arrived. `reset` means an RST arrived. `no-response` means nothing arrived. That is section 28.8, word for word.

Outcome	nmap state	nmap reason
Success	open	syn-ack
Refused	closed	reset or ICMP
Dropped	filtered	no-response

■ 28.12.4 PLAIN – what is really happening inside

1. `nc -z` and `telnet` perform a normal connect call through the operating system, so they see exactly what any ordinary program would see.
2. `nmap -sS` builds raw packets itself, sends a SYN, reads the reply, and never completes the handshake. That is why it needs administrator rights and is called a half-open or stealth scan. `nmap -sT` uses connect instead.
3. `tcpdump` asks the kernel for a copy of packets at the interface, so it shows outbound packets even when nothing comes back. A capture showing three or four outgoing SYNs with growing gaps and nothing inbound is direct proof of the drop case.

■ 28.12.5 TECHNICAL – the engineer’s version

1. Telnet is RFC 854, May 1983, by Jon Postel and Joyce Reynolds. Apple removed the `telnet` and `ftp` clients from macOS in version 10.13 High Sierra, September 2017. On Windows the Telnet Client has been an optional feature, off by default, since Windows Vista. It is unencrypted and must never be used for real remote login; as a reachability probe `nc` is better in every way.
2. Nmap’s authoritative state definitions: `open` means an application is actively accepting connections; `closed` means the port is reachable but no application is listening; `filtered` means packet filtering prevents probes from reaching the port, so `nmap` cannot determine the state.
3. Always pass `-Pn`. Without it, a host that ignores pings is recorded as down and never scanned at all, which yields a misleading empty result.

```
# only SYN packets, either direction, to one host
sudo tcpdump -n 'host 20.207.73.82 and tcp[tcpflags] & tcp-syn != 0'

# only resets, useful for spotting refusals
sudo tcpdump -n 'tcp[tcpflags] & tcp-rst != 0'

# all ICMP, for unreachable and MTU messages
sudo tcpdump -n icmp

# write a file for later analysis in Wireshark
sudo tcpdump -n -w capture.pcap 'host 20.207.73.82'
```

4. Complementary socket-state tools: `ss -tan state syn-sent` on Linux shows every connection stuck in exactly the reader’s state; on macOS use `netstat -an -p tcp` and look for `SYN_SENT`. `lsof -i -P -n` maps sockets to process names on both.

■ 28.12.6 WORDS – remember these

1. `netcat` — a bare TCP connect tester — `nc`, opening a socket and reporting the result without any higher protocol.
2. `Filtered` — `nmap`’s word for silence — a port state where probes get no response, matching the dropped outcome.
3. Packet capture — recording the actual packets — `tcpdump` or Wireshark reading frames from an interface, selected with a filter such as `tcp port 443`.

28.13 Sockets from the programmer’s side

■ 28.13.1 PLAIN – in simple words

1. Everything in this chapter is reached from a program through about eight function calls, with the same names in almost every language.

2. `socket` creates an endpoint and returns a small number, a handle. `bind` attaches that endpoint to a local address and port. Servers do this; clients usually skip it and let the system choose.
3. `listen` turns a server endpoint into a door and sets how many callers may queue. `accept` takes one waiting connection off that queue and gives you a new handle for that one client.
4. `connect` is the client's call. It sends the SYN and waits for the handshake. This is the call that took 15 seconds and failed for the reader.
5. `send` and `recv` move bytes. On a stream socket they may move fewer bytes than you asked for, and you must handle that.
6. `close` releases the handle and starts the four-way close. And separately you set a timeout, because without one your program inherits the operating system's very long default.

■ 28.13.2 PLAIN – a picture in your head

1. `socket` is renting a telephone. `bind` is being assigned an extension number.
2. `listen` is telling the switchboard to put callers through, with a queue of at most so many people on hold. `accept` is picking up the next caller from that queue; the main line stays free for the next one.
3. `connect` is dialling out. `send` and `recv` are talking and listening, and `close` is the polite mutual goodbye of section 28.5.

Where this comparison breaks: a phone conversation preserves the boundaries between sentences. A TCP stream does not. If you send three times, the other side may `recv` all of it in one go, or in seven pieces. Every real protocol therefore carries its own length prefix or delimiter.

■ 28.13.3 PLAIN – a worked example

1. A minimal server, in C, that echoes back whatever it is sent.

```
int s = socket(AF_INET, SOCK_STREAM, 0);

struct sockaddr_in a;
memset(&a, 0, sizeof a);
a.sin_family      = AF_INET;
a.sin_addr.s_addr = htonl(INADDR_ANY);
a.sin_port        = htons(8080);

int one = 1;
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &one, sizeof one);

bind(s, (struct sockaddr *)&a, sizeof a);
listen(s, 128);          /* 128 = backlog queue length */

for (;;) {
    int c = accept(s, NULL, NULL); /* one client */
    char buf[1024];
    ssize_t n = recv(c, buf, sizeof buf, 0);
    if (n > 0) send(c, buf, (size_t)n, 0);
    close(c);          /* this side does the active close */
}
```

2. A client with a timeout, in Python, that reports which of the three outcomes happened.

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(5.0)          # applies to connect and to I/O
```

```

try:
    s.connect(("20.207.73.82", 443))
    print("SUCCESS: handshake completed")
except ConnectionRefusedError:
    print("REFUSED: an RST came back, something said no")
except socket.timeout:
    print("DROPPED: nothing came back at all")
except OSError as e:
    print("OTHER:", e)
finally:
    s.close()

```

3. Those three branches are exactly the three outcomes of section 28.8, in twelve lines of code.

■ 28.13.4 PLAIN – what is really happening inside

1. `socket` allocates a kernel structure and a file descriptor, and `bind` writes a local address and port into it. Neither sends any packet.
2. `listen` moves the socket into LISTEN state and creates two queues: one for half-open connections awaiting the final ACK, one for completed connections awaiting `accept`, which hands one over as a new descriptor.
3. The number passed to `listen` is the backlog. If the completed queue is full, new connections are dropped or reset, and clients see a timeout or a refusal.
4. `connect` builds and sends the SYN, puts the socket in SYN_SENT, and sleeps until the SYN-ACK arrives, an RST arrives, or the timer runs out.
5. `send` copies your bytes into the kernel send buffer and returns without waiting, so a successful `send` proves nothing about delivery. `recv` copies out of the receive buffer and returns 0 exactly when the peer has sent a FIN.
6. `close` releases the descriptor and begins the FIN exchange. `shutdown` is the finer tool: it closes one direction while leaving the other open.

■ 28.13.5 TECHNICAL – the engineer's version

1. The Berkeley sockets interface first shipped in 4.2BSD in August 1983 and is standardized in POSIX.1-2001. That is why network code looks the same in C, Python, Go, Java and Rust.
2. Timeouts are the part people get wrong. `SO_RCVTIMEO` and `SO_SNDTIMEO` apply to data operations. On Linux they do not reliably bound `connect`.
3. The portable way to bound `connect` is to set the socket non-blocking, call `connect`, expect EINPROGRESS, wait on `select` or `poll` for writability with your own deadline, then read `SO_ERROR` for the result.
4. Linux additionally offers `TCP_USER_TIMEOUT` for established connections and `TCP_SYNCNT` to override `tcp_syn_retries` per socket, which is the clean way to make one connection give up in 5 seconds instead of 127.

Option	Level	Effect
<code>SO_REUSEADDR</code>	<code>SOL_SOCKET</code>	bind despite <code>TIME_WAIT</code>
<code>SO_KEEPALIVE</code>	<code>SOL_SOCKET</code>	probe idle connections
<code>TCP_NODELAY</code>	<code>IPPROTO_TCP</code>	disable Nagle
<code>TCP_SYNCNT</code>	<code>IPPROTO_TCP</code>	SYN retries, Linux only

5. A successful return from `send` means the data is in your kernel's buffer. It does not mean it left the machine, arrived, or was read by the far end.
6. This is the same principle the reader met in `git`: a `disconnect` while reading a response tells you nothing about whether the write landed. The only way to know is to ask again.

■ 28.13.6 WORDS – remember these

1. `EINPROGRESS` — “started, not finished” — the error returned by `connect` on a non-blocking socket, the basis of a custom connect timeout.
2. `shutdown` — close one direction only — the call that sends a FIN without releasing the descriptor.
3. `TCP_NODELAY` — send small writes at once — the socket option that disables Nagle's algorithm, standard in request-response systems.

28.98 Common wrong ideas

1. Wrong: a timeout means the server is down. Right: a timeout means nothing came back. A down server usually sends a reset or produces a fast error page. Silence points at the path, not at the service.
2. Wrong: if `ping` works, the site should work. Right: `ping` uses ICMP, not TCP. A device can forward ICMP and drop TCP port 443, and many do. `ping` failing also proves nothing, because ICMP is often blocked on purpose.
3. Wrong: TCP guarantees delivery. Right: TCP guarantees that data arrives complete and in order, or that the connection fails and you are told. It turns silent loss into correct delivery or a reported error. It cannot deliver across a cut cable.
4. Wrong: `closed` and `filtered` are the same thing. Right: `closed` means something replied with a reset, which proves the host is reachable. `filtered` means nothing replied, which proves almost nothing. They differ in evidence and in how long they take.
5. Wrong: UDP is just a worse TCP. Right: UDP is TCP with the reliability removed on purpose, so applications that cannot use late data are not forced to wait for it. Live voice, video calls, DNS and NTP all want it.
6. Wrong: thousands of `TIME_WAIT` sockets are a leak. Right: `TIME_WAIT` is normal for whoever closes first, and lasts 60 seconds on Linux. Thousands of `CLOSE_WAIT` sockets, by contrast, usually are a real application bug.
7. Wrong: * * * at the end of a traceroute proves the fault is at that hop. Right: many routers deprioritise or suppress ICMP time-exceeded replies. Silence at the end of a trace is the normal case, not a diagnosis.
8. Wrong: a successful `send` means the other side received the data. Right: it means the bytes are in your own kernel's buffer. Delivery is confirmed only by an acknowledgement you cannot see, or by a reply from the application.
9. Wrong: raising the timeout will fix a connection that times out. Right: a longer timeout only makes you wait longer for the same silence. It helps only when the path is slow, not when it is discarding packets.

28.99 Chapter summary in 20 lines

1. IP promises only to try. It gives no delivery, no ordering, no duplicate suppression, and no notification when a packet is lost.
2. TCP turns that into a reliable, ordered byte stream, and UDP is the layer that deliberately does not.

3. UDP is not a worse TCP. It is right whenever late data is worthless, such as live voice, live video, DNS queries and clock synchronization.
4. A port is a 16-bit number, 0 to 65,535, naming which program a packet is for.
5. A connection is identified by a four-tuple: source IP, source port, destination IP, destination port. That is why one server port serves millions.
6. A listening socket is matched on two values; a connected socket is matched on all four. Confusing them is the root of many bugs.
7. Port 443 is HTTPS, and it became the universal port because outbound 443 is nearly always permitted and its contents cannot easily be inspected.
8. Opening a connection costs a three-way handshake, SYN, SYN-ACK, ACK, and one full round trip before any application byte moves.
9. Initial sequence numbers are chosen unpredictably, per RFC 6528, because predictable ones allow blind injection attacks.
10. TCP is defined today by RFC 9293, STD 7, August 2022, which replaced the original RFC 793 of September 1981.
11. A connection walks eleven states. Closing takes four packets because each direction closes separately, and the active closer waits in TIME_WAIT.
12. Reliability comes from numbering every byte, cumulative acknowledgements, selective acknowledgement, retransmission timers, and fast retransmit on three duplicate acknowledgements.
13. Speed comes from the sliding window; the receiver's window is flow control and the sender's congestion window is congestion control. They are different.
14. Without window scaling, 65,535 bytes over a 250 millisecond path caps you at about 2 megabits per second regardless of link speed.
15. Loss-based congestion control treats every loss as congestion, so one percent loss on a long path cuts throughput to a fraction of a megabit per second.
16. A connection attempt has exactly three endings: SYN-ACK and success, RST or ICMP and a fast refusal, or silence and a slow timeout.
17. The diagnostic rule: fast failure means something answered; slow silent failure means something swallowed it.
18. The reader's `curl -v https://github.com` printed `Trying 20.207.73.82:443` and returned nothing for 15 seconds, proving DNS worked, some path existed as far as hop 12, the handshake never completed, and nothing refused it.
19. It does not prove where the drop happened, who caused it, or that it was deliberate. It does contradict "the server is down", because the same site loaded instantly on mobile data at the same moment.
20. `nc` isolates the transport, `nmap` names the outcome as open, closed or filtered, and `tcpdump` is the only tool that can prove the SYN ever left the machine.

Why Mobile Data Worked, and the Diagnostic Method

29.0 What this chapter gives you

1. You will be able to state the difference between “the site is down” and “I cannot reach the site”, and say who is affected by each.
2. You will be able to give a one-line test that separates those two claims, and explain why one success anywhere beats a hundred failures here.
3. You will be able to list every single thing that changed when the reader’s phone switched from home broadband to mobile data, and there are more than a dozen.
4. You will be able to explain why that test proved the destination was alive and yet proved nothing at all about which difference mattered.
5. You will be able to run a controlled experiment on a network: one variable at a time, with a stated hypothesis and a stated eliminating power.
6. You will be able to climb a seven-rung diagnostic ladder from “is my cable plugged in” to “where exactly does the path stop”, and say for each rung what a pass proves, what a pass does not prove, and what a failure means.
7. You will be able to map the reader’s own real fault onto that ladder and name the small set of causes that survive the evidence.
8. You will be able to use ping, traceroute, dig, nc, curl, mtr and a dozen other macOS commands with the flags that matter, and know what each one can and cannot prove.
9. You will be able to write up a fault so that an internet provider acts on it instead of asking you to reboot your router.
10. You will be able to catch an intermittent fault with a logging script, and tell a regional outage from a personal one.

29.1 “The site is down” and “I cannot reach the site”

■ 29.1.1 PLAIN – in simple words

1. These two sentences sound like the same complaint. They are not.
2. “The site is down” is a claim about the whole world. It says nobody can use it. Not you, not your friend, not a person in another country.
3. “I cannot reach the site” is a claim about you. It says that from where you are sitting, right now, with this machine and this connection, it fails.
4. The second claim is small and almost always true when you say it. You did try. It did fail. There is nothing to argue about.

5. The first claim is enormous. To be true, it has to hold for every person on the planet at the same moment.
6. You almost never have the evidence for the first claim. You only ever have evidence for the second.
7. So when you say “GitHub is down” after one failed page load, you have quietly upgraded a small true statement into a huge unproven one.
8. That upgrade is what wastes the hours. Once you believe the site is down, you stop testing. Waiting feels like the correct action.
9. And if the real fault is on your own path, waiting fixes nothing, because nobody at the other end knows anything is wrong.
10. The whole method in this chapter is built to stop you making that jump.

■ 29.1.2 PLAIN – a picture in your head

1. Imagine you post a letter to a friend in another city, and no reply comes.
2. You could say “my friend has moved away”. That is a claim about your friend.
3. Or you could say “my letter did not get a reply”. That is a claim about one letter, on one day, through one postal route.
4. The second is what you actually know. The first is a guess dressed up as knowledge.
5. There are many ways to get silence. The friend moved. The friend is ill. The letter was lost. The postbox at the end of your street is broken. A sorting office is on strike. The address on the envelope was fine but a customs check quietly binned it.
6. Now here is the trick that settles it in one move. You ask a person in another city to post the same letter to the same address.
7. If they get a reply, your friend has not moved. That is now certain. The problem is somewhere on your route, not at the far end.
8. You still do not know which part of your route. But you have cut the space of possible causes roughly in half, with one action.

Where this comparison breaks: letters are slow and rare, so you only get to test once or twice. Network tests are cheap and instant, so you can run twenty of them in a minute and narrow the cause much further than a letter ever could. Also, a letter has one route. A network packet may take a different route on every attempt, which is why an intermittent fault is so much harder than a missing friend.

■ 29.1.3 PLAIN – a worked example

1. Here is the reader’s real situation, in the two forms.
2. Weak form, the one that wastes time: “GitHub is down.”
3. Strong form, the one that leads somewhere: “From my Mac, on my home broadband in India, curl to github.com on port 443 gets no response of any kind and gives up after 15 seconds.”
4. The strong form names the machine, the network, the destination, the port, the tool, the symptom and the time. Every one of those is a variable you can change to learn something.
5. Now the one-line test that separates the two claims.

```
# Run this from a network that shares nothing with yours.
# Mobile data on a phone, or a friend's connection.
curl -sS -o /dev/null -w '%{http_code}\n' \
    --max-time 10 https://github.com
```

6. If that prints 200, the site is not down. That is settled, permanently, by a single observation. No amount of failing from your desk can undo it.

7. If it also hangs and prints nothing, then the site being down has become a live possibility, and now a status page is worth reading.
8. The reader did exactly this test, in the simplest possible way. The same phone, switched from home Wi-Fi to mobile data, loaded `github.com` instantly.
9. So the claim “GitHub is down” was dead within about ten seconds. Everything after that was about finding the reader’s own path problem.

■ 29.1.4 PLAIN - what is really happening inside

1. There is a piece of plain logic underneath this, and it is worth learning once, because it applies far beyond networks.
2. “The site is down” means “for all people, the site fails”. That is called a universal claim. It talks about everybody.
3. “I cannot reach the site” means “there exists at least one person, me, for whom it fails”. That is called an existential claim. It talks about one case.
4. To prove a universal claim you would have to check everybody. You cannot.
5. To destroy a universal claim you need one counterexample. One person who can reach it. That is cheap, and you can do it in ten seconds.
6. To prove an existential claim you need one example. Your own failure is that example. So your own claim is already proved.
7. This gives an unfair-looking rule that is completely correct: a single success anywhere beats any number of failures in one place.
8. It also gives the reverse warning: no number of failures from your desk ever proves the site is down. Not ten, not a thousand. They are all the same observation repeated.
9. So the cheapest useful action in any connectivity fault is to buy one counterexample from a different vantage point.
10. A vantage point is just a place to test from: another network, another machine, another country, another provider.

■ 29.1.5 TECHNICAL - the engineer’s version

1. The distinction maps onto a standard split in incident work: a service-side fault versus a client-side or path-side fault.
2. A service-side fault has a blast radius covering all or a defined subset of users, is visible to the operator’s own monitoring, and appears on a status page. Only the operator can fix it.
3. A path-side fault has a blast radius bounded by a network prefix, an autonomous system, a region or a single subscriber line. The operator’s monitoring usually shows nothing at all.
4. This asymmetry is why large operators publish per-region status, and why “all systems operational” on a status page is compatible with your total inability to reach the service.
5. The logic in block 4 is Karl Popper’s asymmetry between verification and falsification, set out in *Logik der Forschung* in 1934 and in English as *The Logic of Scientific Discovery* in 1959. Universal statements cannot be verified by observation but can be falsified by one counterexample.
6. The operational form of that rule is the single-vantage-point problem. One observer cannot distinguish global outage from local unreachability.
7. Distributed measurement platforms exist precisely to fix this. RIPE Atlas, launched by the RIPE Network Coordination Centre in 2010, runs a global fleet of small hardware and software probes, on the order of ten thousand active at any time, that anyone can task with ping, traceroute, DNS and TLS measurements from many countries at once.
8. Commercial equivalents include ThousandEyes, Catchpoint and Pingdom. Free web-based checkers do a weaker version of the same thing.

9. Terms to use precisely when reporting:

Phrase	What it actually claims
Service is down	Fails for all users
Partial outage	Fails for a named subset
Unreachable from AS X	Fails from one network
Degraded	Works, but slow or lossy
Cannot reproduce	Works from the operator

10. The verb matters. “Down” is a statement about the server. “Unreachable” is a statement about a pair: a source and a destination. Reachability is always a property of a pair, never of a single host.
11. That single sentence, that reachability is a property of a pair, is the most useful idea in this chapter.

■ 29.1.6 WORDS – remember these

1. Down — nobody can use it — a service-side failure affecting all or a defined subset of users, visible in the operator’s own monitoring.
2. Unreachable — I cannot get to it from here — a property of a source and destination pair, not of the destination alone.
3. Vantage point — a place you test from — an independent measurement source with its own address, provider and path.
4. Universal claim — a statement about everybody — falsifiable by a single counterexample, not verifiable by observation.
5. Existential claim — a statement about at least one case — proved by a single example, which your own failed attempt already supplies.
6. Blast radius — how many people a fault hits — the set of users, prefixes or regions affected by a given failure.

29.2 Everything that changed when the phone used mobile data

■ 29.2.1 PLAIN – in simple words

1. The reader’s test was simple. Same phone. Turn off Wi-Fi. Turn on mobile data. Load `github.com`. It loaded instantly.
2. It is tempting to read that as “so the problem is my Wi-Fi” or “so the problem is my router”. Both may be wrong.
3. Switching to mobile data does not change one thing. It changes almost everything between you and the destination.
4. Here is the honest list of what changed, and there are more items than most people expect.
5. The company carrying your traffic changed. Home broadband company out, mobile company in.
6. Your address on the internet changed. Every reply now comes back to a different number.
7. The network that owns that address changed, which means the route the reply takes back to you is completely different too.
8. The physical path changed. Different cables, different cities, different handover point into the destination’s network.
9. The name lookup probably changed. The phone on mobile data usually asks the mobile company’s name server, not the one your Mac was using.
10. Because the lookup changed, the address you got back may have changed, so you may have talked to a different machine at the far end entirely.

11. The boxes in the middle changed. Home router, home firewall and your provider's inspection equipment are all out of the picture. The mobile company's own equipment is in it.
12. The kind of address may have changed. Mobile networks in India commonly give you a modern address type that home broadband often does not.
13. And a small one that matters more than people think: time passed. The second test happened minutes after the first.
14. Only one of those needs to be the cause. The test cannot tell you which.

■ 29.2.2 PLAIN – a picture in your head

1. Imagine your parcel never arrives, so you try a different courier.
2. The second courier delivers it in an hour. You now know the address is real and somebody is there to sign for it.
3. That is genuinely valuable. It ends the argument about whether the recipient exists.
4. But look at what else changed when you switched courier. A different van, a different driver, a different depot, a different route, a different customs agent, a different set of rules about what they will carry, and a different day.
5. If someone now says “so the first courier's van was broken”, they have invented an explanation. The van is one of seven things that changed.
6. Maybe the first courier refuses parcels over a certain size. Maybe their depot in one city is on strike. Maybe their scanner rejects your postcode.
7. You cannot tell. You changed everything at once, so the experiment has no resolving power about the cause.
8. It has enormous power about one specific question, and that question is “does the recipient exist”. That question is now closed.

Where this comparison breaks: couriers deliver or fail visibly, and you get a tracking page. A network path can fail silently, with no message to anybody, and neither end learns that a packet was dropped. The far end in the reader's case never knew a connection had been attempted at all.

■ 29.2.3 PLAIN – a worked example

1. Let us do the enumeration properly for the reader's case, item by item.
2. On home broadband, the Mac had a private address behind the router at 192.168.0.1, and used the public name server 1.1.1.1.
3. github.com resolved to 20.207.73.82, which is an address owned by Microsoft, the company that has owned GitHub since 2018.
4. The connection attempt to that address on port 443 got no reply of any kind.
5. On mobile data, at minimum, the following changed.

What changed	Home broadband	Mobile data
Carrier	Fixed-line provider	Mobile operator
Public address	One number	A different number
Owning network	ISP's own network	Operator's network
Name server	1.1.1.1	Usually operator's
Answer for github.com	20.207.73.82	Possibly different
Address family	IPv4 only, no IPv6	Often IPv6 first
Middle boxes	Home router, ISP kit	Operator kit only
Physical path	Fixed line to city	Radio to core
Device stack	macOS	Phone OS

What changed	Home broadband	Mobile data
Client software	curl	Browser or app
Time of test	First attempt	Minutes later

6. That is eleven differences, and a careful person could find more.
7. Now the key sentence. Only one of those eleven needs to be the cause.
8. The test therefore proves exactly one thing: at that moment, some client somewhere could complete a request to `github.com`, so the service was up.
9. It does not prove your Wi-Fi is bad. It does not prove your router is bad. It does not prove your provider is blocking anything. It does not prove the address was different.
10. Those are all still open. The test closed one question and left eleven.

■ 29.2.4 PLAIN – what is really happening inside

1. Let us go through why each change could, on its own, produce the observed result.
2. Different provider. Providers buy connectivity from different upstream networks. If one of the reader's provider's upstream links to the destination network is broken, packets die there and nowhere else.
3. Different public address. Some destinations block or rate-limit ranges of addresses. If the reader's home range was on such a list, packets would be dropped at the far end with no reply. That behaviour is normal for abuse controls.
4. Different owning network. The reply has to find its way back to you. If the return route from the destination network to your provider is broken, your request may arrive perfectly and the answer never comes home. You cannot tell the difference from your side.
5. Different name server. A different name server may hand back a different address for the same name, because large services steer users to a nearby machine based on where the question came from.
6. Different address, therefore a different machine at the far end. If the original machine was in trouble and the new one was not, the fault is at the destination but only for one of its many front doors.
7. Different address type. If mobile data used the newer, longer kind of address and home broadband had none, then the successful request travelled over a completely separate network layer, sharing almost nothing with the failed one.
8. Different middle boxes. Equipment that inspects traffic can drop a connection silently based on the name being requested, the destination address, the port, or a pattern inside the first packet.
9. Different time. A brief routing problem lasting a few minutes looks exactly like a permanent block if you only test twice.
10. So the correct conclusion from the mobile-data test is one sentence: the destination service is alive and serving at least some clients, and the fault lies in something that differs between the two attempts.
11. Anything more specific than that is a guess. Guesses are allowed, but they must be labelled as guesses and then tested one at a time.

■ 29.2.5 TECHNICAL – the engineer’s version

1. Enumerated precisely, the variables changed by a Wi-Fi to cellular switch are as follows.
2. Access technology: IEEE 802.11 wireless local area network to 4G LTE or 5G New Radio, which means a different link layer, a different maximum frame size and a different scheduler.
3. Source address: a new public IPv4 address, almost always behind carrier grade network address translation on the mobile side.
4. Autonomous system: the autonomous system number, the identifier for a network under one routing policy, changes. The reader’s fixed-line provider and mobile operator hold different numbers.
5. Border Gateway Protocol path: both the forward path and, critically, the return path change. The return path is chosen by the destination network based on the new source prefix and is invisible from the client.
6. Peering relationship: the point where traffic hands off into Microsoft’s network changes. The reader’s fixed-line trace entered Microsoft at ae66-0.del01-96cbe-1b.ntwk.msn.net, a Delhi edge router. A mobile operator may hand off in Mumbai, Chennai or through a different transit provider altogether.
7. Recursive resolver: from 1.1.1.1, Cloudflare’s public resolver launched on 1 April 2018, to the mobile operator’s own resolver, reached through the cellular data profile.
8. Resolved address: GitHub fronts through Microsoft edge, and the answer depends on the querying resolver’s location and on any EDNS Client Subnet information, defined in RFC 7871. A different resolver can and often does yield a different anycast front end.
9. Address family: the session reported IPv6: (none) on the fixed line. Indian mobile networks commonly run IPv6-only bearers with 4GXLAT, defined in RFC 6877 from April 2013, so the phone may have used a AAAA record and an entirely separate IPv6 path.
10. Middlebox set: home NAT, provider transparent proxies, deep packet inspection appliances and any Server Name Indication filtering differ completely between a fixed-line and a mobile bearer.
11. Path maximum transmission unit: typical fixed-line paths carry 1500 or 1492 bytes; cellular bearers frequently present 1400 to 1428 bytes, which changes fragmentation and TCP maximum segment size behaviour.
12. TCP and TLS fingerprint: a different operating system produces a different initial window, a different option order and a different TLS ClientHello. Filtering devices that key on ClientHello fingerprints will treat them differently.
13. Temporal variable: the two observations are not simultaneous, so any transient routing event separates them.
14. In experimental terms this is a confounded comparison. Thirteen variables moved together, so the experiment has a very high power to answer one binary question and near-zero power to localize a cause.
15. The formally correct conclusion is exactly this: the destination service was reachable and functional from at least one vantage point at time T, therefore the hypothesis “service down” is falsified for time T.

■ 29.2.6 WORDS – remember these

1. Autonomous system — one company’s network under one policy — a routing domain identified by an autonomous system number and announced in BGP.
2. Return path — how the answer comes back — the route chosen by the remote network towards your source prefix, invisible from the client side.
3. Peering point — where two networks hand traffic over — a physical or virtual interconnection governed by a peering or transit agreement.

4. Anycast front end — many machines sharing one address — the same address announced from multiple locations, with routing picking the nearest.
5. Confounded comparison — too many things changed at once — an experiment in which multiple independent variables move together, so no single cause can be attributed.
6. Middlebox — a box in the path that does more than forward — any device that inspects or alters traffic, such as a NAT, proxy, firewall or DPI appliance.
7. 464XLAT — a way to run IPv4 apps on an IPv6-only phone network — RFC 6877, combining a customer-side translator with a provider-side NAT64.

29.3 The logic of a controlled experiment on a network

■ 29.3.1 PLAIN – in simple words

1. A controlled experiment is an old and simple idea. Change one thing. Keep everything else the same. See what happens.
2. If the result changes, the thing you changed is involved. If it does not, the thing you changed is probably innocent.
3. That is it. There is no more to the method than that sentence.
4. Networking breaks the rule constantly, because the easy tests change many things at once. Switching to mobile data changes thirteen things.
5. So the discipline is to design tests that move one thing and hold the rest still.
6. Before you run a test, say out loud what each possible result would mean.
7. If you cannot say what a pass would prove and what a fail would prove, the test is not worth running yet. Think first, type second.
8. This book gives that idea a name: the **eliminating power** of a test. It means how many possible causes a test can strike off your list.
9. A good test has high eliminating power. It strikes off half the list whichever way it goes.
10. A weak test is one where a pass tells you nothing and only a fail helps, or the other way round.
11. And there is one rule you must never break. Never conclude more than the evidence supports. Write down what you proved, not what you suspect.

■ 29.3.2 PLAIN – a picture in your head

1. Think of the game where someone thinks of a number between 1 and 100 and you have to find it, and they only answer yes or no.
2. A bad question is “is it 37”. Usually the answer is no, and you have removed one number out of a hundred.
3. A good question is “is it more than 50”. Whatever the answer, half the numbers vanish.
4. Seven good questions find any number in 1 to 100. Ninety-nine bad questions might still miss.
5. Fault finding is that game. The list of possible causes is the numbers. Each command you type is a question.
6. Your job is to ask the question that halves the list, not the question that confirms your favourite theory.
7. And notice something about the game. You are only allowed to trust the answers you actually got. If they said “more than 50”, you may not write down “so it is 73”.

Where this comparison breaks: in the number game each answer is truthful and final. Networks lie. A test can pass by luck on an intermittent fault, and a test can fail for a reason unrelated to what you were testing, such as a tool that blocks certain traffic. So you sometimes need to ask the same question twice, in two different ways.

■ 29.3.3 PLAIN – a worked example

1. Suppose the list of possible causes for the reader's fault starts like this.

- A The Mac's network settings are wrong
 - B The Wi-Fi link to the router is bad
 - C The home router is broken or blocking
 - D The ISP's access network is broken
 - E The ISP blocks this destination
 - F The path into Microsoft is broken
 - G Microsoft drops traffic from this address range
 - H github.com is down
 - I DNS gave a wrong or stale address
2. Nine causes. Now consider two candidate tests.
3. Test one: reload the page in a different browser. If it fails, you have removed nothing at all, because every cause above would still produce a failure. If it succeeds, you have learned that the first browser was at fault, which was not even on the list. Very low eliminating power.
4. Test two: from the same Mac, run a name lookup and then try to open a connection to the numeric address directly.
5. If the name lookup works and the connection still fails, you strike off I, because naming worked, and you learn the failure is later than naming.
6. Test three: load a different website, say `example.com`, from the same Mac on the same Wi-Fi. If it loads, you strike off A, B, C and D in one move, because a working page proves the Mac, the Wi-Fi, the router and the access network all carry traffic to the wider internet.
7. That single test removed four of nine causes. That is high eliminating power, and it takes two seconds.
8. Test four: mobile data. It strikes off H, because the site answered someone.
9. After those tests the list is E, F and G. Three causes instead of nine, in under a minute, and every step was justified.
10. Notice that no test so far can separate E, F and G, because all three produce identical silence from where you sit. That is an honest and useful place to stop and say so.

■ 29.3.4 PLAIN – what is really happening inside

1. There is a hidden structure to good fault finding, and it is worth naming.
2. Every fault has a chain of things that must all work. Cable, address, router, route, naming, connection, encryption, application.
3. A chain fails at exactly one link first. Your job is to find the first broken link, not to admire the whole chain.
4. Searching a chain has a best method: cut it in the middle. Test something halfway along. Then you know which half holds the break, and you repeat.
5. That is called bisection, and it is the same method used to find which change in a code history broke a program.
6. The ladder in the next section is a chain laid out in order, from the nearest link to the furthest, so that you can walk it or bisect it.
7. Walking from the bottom is slower but never confuses you. Bisecting is faster but needs you to know the chain well.
8. One more rule. When a test surprises you, believe the test and change your model. Do not explain the result away.
9. Most long debugging sessions are long because somebody decided early what the answer was and then read every result as support for it.

10. Write down, in words, what you have proved after each step. A three-line note stops you from re-testing the same thing four times.

■ 29.3.5 TECHNICAL – the engineer’s version

1. Formally, define the hypothesis space H as the set of candidate root causes consistent with all evidence so far.
2. A test T partitions H into the subset consistent with a pass and the subset consistent with a fail. Define the eliminating power of T as the size of the smaller of those two subsets, expressed as a fraction of H .
3. A test with eliminating power near 0.5 is a bisecting test. A test with power near 0 is decorative.
4. This is the information-theoretic view: the ideal test maximizes expected information gain, which for a binary test is maximized when the two outcomes are equally likely.
5. Medicine names the same two properties. A highly sensitive test with a negative result rules a condition out. A highly specific test with a positive result rules it in. Network tests have exactly these two flavours.
6. Worked eliminating power for common tests, taking the nine-cause list above:

Test	Rules out on pass	Power
Different browser	Nothing	Very low
Load example.com	Local chain, 4 of 9	High
dig github.com	DNS, 1 of 9	Medium
Mobile data	Site down, 1 of 9	Medium
nc to port 443	Nothing new on fail	High on pass
Different port, 22	Port policy	High

7. Design rules that follow from this.
8. Hold the destination fixed and vary the source, or hold the source fixed and vary the destination. Never vary both.
9. Vary one layer at a time. Name resolution, then network reachability, then transport, then application.
10. Prefer tests whose failure mode is distinguishable. A silent timeout is a weak signal because many causes produce it. A TCP reset, an ICMP administratively prohibited message, or an HTTP 403 are strong signals because few causes produce them.
11. Record the exact command, the exact output and the timestamp. Use `date -u +%FT%TZ` on macOS to stamp a log line in coordinated universal time, so that evidence from two machines can be lined up.
12. Beware of the tool as a variable. `curl`, a browser, and `git` do not use identical name resolution, proxy settings or TLS libraries. A browser may use its own encrypted DNS while `curl` uses the system resolver.
13. Beware of caching as a hidden variable. Negative DNS answers, TCP connection reuse and HTTP caches can make a second test not independent of the first. Start a fresh process for each test where it matters.
14. State conclusions in the form: evidence, therefore eliminated set. Anything else is a hypothesis and must be labelled as one.

■ 29.3.6 WORDS – remember these

1. Controlled experiment — change one thing only — a test in which a single independent variable is varied while all others are held constant.
2. Eliminating power — how much of your list a test removes — this book's name for the fraction of the hypothesis space excluded by either outcome.
3. Bisection — cut the chain in the middle — binary search over an ordered sequence of dependencies to locate the first failing link.
4. Confounding variable — something else that moved too — an uncontrolled variable that could equally explain the observed change.
5. Silent failure — nothing comes back at all — a drop with no ICMP error and no TCP reset, which is the least informative failure mode there is.
6. Strong signal — a failure that names its own cause — an explicit error such as TCP RST, ICMP type 3 code 13, or an HTTP status code.

29.4 The diagnostic ladder, rung by rung

■ 29.4.1 PLAIN – in simple words

1. Getting a web page involves a chain of things that must all work.
2. The ladder puts that chain in order, from the closest thing to you up to the furthest thing away.
3. Rung 1: does my machine have a working network connection with an address.
4. Rung 2: can I talk to my own router, the box in my home.
5. Rung 3: can I talk to a machine out on the internet using its number.
6. Rung 4: can I turn a name into a number.
7. Rung 5: can I open a connection to the exact service I want, on its port.
8. Rung 6: does the full request, with encryption, actually succeed.
9. Rung 7: if it fails, how far along the path do my packets get.
10. You climb from 1 upwards. The first rung that fails is where to look.
11. Every rung passing means the rungs below it are fine. That is the whole value of putting them in order.
12. And every rung has a limit. Passing a rung proves something specific and nothing more, and it is worth knowing exactly what.

■ 29.4.2 PLAIN – a picture in your head

1. Think of posting a parcel, and the checks a careful person makes.
2. Rung 1 is “do I have a pen and paper and a return address”. Without an address of your own, nothing can come back.
3. Rung 2 is “is the postbox at the end of my street working”. If your own postbox has been sealed, nothing leaves.
4. Rung 3 is “can I send anything at all to another city”. Not to a specific person. Just anything, anywhere far away.
5. Rung 4 is “can I look up my friend's street address from their name”.
6. Rung 5 is “is there a door at that address, and does it open”.
7. Rung 6 is “did the person inside take the parcel, check my identity papers and give me a receipt”.
8. Rung 7 is “if it never arrived, which sorting office was the last to scan it”.
9. Notice you cannot sensibly ask question 6 before question 5. There is no point asking whether the person signed for it if the door does not exist.
10. That ordering is what stops you wasting time. Most people start at rung 6, fail, and then guess.

Where this comparison breaks: with a parcel, every stage leaves a record and somebody can tell you where it stopped. On a network the middle stages usually leave no record at all, and a packet can be dropped by a machine that never tells anyone, including its own operator.

■ 29.4.3 PLAIN – a worked example

- Here is the ladder as a table. macOS commands, since that is what the reader uses. Linux differences come later in this section.

Rung	Question	macOS command
1	Do I have an address	<code>ipconfig getifaddr en0</code>
2	Can I reach my router	<code>ping -c 3 192.168.0.1</code>
3	Can I reach the internet	<code>ping -c 3 1.1.1.1</code>
4	Can I resolve a name	<code>dig +short github.com</code>
5	Can I open the port	<code>nc -vz github.com 443</code>
6	Does the request work	<code>curl -sSI https://github.com</code>
7	Where does it stop	<code>traceroute -n github.com</code>

- And here is what each rung settles, and what it leaves open.

Rung	A pass proves	A pass does not prove
1	Link up, address held	Anything can be reached
2	Local link works	Anything beyond the router
3	Routing off the LAN	Naming works
4	Naming works	The address is reachable
5	TCP reaches service	TLS or HTTP work
6	Whole stack works	It works for others
7	Path up to last hop	Silence means fault

- Third table: what a failure at each rung means.

Rung	A failure means
1	No link, no DHCP, or wrong port
2	Wi-Fi, cable, or router fault
3	No route, or ICMP filtered
4	Resolver or DNS path problem
5	Filtering, route, or port shut
6	TLS, certificate, or server error
7	Only shows where replies stop

- Two of those entries deserve a warning right away.
- Rung 3 failing may only mean that ping replies are blocked, not that routing is broken. Always confirm with a second test that does not use ping.
- Rung 7 failing to reach the end is the weakest evidence on the whole ladder. Silence at the end of a trace is completely normal.

■ 29.4.4 PLAIN – what is really happening inside

1. Rung 1. Your machine needs three things: a working link, an address of its own, and a default route, which is the instruction “send anything not local to the router”.
2. A pass proves the network card is on and the machine has an address, usually handed out by the router. A pass does not prove that any packet has ever left the building.
3. A failure means the cable is out, the Wi-Fi did not join, or the address handout failed. If you see an address starting 169.254, the handout failed and the machine gave itself a placeholder.
4. Rung 2. Ping your own router. On the reader’s network that is 192.168.0.1.
5. A pass proves the wireless or cable link works, your address is on the right network, and the router is alive and answering.
6. A pass does not prove the router can reach anything else. A router with a dead internet line still answers you cheerfully.
7. A failure means the local link is broken, or your address is on a different network from the router, or the router blocks ping to itself, which is rare for home routers and common for company ones.
8. Rung 3. Ping a numbered address out on the internet, such as 1.1.1.1 or 8.8.8.8. Use the number, never a name.
9. A pass proves your packets leave your network, cross your provider, reach a distant machine, and come back. That is a lot, and it is proved in one line.
10. It also separates two questions that beginners mix up: reaching things, and naming things. Using a number takes naming out of the experiment.
11. A pass does not prove that naming works, and does not prove any particular site is reachable. Different destinations take different paths.
12. A failure is ambiguous, because many networks drop ping on purpose. Retest with something that is not ping before concluding anything.
13. Rung 4. Turn a name into a number. This is the naming system doing its job.
14. A pass proves your machine can reach a name server and get an answer, so both naming and the path to the name server are healthy.
15. A pass does not prove the returned address is reachable, or correct, or fresh. A cached wrong answer looks exactly like a right one.
16. A failure means your name server is unreachable or refusing. Test again naming a different server directly to separate the two.
17. Rung 5. Open a connection to the actual port. Web pages over encryption use port 443. This is the first rung that tests the service you actually want.
18. A pass proves that packets reach the service, that the service is listening and that a full handshake completed in both directions. This is the strongest single proof of reachability there is.
19. A pass does not prove the encryption will succeed, that the certificate is valid, or that the site will answer with a page.
20. A failure with a refusal message means something answered and said no. A failure with silence means nothing answered at all, which is the harder case and the one the reader hit.
21. Rung 6. Make the real request. Now encryption and the web protocol are involved as well.
22. A pass proves everything below, plus certificate checking, plus a server that produced a real answer.
23. A pass does not prove it works for other people, other names on the same machine, or the rest of the site.
24. A failure here, with rung 5 passing, is a very specific and useful state: the network is fine and the problem is encryption or the application.
25. Rung 7. Find out how far your packets get by sending probes with a short life and collecting the

complaints from each machine along the way.

26. A pass, meaning a complete trace, proves the forward path exists and each machine on it was willing to identify itself.
27. A pass does not prove your actual traffic follows that same path, because the probe may use a different protocol and port from your real connection.
28. A failure, meaning stars at the end, mostly means the last machines choose not to reply. It is a hint, not a verdict.

■ 29.4.5 TECHNICAL - the engineer's version

1. Rung 1, interface and address. Three commands, each answering a different part of the question.

```
ifconfig en0
ipconfig getifaddr en0
networksetup -getinfo "Wi-Fi"
netstat -rn -f inet | head -5
```

2. `ifconfig en0` prints link state and addresses. Look for `status: active` and an `inet` line. macOS prints the mask in hexadecimal, for example `netmask 0xfffff000`, which is a /24.
3. `ipconfig getifaddr en0` prints only the IPv4 address and exits with status 1 and no output if there is none, which makes it ideal inside scripts.
4. `networksetup -getinfo "Wi-Fi"` prints the service-level view: address, mask, router and hardware address. Use `networksetup -listallhardwareports` first to map `en0` to a service name, because on many Macs `en0` is Wi-Fi and on others it is a wired or adapter port.
5. `netstat -rn -f inet` shows the routing table. The default line names the gateway and interface. Without a default route, nothing off-link works.
6. On macOS, `utun` interfaces are point-to-point tunnels. The reader had several. They are created by VPN clients, by Apple's own services such as Back to My Mac and iCloud Private Relay, and by content filters. A `utun` with a default route can silently capture traffic. Check with `route -n get default` which interface actually wins.
7. Rung 1 pass criteria: `status: active`, a non-169.254 address, a default route. 169.254.0.0/16 is link-local, defined in RFC 3927 from May 2005, and self-assigned when DHCP fails.
8. Rung 2, the default gateway. The reader's is 192.168.0.1, inside the 192.168.0.0/16 private range from RFC 1918, published February 1996.

```
ping -c 3 -W 1000 192.168.0.1
arp -a -n | grep 192.168.0.1
```

9. `-c 3` sends three packets. `-W 1000` waits at most 1000 milliseconds per reply. Note that on macOS `-W` is milliseconds, while on Linux `-W` is seconds. This is a real and frequent source of confusion.
10. `arp -a -n` shows the Address Resolution Protocol cache. If the gateway appears with a hardware address, layer 2 resolution succeeded even if ICMP is filtered. An (incomplete) entry means the router did not answer ARP at all, which is a much more serious finding.
11. Rung 3, off-network reachability by address.

```
ping -c 5 1.1.1.1
ping -c 5 8.8.8.8
nc -vz -G 3 1.1.1.1 443
```

12. 1.1.1.1 is Cloudflare's resolver, launched 1 April 2018. 8.8.8.8 is Google Public DNS, launched 3 December 2009. Both answer ICMP echo and both listen on TCP 443, so you can test the same target two ways.

13. If ping fails and nc succeeds, ICMP is filtered and routing is fine. That pair of results is worth more than either alone.

14. Rung 4, name resolution.

```
dig +short github.com
dig @1.1.1.1 +short github.com
dig @8.8.8.8 +short github.com
scutil --dns | head -20
dscacheutil -q host -a name github.com
```

15. Plain dig uses the system resolver list. dig @server bypasses it and asks a named server directly. Running both separates “resolution is broken” from “my configured resolver is broken”.

16. scutil --dns is the macOS-specific view of resolver configuration, including per-domain resolvers installed by VPN clients. This is where a utun interface reveals that it has hijacked name resolution for some domains.

17. dscacheutil -q host -a name github.com asks the macOS directory service, which is what applications actually use. dig talks to DNS directly and bypasses the system cache, so the two can disagree. That disagreement is itself a diagnosis.

18. Apple ships dig, host and nslookup as an old bundled copy of the BIND utilities from the Internet Systems Consortium. The bundled version has lagged upstream by years. If your Mac lacks dig, install a current one with brew install bind and use dscacheutil or dns-sd -G v4 name in the meantime. Treat the exact presence of these tools as version-dependent rather than guaranteed.

19. Rung 5, transport reachability.

```
nc -vz -G 5 github.com 443
nc -vz -G 5 20.207.73.82 443
curl -v --connect-timeout 5 -o /dev/null https://github.com
```

20. -v makes nc report the outcome, -z means connect and send no data, -G 5 sets a five second TCP connection timeout. Without -G, nc waits for the operating system default.

21. On macOS the operating system default for a connection attempt is governed by sysctl net.inet.tcp.keepinit, whose default is 75000, in milliseconds. So an unanswered connection takes about 75 seconds to fail if no shorter timeout is set.

22. On Linux the equivalent is net.ipv4.tcp_syn_retries, default 6, giving roughly 127 seconds with exponential backoff from an initial retransmission timeout of 1 second as specified in RFC 6298 from June 2011.

23. Distinguish the three transport outcomes precisely:

Outcome	On the wire	Meaning
Connected	SYN, SYN-ACK, ACK	Service reachable
Refused	SYN, then RST	Host up, port shut
Timed out	SYN, silence	Dropped somewhere
Unreachable	ICMP type 3	Router said no

24. Rung 6, application level.

```
curl -sSI https://github.com
curl -sS -o /dev/null -w '%{http_code} %{time_total}\n' \
https://github.com
```

25. -I sends a HEAD request, which asks for headers only. -s silences the progress meter, -S keeps error messages visible. Together -sS is the standard pairing for scripts.

26. A pass here exercises TCP, TLS 1.2 or 1.3, certificate chain validation against the system trust store, ALPN negotiation and HTTP.
27. Rung 7, path discovery.

```
tracert -n -q 1 -w 2 -m 20 github.com
sudo tracert -I -n github.com
sudo tracert -n -P TCP -p 443 github.com
```

28. -n skips reverse name lookups and makes the trace much faster. -q 1 sends one probe per hop instead of three. -w 2 waits two seconds instead of five. -m 20 stops at twenty hops instead of the macOS default, which is the system TTL, normally 64.
29. macOS tracert defaults to UDP probes to ports from 33434 upwards, the base port chosen by Van Jacobson when he wrote the tool in 1987 at Lawrence Berkeley Laboratory.
30. Path asymmetry rule: tracert shows the forward path only, one hop at a time, and each displayed round-trip time includes the return path from that intermediate router, which may be entirely different.

■ 29.4.6 WORDS – remember these

1. Default route — where to send anything not local — the 0.0.0.0/0 entry in the routing table, pointing at the gateway.
2. Link-local address — the placeholder a machine gives itself — 169.254/16 under RFC 3927, a sign that DHCP failed.
3. ARP cache — the table of neighbours' hardware addresses — the result of Address Resolution Protocol, viewable with `arp -a`.
4. Three-way handshake — the opening of a connection — SYN, SYN-ACK, ACK, after which TCP considers the connection established.
5. RST — an explicit refusal — a TCP reset segment, meaning a host exists but nothing is listening or a device rejected the attempt.
6. HEAD request — ask for headers, not the page — an HTTP method returning the same headers as GET with no message body.
7. Bisection point — the rung that halves the problem — in practice rung 5, because it separates all network causes from all application causes.

29.5 The reader's own fault, mapped onto the ladder

■ 29.5.1 PLAIN – in simple words

1. Now we take the real fault from the reader's session and put every piece of evidence on a rung.
2. The Mac had an address and a working local network. Rung 1 passed.
3. The router at 192.168.0.1 answered. Rung 2 passed.
4. Packets reached machines out on the internet and came back. Rung 3 passed.
5. The name `github.com` turned into the number 20.207.73.82 quickly and correctly. Rung 4 passed.
6. The attempt to open a connection to 20.207.73.82 on port 443 got nothing at all. Rung 5 failed, and failed by silence.
7. Because rung 5 failed, rung 6 never happened. There was no encryption step and no web request, because there was no connection to run them over.
8. The trace showed packets travelling through the reader's provider, into Microsoft's network at Delhi, through Mumbai, into Pune, and then silence from hop 13 onwards. Rung 7 gave a partial answer.
9. And the same site loaded instantly on mobile data, which killed the theory that GitHub was down.

10. Put those together and something very specific appears.
11. Naming worked but connecting did not. That combination is one of the most informative results you can get, and the next block explains why.

■ 29.5.2 PLAIN – a picture in your head

1. Go back to the parcel. You looked up your friend’s street address from a directory, and the directory answered instantly with a correct address.
2. So the directory service is fine, the roads to the directory office are fine, and you hold a real address in your hand.
3. You then walk to that address, knock, and nothing happens. No answer. No “go away”. No locked-gate sign. Just silence.
4. And you know from a friend in another city that somebody was home the whole time and answered their knock.
5. What is left? Something between your street and that door swallowed you.
6. The important part of the picture is the silence. A locked gate is information. A sign saying “no entry” is information. Nothing at all is the deliberate absence of information.
7. Anything that wants to hide the fact that it is stopping you produces exactly this: nothing.

Where this comparison breaks: a person walking can see the buildings they pass. Your packet cannot. It has no memory of where it has been, and if it is dropped it does not scream. Neither end is told. That is not a flaw in your test, it is how the network was designed.

■ 29.5.3 PLAIN – a worked example

1. Here is the reader’s evidence, rung by rung.

Rung	Result	Conclusion
1 Interface	Pass	Mac has address, link up
2 Router	Pass	Local network is healthy
3 Public IP	Pass	Routing off the LAN works
4 DNS	Pass	Naming is not the fault
5 TCP 443	Fail, silent	Fault is at or before service
6 TLS, HTTP	Not reached	No information
7 Traceroute	Partial	Reaches Pune, then quiet

2. And the cross-check, which sits outside the ladder.

Cross-check	Result	Conclusion
Mobile data, same phone	Pass	Site is up, not down

3. A note on rung 3, told honestly. The session record does not include a ping 1.1.1.1 result. What it does include is that name lookups against 1.1.1.1 worked.
4. That is stronger evidence than ping, not weaker. It means packets left the Mac, crossed the provider, reached Cloudflare’s resolver, were processed, and the answers came all the way back.
5. So rung 3 is proved, but by a different observation than the one in the table. Say which observation proved a thing, not just that it is proved.
6. Here is the same picture as a diagram.

RUNG	READER'S RESULT
7 traceroute	hop 12 Pune, then * * *
6 curl -sSI https://	never ran
5 nc / curl to :443	FAIL - silence, 15 s
4 dig github.com	PASS -> 20.207.73.82
3 reach a public IP	PASS via DNS to 1.1.1.1
2 ping 192.168.0.1	PASS - hop 1 answered
1 ifconfig / address	PASS - address held

break is between rung 4 and rung 5	

■ 29.5.4 PLAIN - what is really happening inside

1. Why is “naming works, connecting fails” such a strong pair of results?
2. Because the two use different machines, different protocols, different ports, and, most importantly, different destinations.
3. The name lookup went to 1.1.1.1, a Cloudflare machine, over the simple message protocol on port 53.
4. The connection attempt went to 20.207.73.82, a Microsoft machine, over the connection protocol on port 443.
5. Naming working proves the general machinery is fine: the Mac, the router, the provider’s access network, the provider’s core, and the ability to send and receive from a distant network.
6. So the fault cannot be anything general. It must be specific to something about the second attempt.
7. Only three kinds of thing differ between the two attempts: the destination address, the port and protocol, and the path taken to that destination.
8. That is the entire surviving list. Everything else has been eliminated by the naming test.
9. Now add the silence. The failure produced no refusal message and no error report from any router.
10. If the far machine were up but not listening, you would get a refusal. You did not.
11. If a router had no route, it would normally send back a “cannot get there” complaint. You did not get one.
12. If a firewall were configured to reject politely, it would say so. It did not.
13. Silence is what a device does when it is configured to drop, or when a packet enters a hole in the routing and dies without anyone taking responsibility.
14. Now add the mobile data result. The destination answered somebody else at the same time.
15. The honest conclusion: the destination service was up, the reader’s own machine and local network were fine, naming was fine, and packets to that specific address on that specific port were being dropped somewhere between the reader’s provider and that service, with no error returned.
16. What we cannot say: which device dropped them, whether it was intentional, or which organization owns it. The evidence does not reach that far.

■ 29.5.5 TECHNICAL – the engineer’s version

1. The evidence set, stated formally.
2. Layer 1 and 2: link active, ARP resolution to 192.168.0.1 succeeded, hop 1 in the trace answered. Eliminated as a cause.
3. Layer 3 local: address held, default route present, gateway responsive. Eliminated.
4. Layer 3 transit: UDP port 53 to 1.1.1.1 completed round trips with normal latency. Forward and return paths to Cloudflare’s anycast prefix are intact. Provider access network and core eliminated as general failures.
5. Layer 4 naming path: github.com resolved to 20.207.73.82, inside 20.192.0.0/10, registered to Microsoft and announced by AS8075. Correct and geographically plausible for India. DNS eliminated.
6. Layer 4 transport to target: `curl -v https://github.com` printed `Trying 20.207.73.82:443...` and produced no further output for 15 seconds. No TCP RST, no ICMP type 3 of any code.
7. That means the SYN was sent and retransmitted according to the operating system’s schedule, and nothing came back. On macOS a bare connect would run for about 75 seconds under `net.inet.tcp.keepinit`, so the 15 second cutoff indicates a client-imposed limit rather than the kernel giving up. Many tools set one.
8. Layer 7: never reached. No TLS ClientHello was ever sent, because TCP never established. Therefore no conclusion about certificates, SNI filtering at the TLS layer, or HTTP is available from this evidence.
9. Path evidence. The recorded trace, exactly as captured:

```

1 192.168.0.1
2 172.31.0.17
3 137.97.29.249
4 172.26.22.235
5 172.16.18.33 / 172.26.14.75 / 172.26.22.234
6 172.26.14.75 / 172.16.18.33
7 ae66-0.del01-96cbe-1b.ntwk.msn.net (104.44.196.187)
8 be23.rwa02.bom01.ntwk.msn.net (104.44.55.163)
9 be1010.owr02.bom01.ntwk.msn.net (104.44.53.119)
10 be5.ibr02.pnq21.ntwk.msn.net (104.44.31.62)
11 ae104-0.rwa03.pnq21.ntwk.msn.net (104.44.20.52)
12 ae106-0.rwa04.pnq20.ntwk.msn.net (104.44.20.42)
13-20 * * * (no reply)
```

10. Hops 2 to 6 are RFC 1918 private addresses inside the provider’s core. 172.16.0.0/12 covers 172.16.x through 172.31.x, so 172.31.0.17, 172.26.x and 172.16.18.33 are all private. Hop 3, 137.97.29.249, is public and belongs to the provider.
11. Hops with two or three addresses listed, namely 5, 6, 8, 9 and 11, are per-flow or per-packet load balancing across parallel links. Normal, not a fault.
12. From hop 7 the path is inside Microsoft’s backbone, `ntwk.msn.net`. Site codes follow airport-style city codes: `del01` Delhi, `bom01` Mumbai, `pnq20` and `pnq21` Pune.
13. Router role codes `ibr`, `rwa` and `owr` mean inter-region backbone router, regional wide-area router and outer wide-area router. These are Microsoft naming conventions, not a standard. Interface prefixes `ae`, `be` and `po` all mean a bundle of physical links treated as one logical link, and are vendor-specific words for the same idea.
14. Hops 13 to 20 returning * * * proves nothing on its own. Control-plane ICMP time-exceeded generation is routinely rate-limited or disabled inside large backbones. Cisco IOS, for example, rate-limits ICMP unreachable to one every 500 milliseconds by default.

15. The decisive evidence is therefore not the trace. It is the pairing of a silent TCP timeout with a successful request from a different vantage point at the same time.
16. Surviving hypothesis set, in the reader's case:

Hypothesis	Consistent with evidence
Site down	No, refuted by mobile test
DNS wrong	No, address is correct
Local network fault	No, DNS round trips worked
Path or policy drop	Yes
Return path missing	Yes
Source prefix filtered	Yes
Transient routing event	Yes

17. Four hypotheses survive and no test performed so far separates them, because all four produce identical silence at the client.
18. Separating them requires evidence from outside the client: a TCP traceroute to port 443, a test to a second address in the same Microsoft prefix, a test from another host inside the same provider, or a looking-glass query from the destination side.

■ 29.5.6 WORDS - remember these

1. Silent drop — nothing comes back — a packet discarded without generating ICMP or TCP error, by policy or by a routing black hole.
2. Black hole — a place packets vanish — a route that accepts traffic and discards it, often a null route or a stale advertisement.
3. Return path failure — your request arrives, the answer cannot come home — asymmetry where the destination network lacks a working route to your prefix.
4. Load-balanced hop — several addresses on one trace line — parallel links between the same two routers, chosen per packet or per flow.
5. Control plane — the router's own brain — the processor that generates ICMP errors and runs routing protocols, deliberately rate-limited.
6. Surviving hypothesis set — what is still possible — the causes not yet eliminated by any observation, which is the only honest output of a diagnosis.

29.6 Cross-checks that multiply your certainty

■ 29.6.1 PLAIN - in simple words

1. A cross-check is a test that deliberately changes one big thing and keeps the rest as similar as possible.
2. Five of them cover almost every case, and each one eliminates a different family of causes.
3. Same destination, different network. Use mobile data or a hotspot.
4. Same network, different device. Use a phone on the same Wi-Fi, or a second computer.
5. Same destination, tested from far away. Use a shell on a server elsewhere, or a website that checks a site for you.
6. Same network, different destination. Load a different site on the same Wi-Fi.
7. Same destination, but tunnelled. Turn on a virtual private network, which wraps your traffic and sends it out from somewhere else.
8. Each of these answers a different question, and it is worth learning which.

9. The order matters too. Do the cheap ones first. Loading a different site takes two seconds and removes four causes.

■ 29.6.2 PLAIN – a picture in your head

1. Think of a doctor with five simple tests and no scanner.
2. Press here. Does it hurt? Press there. Does it hurt? Each answer removes a set of possibilities.
3. No single press gives a diagnosis. The pattern of yes and no does.
4. The doctor is not looking for the answer in one test. The doctor is narrowing.
5. And a good doctor knows which press is worth doing first: the one that will change what they do next, whichever way it goes.
6. A test whose result would not change your next action is not worth doing.

Where this comparison breaks: a body is one system and stays put. Your network path can change between two presses, and a fault that lasts four minutes will give you a yes and then a no with no change in your hands.

■ 29.6.3 PLAIN – a worked example

1. Here are the five cross-checks with exactly what each result eliminates.

Cross-check	Pass eliminates	Fail eliminates
Other network	Site down	Path-only causes
Other device, same net	That device	Device-specific causes
Public shell elsewhere	Site down	Site-side doubt
Other site, same net	Local chain	Destination-specific
Same site over VPN	Path policy	Nothing much

2. Walk through them one at a time with the reader's case.
3. Other network. The reader ran this. It passed. The claim "GitHub is down" is dead. What remains is everything on the reader's own path.
4. Other device, same network. If a phone on the same Wi-Fi also fails, the Mac is not the problem, and every Mac-specific idea can be struck off: the `utun` tunnel interfaces, a local firewall, a broken resolver setting.
5. If the phone succeeds on the same Wi-Fi, that is enormous. It means the router and the provider carry the traffic, and something on the Mac is the cause.
6. Public shell elsewhere. Any server you can log into, in any country, confirms the site independently and with a proper command rather than a phone browser.
7. Other site, same network. Loading `example.com` from the same Mac on the same Wi-Fi proves the Mac, the Wi-Fi, the router, the provider's access network and the provider's core all work.
8. That is the highest-value two-second test on the list.
9. Same site over a virtual private network. Your traffic is wrapped up and sent out from a machine elsewhere, so it enters the destination from a different address and a different path.
10. If it works over the tunnel and not without it, that is a strong pointer towards path or policy rather than anything at the destination.
11. But notice the honest entry in the table. A failure over the tunnel eliminates very little, because tunnels themselves can be blocked, slow, or misconfigured, and you have added a whole new set of possible faults.

■ 29.6.4 PLAIN – what is really happening inside

1. Each cross-check works by holding a set of components constant and swapping one whole layer.
2. Different network swaps the entire path and your address, and keeps the destination.
3. Different device swaps the operating system, the software and the local settings, and keeps the path.
4. Different destination swaps the far end and most of the path, and keeps everything local.
5. Tunnel swaps the middle of the path and your apparent address, and keeps both endpoints.
6. Elsewhere-shell swaps everything except the destination.
7. Put them in a grid and you can see that between them they cover every component: your machine, your local network, your provider, the path, and the destination.
8. This is why five tests can produce a confident answer where one test cannot.
9. There is a trap here. Tests are only independent if they really do differ.
10. A phone on the same Wi-Fi that has fallen back to mobile data without you noticing is not a same-network test. Turn mobile data off first.
11. A second computer that shares your virtual private network is not an independent path.
12. And an online checker that tests from a data centre may reach a completely different front-door machine than a home connection would, so a pass there does not prove your address family or your region is served.

■ 29.6.5 TECHNICAL – the engineer's version

1. Precise commands for each cross-check on macOS.

```
# 1. Same network, different destination
curl -sSI --max-time 8 https://example.com
curl -sSI --max-time 8 https://www.wikipedia.org

# 2. Same destination, force the address, skip DNS
curl -v --max-time 8 --resolve github.com:443:20.207.73.82 \
  https://github.com

# 3. Same destination, different port
nc -vz -G 5 github.com 22
nc -vz -G 5 github.com 443

# 4. Same destination, different address family
curl -4 -sSI --max-time 8 https://github.com
curl -6 -sSI --max-time 8 https://github.com
```

2. Test 2 is the one people forget. `--resolve host:port:address` tells curl to use that address for that name without a lookup, so you can test a specific front end while keeping the correct Server Name Indication and certificate name.
3. Test 3 separates destination reachability from port policy. In the reader's session, the git transports gave exactly this kind of split evidence: one transport worked while the other did not, and those two use different ports.
4. Test 4 matters because the session reported IPv6: (none). If a destination is reachable over IPv6 and not IPv4 on your network, or the reverse, forcing the family names the fault immediately.
5. Independent vantage points, ranked by strength:

Vantage point	Independence	Effort
Mobile hotspot	High	Seconds

Vantage point	Independence	Effort
Friend on other ISP	Very high	Minutes
Cloud shell or VPS	Very high	Minutes
RIPE Atlas measurement	Highest	Hours
Web outage checker	Low	Seconds
Same-network device	Low path	Seconds

6. A looking glass is a public web or telnet interface on a provider's router that lets you run ping and traceroute from that provider's network towards you. Many transit providers and internet exchanges publish one. This is the only easy way to see the return path towards your own address.
7. RIPE Atlas, run by the RIPE Network Coordination Centre since 2010, will run a traceroute or TLS check towards your target from probes in chosen countries or chosen autonomous systems, which turns "is it just me" into a measurable question.
8. Record for each cross-check: source address family, source autonomous system, resolver used, resolved address, port, exact command, exact result, and a coordinated universal time stamp. Without the resolved address, two tests may not have been aimed at the same machine at all.

■ 29.6.6 WORDS – remember these

1. Cross-check — a test that swaps one whole layer — an independent measurement designed to eliminate a family of causes at once.
2. Vantage point — where you measure from — a source with its own address, provider and path to the target.
3. Looking glass — someone else's router you may query — a public interface for running ping, traceroute and BGP queries from a remote network.
4. Server Name Indication — telling the server which site you want — the SNI field in the TLS ClientHello, sent before encryption of the certificate exchange, and therefore visible to middleboxes.
5. Independent test — one that shares nothing with the last — a measurement whose failure modes do not overlap with the previous one.
6. Address family — the kind of address in use — IPv4 or IPv6, selectable in most tools with -4 and -6.

29.7 Ping, honestly

■ 29.7.1 PLAIN – in simple words

1. Ping sends a tiny message that means "are you there", and waits for a message back that means "yes".
2. That is the whole of it. It does not fetch a page. It does not check a service. It does not know what a website is.
3. It is not part of the web, and it is not part of the connection machinery that carries your data. It is a separate, simpler kind of message.
4. Because it is separate, a machine can answer ping while its website is broken, and can refuse ping while its website works perfectly.
5. Both of those situations are completely normal. Neither is a bug.
6. Ping is the most over-trusted tool in networking. People treat "it pings" as proof that a service works, and "it does not ping" as proof of a fault.
7. Neither reading is safe. Ping tells you about one particular kind of message reaching one particular machine and coming back.

8. It is still useful. It is fast, it needs no name lookup if you use a number, and it measures how long the round trip takes.
9. Use it to measure, not to conclude.

■ 29.7.2 PLAIN – a picture in your head

1. Ping is shouting “hello” at a building and listening for “hello” back.
2. If you hear it, someone is inside and the air between you carries sound.
3. If you hear nothing, there are three possibilities and you cannot tell them apart: nobody is in, somebody is in but has been told not to shout back, or the wind ate your shout.
4. And even a cheerful “hello” back tells you nothing about whether the shop inside is open, whether it has your item, or whether the till works.
5. The shout and the shopping are different activities.

Where this comparison breaks: your shout travels in a straight line, but a ping travels through a dozen machines that each decide whether to pass it on, and several of them treat it as the least important thing they handle that second.

■ 29.7.3 PLAIN – a worked example

1. Here is real output from a Mac, with every field named.

```
$ ping -c 4 1.1.1.1
PING 1.1.1.1 (1.1.1.1): 56 data bytes
64 bytes from 1.1.1.1: icmp_seq=0 ttl=57 time=8.412 ms
64 bytes from 1.1.1.1: icmp_seq=1 ttl=57 time=9.021 ms
64 bytes from 1.1.1.1: icmp_seq=2 ttl=57 time=8.703 ms
64 bytes from 1.1.1.1: icmp_seq=3 ttl=57 time=9.633 ms

--- 1.1.1.1 ping statistics ---
4 packets transmitted, 4 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 8.412/8.942/9.633/0.441 ms
```

2. 56 data bytes is what you sent. Add the 8-byte header and you get the 64 bytes that came back.
3. icmp_seq is a counter so you can tell which reply belongs to which request, and see if any went missing. On macOS it starts at 0. On Linux it starts at 1.
4. ttl=57 is the remaining life of the reply packet. Every packet starts with a number and every machine that forwards it subtracts one. When it hits zero the packet is thrown away, which is what stops packets circling forever.
5. You can read hop count from it. Most systems start replies at 64. Seeing 57 means about 7 machines forwarded it on the way back.
6. time=8.412 ms is the round trip: out and back, in thousandths of a second. It is not the one-way time and you cannot halve it reliably, because the two directions may take different routes.
7. 0.0% packet loss is the fraction that never came back.
8. The last line is the spread. A small difference between minimum and maximum means a steady path. A large one means the path is congested or wireless is struggling.

■ 29.7.4 PLAIN – what is really happening inside

1. Ping works but the site does not. Why this is normal.
2. The machine answering ping may not be the machine running the site. Large services put a shared address at the front, and the thing that answers a simple “are you there” is not the thing that serves pages.
3. The site’s software may have crashed while the machine itself is fine.

4. The port you need may be closed or filtered while the simple message type is allowed.
5. Ping fails but the site works. Why this is also normal.
6. Very many networks are configured to ignore this message type, on purpose, to reduce noise and to make scanning harder.
7. Routers treat it as low-priority work handled by their slow general-purpose processor, while real traffic goes through fast dedicated hardware. Under load, the simple message is dropped first.
8. Some large cloud platforms block it by default. On Amazon Web Services, a new security group does not permit it unless you add a rule. Traffic through several kinds of cloud load balancer does not answer it at all.
9. The practical rule: use ping to measure a path you already know works, and never use its failure alone as evidence.
10. And how to read loss. One or two percent on a wireless link is common and usually harmless for web browsing. Ten percent will make everything feel broken. A hundred percent means either a real break or a policy of silence, and you cannot tell which from ping.
11. The most misleading case: loss shown at a middle hop of a trace but not at the end. That is almost always the middle router deprioritizing replies to itself, not loss on the path. If the final destination shows no loss, the path has no loss.

■ 29.7.5 TECHNICAL – the engineer’s version

1. Ping uses the Internet Control Message Protocol, specified by Jon Postel in RFC 792 in September 1981. Echo Request is type 8, code 0. Echo Reply is type 0, code 0.
2. RFC 1122, the host requirements document from October 1989, states that every host must implement an ICMP Echo server. Blocking it is therefore a deliberate deviation from the standard, and a very common one.
3. Mike Muuss wrote the original ping in December 1983 at the United States Army Ballistic Research Laboratory, naming it after sonar.
4. The default payload is 56 bytes, making a 64-byte ICMP message and an 84-byte IPv4 packet including the 20-byte IP header.
5. The first 8 bytes of payload on BSD systems carry a timestamp, which is how the round-trip time is computed without keeping state.
6. Initial TTL values by system, useful for inferring hop counts:

System	Initial TTL
Linux, macOS, BSD	64
Windows	128
Cisco IOS, Solaris	255

7. ICMP messages you should recognize when they do appear:

Type and code	Meaning
0/0	Echo reply
3/0	Network unreachable
3/1	Host unreachable
3/3	Port unreachable
3/4	Fragmentation needed
3/13	Administratively prohibited

Type and code	Meaning
11/0	Time to live exceeded

8. Type 3 code 13 is a filter announcing itself. Type 3 code 4 is a path maximum transmission unit problem and causes connections that establish and then stall on the first large transfer.
9. Type 11 code 0 is the message traceroute depends on entirely.
10. Rate limiting is standard practice. Cisco IOS by default limits generation of ICMP unreachables to one every 500 milliseconds. Linux uses `net.ipv4.icmp_ratelimit`, default 1000 milliseconds, with a bitmask controlling which types are limited.
11. For IPv6 the message set is ICMPv6, RFC 4443 from March 2006, and filtering it wholesale breaks path MTU discovery and neighbour discovery. RFC 4890, May 2007, gives explicit guidance on which ICMPv6 types must be permitted.
12. Ping proves ICMP reachability of an address. It does not prove TCP or UDP reachability, service health, or that the responder is the service host, because anycast and load balancers separate the two.

■ 29.7.6 WORDS – remember these

1. ICMP — the network’s own message system — Internet Control Message Protocol, RFC 792, carrying errors and diagnostics rather than user data.
2. Echo request and reply — “are you there” and “yes” — ICMP types 8 and 0, what ping sends and receives.
3. TTL — how many machines a packet may pass — an 8-bit hop counter decremented by each router, discarded at zero, called hop limit in IPv6.
4. Round-trip time — out and back, in milliseconds — measured from a timestamp embedded in the echo payload.
5. Jitter — how much the timing wobbles — variation in round-trip time, shown as the standard deviation in ping’s summary line.
6. Control plane policing — routers protecting their own brain — rate limits on traffic addressed to the router itself, the reason middle hops show loss.

29.8 The commands in full, for macOS

■ 29.8.1 PLAIN – in simple words

1. A command is a word you type into the Terminal application, sometimes followed by short options that begin with a dash.
2. Those options are called flags. Each flag changes one small thing about how the command behaves.
3. You do not need to memorize them. You need to know which command answers which question, and about three flags for each.
4. The rest of this section is a reference. Read it once, then come back to it.
5. Everything here is for macOS. Where Linux differs, it says so, because the differences are exactly the kind that waste an afternoon.

■ 29.8.2 PLAIN – a picture in your head

1. Think of a mechanic’s toolbox with a dozen tools, not one big machine.
2. A torch shows you whether a thing is there. A meter measures. A stethoscope listens without touching anything.

3. Nobody uses a torch to measure voltage. Each tool answers one question.
4. ping is the torch. curl is the meter. tcpdump is the stethoscope.
5. Picking up the right one first is most of the skill.

Where this comparison breaks: mechanics' tools do not change what they are measuring. Some network tools do. A packet capture can slow a machine, and a port scan can trip a security system into blocking you.

■ 29.8.3 PLAIN – a worked example

1. One question, five tools, five different answers.

```
ping -c 3 20.207.73.82      # does it answer hello
traceroute -n 20.207.73.82  # how far do probes get
nc -vz -G 5 20.207.73.82 443 # does the port open
curl -sSI https://github.com # does the site answer
sudo tcpdump -n host 20.207.73.82 # what is on the wire
```

2. In the reader's case, the fourth and third both fail with silence, the second stops at hop 12, and the fifth would show outgoing SYN packets with nothing coming back at all.
3. That last observation, from the packet capture, is the one that converts "it does not work" into "we send, they never reply".

■ 29.8.4 PLAIN – what is really happening inside

1. These tools split into four groups, and knowing the group tells you what a result is worth.
2. Group one asks the operating system what it thinks: ifconfig, networksetup, scutil, netstat, route, arp, lsof. They never send a packet. They report configuration and state.
3. Group two sends probes and interprets replies: ping, traceroute, mtr, nmap.
4. Group three does real work with a real protocol: dig, nc, curl.
5. Group four watches without taking part: tcpdump.
6. Group one can be wrong about reality. It shows intention, not outcome.
7. Group four cannot be wrong about what left your machine, which is why it settles arguments.
8. When groups three and four disagree with group one, believe four, then three, then two, then one.

■ 29.8.5 TECHNICAL – the engineer's version

1. The command table. One line each on what the command proves.

Command	What it proves
ifconfig	Link state and addresses
ipconfig getifaddr en0	An IPv4 address is held
ipconfig getpacket en0	What DHCP actually gave
networksetup	Service-level configuration
scutil --dns	Which resolvers apply
netstat -rn	Routing table contents
route get	Which route a target uses
arp -a	Layer 2 neighbour resolution
ping	ICMP reachability only
traceroute	Where probe replies stop
mtr	Loss and latency over time
dig	DNS resolution, per server
nslookup	DNS, older interactive tool

Command	What it proves
host	DNS, short one-line answers
nc -vz	TCP or UDP port reachability
curl -v	Full TCP, TLS and HTTP path
lsof -i	Which process owns a socket
tcpdump	What actually left the machine
nmap	Port state across a range

2. ping flags that matter on macOS.

Flag	Meaning
-c n	Stop after n packets
-i s	Seconds between packets
-s n	Payload bytes, default 56
-t s	Give up after s seconds
-W ms	Per-reply wait, milliseconds
-D	Set the do-not-fragment bit
-n	No reverse name lookups

- Linux difference, and it bites: on Linux -t sets the TTL, not a timeout; the deadline flag is -w in seconds, and -W is a per-reply timeout in seconds rather than milliseconds. Intervals below 0.002 seconds require superuser rights on macOS.
- traceroute flags on macOS: -I uses ICMP echo instead of UDP; -P proto selects UDP, TCP, GRE or ICMP; -p port sets the base port, default 33434; -n prints numbers only; -q n sets probes per hop, default 3; -w s sets the wait, default 5 seconds; -m n sets maximum hops, defaulting to the system TTL of 64; -f n sets the starting hop.
- Linux difference: the common Linux traceroute takes -T for TCP and -U for UDP, defaults to 30 hops, and ships alongside tracepath.
- dig options: +short prints only the answer; +trace walks the hierarchy from the root itself; @server queries a named server directly; -x addr does a reverse lookup; +tcp forces TCP instead of UDP, which is how you test large answers and firewalls that only allow UDP port 53.
- nc flags: -v reports the outcome, -z connects and sends no data, -w s sets an idle timeout, -G s sets the TCP connection timeout on macOS, -u uses UDP, -n skips lookups, -4 and -6 force a family.
- UDP with nc -uvz is close to meaningless as a test, because a silent UDP port and an open one look identical. Only an ICMP port-unreachable reply distinguishes them, and that is often filtered.
- curl flags: -v shows the connection steps; -I sends HEAD; -s silences the meter; -S keeps errors visible; --resolve host:port:addr pins an address without DNS; --connect-timeout s bounds the TCP phase; --max-time s bounds everything; -w fmt prints timing fields; -4 and -6 force an address family.
- curl exit codes worth recognizing:

Code	Meaning
6	Could not resolve host
7	Failed to connect
28	Operation timed out

Code	Meaning
35	TLS handshake failure
52	Empty reply from server
60	Certificate not trusted

11. Code 6 is a rung 4 fault. Code 7 and 28 are rung 5. Codes 35 and 60 are rung 6. The exit code alone places the fault on the ladder.

12. Timing breakdown in one command:

```
curl -o /dev/null -sS -w \
'dns %{time_namelookup}\ntcp %{time_connect}\n\
tls %{time_appconnect}\nttfb %{time_starttransfer}\n\
total %{time_total}\n' \
https://github.com
```

13. Each field is cumulative from the start, so subtract to get phase durations. A large gap between tcp and tls is a TLS problem; a large gap between tls and ttfb is a server-side problem.

14. macOS-specific state commands: `scutil --dns` shows the ordered resolver list including per-domain entries installed by VPN software; `ipconfig getpacket en0` prints the full DHCP reply with lease time, router and domain_name_server; `route -n get default` names the winning interface; `networksetup -listallhardwareports` maps en0 to Wi-Fi or Ethernet, which differs between Mac models.

15. `tcpdump` needs `sudo`. Useful invocations:

```
sudo tcpdump -i en0 -n host 20.207.73.82
sudo tcpdump -i en0 -n port 443 and host 20.207.73.82
sudo tcpdump -i en0 -n -w /tmp/gh.pcap host 20.207.73.82
```

16. `-i` picks the interface, `-n` stops name lookups, `host` and `port` are filter expressions in the Berkeley Packet Filter language described by Steven McCanne and Van Jacobson at the USENIX conference in winter 1993, and `-w` writes a capture file you can send to somebody.

17. `mtr` and `nmap` are not shipped with macOS. Install with `brew install mtr nmap`. `mtr` needs `sudo` for ICMP mode; `mtr --tcp --port 443 github.com` probes with TCP instead. `nmap -Pn -p 443 --reason github.com` reports why it reached its conclusion, which is the single most useful `nmap` flag for diagnosis.

18. Linux equivalents, briefly: `ip addr` for `ifconfig`, `ip route` for `netstat -rn`, `ip neigh` for `arp -a`, `ip route get` for `route get`, `ss -tunap` for `lsof -i`, and `resolvectl status` for `scutil --dns` on systems using `systemd-resolved`.

■ 29.8.6 WORDS – remember these

1. Flag — a short option after a command — a switch, usually one dash and a letter, that alters one behaviour.
2. HEAD request — headers only, no body — the HTTP method `curl -I` sends.
3. Exit code — the number a command leaves behind — readable with `echo $?`, and for `curl` it names the failing layer.
4. Berkeley Packet Filter — the language for choosing packets — the expression syntax used by `tcpdump`, from a 1993 USENIX paper.
5. Homebrew — the usual way to install extra tools on a Mac — a package manager providing `mtr`, `nmap`, `bind` and much else.

6. Reverse lookup — number to name — `dig -x` or `host`, using the `in-addr.arpa` tree, and frequently absent or wrong.

29.9 The decision tree for “I cannot reach a site”

■ 29.9.1 PLAIN – in simple words

1. A decision tree is a list of steps where each answer sends you to a different next step.
2. The point is that you stop deciding what to do next. The tree decides.
3. That matters because fault finding goes wrong when you are annoyed and start guessing.
4. Follow it in order. Do not skip ahead because you have a hunch.
5. Every branch ends either in a cause, or in a clear instruction about who to contact and what to tell them.

■ 29.9.2 PLAIN – a picture in your head

1. It is the flowchart on the wall of a repair shop.
2. Junior staff follow it and get the same answer a senior person would.
3. The chart is not clever. It is just written down, so nobody forgets step four when they are in a hurry.
4. Your own memory under pressure is the unreliable part, not your knowledge.

Where this comparison breaks: a repair chart assumes one machine with fixed parts. A network path is shared with strangers and changes shape between two steps, so a tree can send you down a branch that was correct thirty seconds ago and is wrong now. Re-run the step that surprised you before you trust it.

■ 29.9.3 PLAIN – a worked example

1. The procedure, written to be followed without thinking.

```

1 Load a different site (example.com).
  works -> go to 2
  fails -> go to 10
2 dig +short github.com
  gives an address -> go to 3
  fails -> go to 12
3 nc -vz -G 5 <that address> 443
  connects -> go to 4
  refused -> service or port issue, go to 8
  silence -> go to 5
4 curl -sSI https://github.com
  works -> the fault was the browser or a cache
  TLS error -> certificate or clock, go to 9
5 Test from mobile data or another network.
  works there -> path fault, go to 6
  fails there too -> go to 8
6 Test another device on your own Wi-Fi.
  also fails -> not your Mac, go to 7
  works -> fault is on your Mac, check utun and firewall
7 traceroute -n and TCP traceroute to port 443.
  record where it stops, then go to 11
8 Check the service status page and an outage tracker.
  confirmed outage -> stop, wait, nothing to fix
  nothing reported -> go to 11
9 Check date and time, then the certificate chain.
10 Restart Wi-Fi, then check rung 1 and rung 2 commands.
```

```

still fails -> local network or router fault
11 Log the fault for an hour with the script in 29.12.
    then contact the ISP with the evidence in 29.10
12 dig @1.1.1.1 and dig @8.8.8.8 for the same name.
    one works -> your configured resolver is the fault
    both fail -> port 53 is blocked or routing is broken

```

■ 29.9.4 PLAIN – what is really happening inside

1. Step 1 is first because it has the highest eliminating power for two seconds of effort. It clears your machine, your Wi-Fi, your router and your provider's access network in one go.
2. Step 2 is second because naming failures produce confusing symptoms everywhere else, so you remove them early.
3. Step 3 is the bisection point of the whole tree. Above it, everything is application. Below it, everything is network.
4. Step 3 has three outcomes, not two, and the difference between refused and silent is the single most valuable distinction in the tree.
5. Step 5 exists to close the "is it down" question before you spend an hour on your own network.
6. Steps 10 and 12 are the two dead ends that you can actually fix yourself.
7. Step 11 is the honest ending. Sometimes the correct output of a diagnosis is a well-evidenced report to somebody with access you do not have.

■ 29.9.5 TECHNICAL – the engineer's version

1. Dead ends and the correct action at each.

Dead end	Action
Refused on 443	Retest by IP and by name
Silent on 443, mobile OK	Collect trace, report to ISP
Both resolvers fail	Test port 53 with +tcp
No default route	Renew DHCP, check utun
Works on other device	Audit local firewall, VPN
Status page confirms	Stop, no action available

2. Stop and contact the internet provider when all of the following hold: your local chain passes, another destination on the same network works, DNS resolves correctly, a connection to the target port times out with no reply, the same target succeeds from another provider, and the condition persists across at least twenty minutes of logging.
3. What to give them so it is not dismissed: your account or circuit identifier, your public address at the time from `curl -s ifconfig.me` or similar, the target address and port, a numeric traceroute, the exact `curl` or `nc` output including the timeout, a coordinated universal time stamp, and one sentence stating that the same target works from a different provider at the same moment.
4. Phrase the request as a specific technical question. "Please check for a routing or filtering issue between AS-of-yours and 20.207.73.82 on TCP 443" gets escalated. "The internet is not working" gets a modem reboot script.
5. Ask them explicitly to check the return path towards your prefix, not only the forward path. Return-path faults are invisible from your side and are a common cause of exactly this symptom.
6. If they claim it is a customer-side issue, the counter-evidence is the working test to a different destination on the same line, which proves your line carries traffic.

■ 29.9.6 WORDS – remember these

1. Decision tree — a written list of if-this-then-that steps — a diagnostic procedure with defined branch conditions and terminal states.
2. Dead end — a branch where you personally can do no more — a terminal state whose action is escalation rather than repair.
3. Escalation — passing the fault to whoever can act — a handover accompanied by evidence sufficient to reproduce the finding.
4. Circuit identifier — the number your provider knows you by — the account or service identifier that lets a technician find your line.
5. Prefix — a block of addresses — a range written as an address plus a slash and a length, such as 20.192.0.0/10.

29.10 Writing up a fault so someone else can act

■ 29.10.1 PLAIN – in simple words

1. A report is not a complaint. It is a set of instructions for reproducing what you saw.
2. Four things make a report work: what you did, what you expected, what happened, and what you have already ruled out.
3. The last one is the one people leave out, and it is the one that saves the most time.
4. “The internet is broken” gives the reader nothing to check, so they send you the standard script: restart the router.
5. A traceroute plus a command output gives them something to paste into their own tools, so it goes to somebody technical.
6. Keep every line copy-and-pasteable. Never describe output in your own words when you could paste it.

■ 29.10.2 PLAIN – a picture in your head

1. Two people go to a mechanic.
2. The first says “it makes a funny noise sometimes”.
3. The second says “at 40 kilometres per hour, in third gear, on a left turn only, there is a knock from the front right, and it started on Tuesday”.
4. The second person gets a fix. The first gets a bill for an hour of guessing.
5. The information in the second sentence is exactly the information a good fault report contains: conditions, location, timing, and what does not trigger it.

Where this comparison breaks: a mechanic can drive the car. Your reader cannot sit at your desk, so anything you did not write down does not exist.

■ 29.10.3 PLAIN – a worked example

1. A template you can fill in.

```
Subject: TCP 443 to 20.207.73.82 times out from my line
```

WHAT I DID

```
From macOS on my home connection, 13 Aug 2026, 09:40 UTC:  
curl -v --max-time 15 https://github.com
```

WHAT I EXPECTED

```
A TCP connection, then an HTTP 200 response.
```

WHAT HAPPENED

```
* Trying 20.207.73.82:443...
curl: (28) Failed to connect after 15001 ms
No RST, no ICMP unreachable. Total silence.
```

WHAT I HAVE RULED OUT

- DNS: dig +short github.com -> 20.207.73.82, in 12 ms
- Local network: ping 192.168.0.1 -> 0% loss
- Internet access: https://example.com loads normally
- Service down: same URL loads instantly on mobile data
- My device: same failure from a second device on this Wi-Fi

EVIDENCE

```
traceroute -n reaches 104.44.20.42 (hop 12) then * * *
My public address at the time: <address>
Circuit / account: <identifier>
```

WHAT I AM ASKING

Please check for a routing or filtering problem towards 20.207.73.82 on TCP 443, including the return path to my prefix. Happy to run any command you name.

■ 29.10.4 PLAIN – what is really happening inside

1. Why this format works is not politeness. It is about who can act.
2. A first-line support agent has a script and a set of buttons. Your report must contain something their script recognizes, or something so specific that the script does not fit and they must pass it on.
3. A network engineer needs three things to look at their own tools: a source, a destination, and a time. Give all three or they cannot look.
4. The “ruled out” list is doing a specific job. It pre-empts the four questions they would otherwise ask you, each costing a day of email.
5. Pasting exact output matters because your summary may hide the detail that identifies the fault. “It did not work” and “no RST, no ICMP” are wildly different findings.
6. State facts and separate them from your theory. Put any theory in one line at the end, marked as a guess. Reports that assert a cause get argued with; reports that present evidence get investigated.

■ 29.10.5 TECHNICAL – the engineer’s version

1. Minimum evidence set for a reachability report:

Item	Why it is required
Source public address	Identifies your prefix
Destination address	Identifies the target
Port and protocol	Narrows any filter
UTC timestamp	Lets them read logs
Numeric traceroute	Shows last responding hop
curl or nc output	Shows the failure mode
Control test result	Proves the line works

2. Capture the public address at the time of the fault, since it may change: `curl -s https://ifconfig.me` or `dig +short myip.opendns.com @resolver1.opendns.com`.
3. Always paste numeric output. `traceroute -n` avoids reverse lookups that might themselves fail and confuse the reader.

4. Include one control test that succeeded. Without it, the first response will be a line test rather than a routing investigation.
5. Where a packet capture is appropriate, attach a file made with `sudo tcpdump -i en0 -n -w /tmp/gh.pcap host 20.207.73.82`. Retransmitted SYN packets with no response are unambiguous proof of a silent drop and are the strongest artefact you can supply.
6. Use coordinated universal time throughout and say so. Provider logs are kept in it, and a mismatch of five and a half hours makes your evidence unsearchable.
7. Keep the theory to one labelled line, for example: “Hypothesis, unproven: asymmetric routing or a policy filter between your network and AS8075.”

■ 29.10.6 WORDS – remember these

1. Reproduction steps — how to see it yourself — an exact command sequence that produces the reported behaviour.
2. Control test — the thing that worked — a comparison measurement proving the surrounding system is healthy.
3. Artefact — a file that carries evidence — a capture, a log or a saved output, attached rather than described.
4. First-line support — the scripted layer — the tier whose tools cannot see routing, whose job is to filter and escalate.
5. Coordinated universal time — the one clock everyone shares — the reference time zone used in logs, printed by `date -u`.

29.11 When it is not your problem

■ 29.11.1 PLAIN – in simple words

1. Sometimes the fault really is at the other end, or really is your provider, and the right action is to stop working and wait.
2. There are three places to look, and each one has a weakness you must know.
3. A status page is written by the company that owns the service.
4. An outage tracker counts complaints from members of the public.
5. Your provider’s own channels tell you about local work and known faults.
6. None of the three is proof. Together with your own tests, they are enough.

■ 29.11.2 PLAIN – a picture in your head

1. A status page is the shop’s own sign in the window saying “we are open”.
2. An outage tracker is a crowd outside the shop, some of whom came on the wrong day and some of whom wanted a different shop.
3. Your own test is walking to the door and pulling the handle.
4. Only one of those three actually touches the door.

Where this comparison breaks: a shop sign is updated by someone who can see the shop. A status page is often updated by a process that requires a human decision and an internal agreement that an incident exists, which can lag the outage by half an hour or more.

■ 29.11.3 PLAIN – a worked example

1. Three checks and what each is worth.

Source	Strength	Weakness
Official status page	Authoritative	Slow, coarse regions
Outage tracker	Fast, crowd-driven	Noisy, biased to big ISPs
Your own cross-check	Direct evidence	Only one vantage point

2. GitHub publishes a status page, hosted on Atlassian's Statuspage product, which Atlassian acquired in 2016. It reports per-component state such as Git Operations, API Requests, Webhooks, Actions and Pages.
3. That component split matters. In the reader's session plain git kept working while the web interface and the programming interface did not. A status page showing "Git Operations operational" and something else degraded would be consistent with that.
4. Downdetector, launched in 2012, plots reports per minute. A sharp spike across many providers means a service problem. A spike from one provider only means a provider problem.
5. If the tracker shows nothing and your cross-check from another network works, the problem is yours or your provider's. That is the common case.

■ 29.11.4 PLAIN – what is really happening inside

1. How to tell a regional problem from a personal one, in three moves.
2. Move one: another destination on the same line. Works means your line is fine and the fault is destination-specific.
3. Move two: the same destination from another provider. Works means the destination is fine and the fault is provider-specific or path-specific.
4. Move three: ask one other person on the same provider in the same city. If they fail too, it is regional and the provider will already have a ticket.
5. Combine them into a small truth table and the answer falls out without argument.

Same line, other site	Other line, same site	Verdict
Works	Works	Path or policy fault
Works	Fails	Service problem
Fails	Works	Your line or router
Fails	Fails	Wide outage

6. Two honest limits. A status page saying "all systems operational" never contradicts your own failed request, because the service may be fine and unreachable from you. And a tracker with no reports never proves an absence, because trackers under-report small providers and small countries.

■ 29.11.5 TECHNICAL – the engineer's version

1. Verification independent of anybody's page:

```
curl -sS -o /dev/null -w '%{http_code} %{remote_ip}\n' \
  --max-time 10 https://www.githubstatus.com
dig +short api.github.com
curl -sSI --max-time 10 https://api.github.com | head -3
```

2. Query the status host from your own machine and note the resolved address; if the status page itself is unreachable from your network, that is another data point about your path, not about the service.

3. For provider-wide issues, check whether the provider's own portal and speed-test host respond, and whether a public looking glass in your country can reach your prefix.
4. Autonomous system level tools separate provider from path: `whois -h whois.cymru.com -v 20.207.73.82` returns the announcing autonomous system, which for Microsoft is AS8075. Doing the same for your own public address names your provider's autonomous system, and that pair is what an engineer needs to investigate.
5. Outage aggregation has known limits: crowd trackers measure complaints, not reachability, so they lag by minutes, over-represent consumer services and large providers, and cannot see business-hours-only faults.
6. Where experts disagree: some operators argue public status pages should be automated from probes, others that human confirmation prevents false alarms and customer panic. The practical consequence for you is that a page can be green during a real outage, so never treat it as a veto over your own measurements.

■ 29.11.6 WORDS – remember these

1. Status page — the operator's own report — a published per-component state, updated by an internal incident process.
2. Outage tracker — a count of public complaints — a crowd-sourced signal that correlates with outages but does not measure them.
3. Regional fault — broken for a city or a provider — a failure whose blast radius is a geography or an autonomous system.
4. Truth table — every combination of results — a grid mapping test outcomes to conclusions, removing the need for judgement.
5. Whois — who owns this number — a directory service naming the holder of an address block or autonomous system number.

29.12 Intermittent faults, the hardest kind

■ 29.12.1 PLAIN – in simple words

1. An intermittent fault is one that comes and goes.
2. It defeats every test in this chapter, because every test is one shot.
3. You run a command, it passes, and you conclude the problem is gone. Ten minutes later it is back.
4. The only cure is to stop taking single measurements and start recording, continuously, with the time written next to every result.
5. Then the pattern appears: every evening, every hour, only under load, only after the router has been up for two days.
6. A pattern is worth more than a hundred one-shot tests, because it points at a mechanism.

■ 29.12.2 PLAIN – a picture in your head

1. A doctor cannot find your headache in a ten-minute appointment.
2. So they ask you to keep a diary for two weeks: time, what you ate, how much you slept.
3. The diary finds what the appointment could not, because the fault is a pattern in time, not a state in the room.
4. Network logging is that diary, written automatically once a minute.

Where this comparison breaks: a diary records your own experience, which is continuous. A one-minute network probe samples, so a fault lasting twenty seconds can hide completely between two samples. Sample faster than the fault.

■ 29.12.3 PLAIN – a worked example

1. A logging script. Save it as `netlog.sh`, then run `chmod +x netlog.sh` and `./netlog.sh`.

```
#!/bin/bash
TARGET=github.com
IP=20.207.73.82
LOG=$HOME/netlog.csv
echo "time,dns_ms,tcp,http,rtt_ms" >> "$LOG"
while true; do
    T=$(date -u +%FT%TZ)
    D=$(dig +short "$TARGET" | head -1)
    C=$(nc -z -G 5 "$IP" 443 >/dev/null 2>&1 && echo ok || echo fail)
    H=$(curl -sS -o /dev/null --max-time 10 \
        -w '%{http_code}' "https://$TARGET" 2>/dev/null)
    R=$(ping -c 1 -W 1000 1.1.1.1 2>/dev/null \
        | awk -F'time=' '/time={print $2+0}')
    echo "$T,$D,$C,$H:-none},${R:-loss}" >> "$LOG"
    sleep 60
done
```

2. Line by line. `#!/bin/bash` tells the system which interpreter runs the file. `TARGET`, `IP` and `LOG` hold the values you may want to change.
3. `echo "time,..." >> "$LOG"` writes a header row and appends rather than overwriting, because `>>` adds to the end.
4. `while true; do ... done` repeats forever until you press control and C.
5. `T=$(date -u +%FT%TZ)` records the moment in coordinated universal time in a sortable format such as 2026-08-13T09:40:12Z.
6. `D=$(dig +short "$TARGET" | head -1)` records what the name resolved to, keeping only the first answer, so you can see if the address changed.
7. The `nc -z -G 5` line tests the connection with a five-second limit and records ok or fail. Sending output to `/dev/null` throws away the chatter and keeps only the success or failure.
8. The `curl` line records the HTTP status code, or nothing if it failed, and `${H:-none}` substitutes the word none when the value is empty.
9. The `ping` line takes one measurement to a known-good address and `awk` extracts the number after `time=`, giving you a baseline. `${R:-loss}` writes loss when there was no reply.
10. `sleep 60` waits a minute. Lower it to 10 for a fast-moving fault, but be aware that frequent probes can look like scanning to some networks.
11. The result is a comma-separated file you can open in a spreadsheet, sort by time, and plot.

■ 29.12.4 PLAIN – what is really happening inside

1. Why a log beats a test. A test answers “is it broken now”. A log answers “when is it broken, and what else is true then”.
2. Look for three patterns in the file.
3. Time of day. Failures clustered in the evening point at congestion, since that is when a shared line is busiest.
4. Correlation with your own activity. Failures only during large downloads point at your router’s connection table filling up.
5. Uptime correlation. Failures starting a fixed time after every router restart point at a memory or table exhaustion problem in the router.
6. Also watch the resolved address column. If the address changes and the failures follow one particular address, the fault is specific to one front end at the destination, which is a completely different

report to write.

7. Continuous path monitoring is the companion tool. `mtr` sends probes repeatedly to every hop and shows loss per hop over time, which catches a flapping link that a single traceroute misses.

■ 29.12.5 TECHNICAL – the engineer’s version

1. Continuous path measurement:

```
sudo mtr -n --report --report-cycles 100 20.207.73.82
sudo mtr -n --tcp --port 443 --report-cycles 100 github.com
```

2. `--report` runs a fixed number of cycles and prints a summary table. `--report-cycles 100` sets that number. `--tcp --port 443` probes with TCP SYN packets to the port you actually care about, which is far more representative than ICMP when a filter is port-specific.
3. Read `mtr` loss columns from the bottom up. Loss at an intermediate hop that does not persist to the final hop is control-plane rate limiting, not path loss. Only loss that continues to the destination is real.
4. `mtr` was written by Matt Kimball in 1997 as Matt’s Traceroute and has been maintained by Roger Wolff since 1998 under the name My Traceroute.
5. Sampling theory applies. To characterize a fault of duration *D* with confidence, sample at an interval well below *D*. A 60-second interval will miss most 20-second events and will under-report their frequency.
6. Prefer recording raw results rather than pass or fail. Store connect time in milliseconds, not a boolean, because degradation precedes failure and only the numeric series shows the trend.
7. To run the logger unattended on macOS, use a `launchd` property list in `~/Library/LaunchAgents` rather than `cron`, because macOS may not run `cron` reliably under modern power management. On Linux, a `systemd` timer or a `cron` entry is the normal choice.
8. Correlate with local link quality on macOS using `sudo wdutil info` for Wi-Fi signal and noise figures. Apple deprecated the older `airport` command-line tool during macOS 14 and it stopped working in a 14.x update in 2024; treat that detail as version-dependent.
9. Apple added `networkQuality` in macOS 12 Monterey in 2021, which measures throughput and responsiveness under load in round trips per minute. Running it during a suspected congestion window is a useful second signal.

■ 29.12.6 WORDS – remember these

1. Intermittent fault — comes and goes — a failure whose probability of observation depends on when you sample.
2. Sampling interval — how often you measure — the gap between probes, which must be shorter than the shortest fault you hope to catch.
3. Flapping — a link going up and down repeatedly — instability that a single traceroute cannot detect but a continuous probe will.
4. Correlation — two things happening together — evidence of a shared cause, not proof of one causing the other.
5. Comma-separated values — a plain text table — one record per line with fields separated by commas, openable by any spreadsheet.
6. `Launchd` — the macOS way to run things on a schedule — Apple’s service manager, configured with property list files, replacing `cron` in practice.

29.13 What to do next in this specific situation

■ 29.13.1 PLAIN – in simple words

1. The reader's fault has a shape: naming works, the connection to one destination on port 443 dies in silence, and the same destination answers from another network.
2. Four hypotheses survive that shape, and no test done so far separates them.
3. So the next steps are chosen for one reason only: each one splits those four hypotheses further.
4. The order below is by cost. Cheap and informative first.
5. Do not skip to contacting the provider. Do it after step 6, with evidence, or you will be sent round the reboot loop.

■ 29.13.2 PLAIN – a picture in your head

1. You know the parcel gets as far as one particular sorting office and then vanishes.
2. Next you would send a second parcel to a different address served by the same office, and a third to the same address by a different route.
3. If the second arrives and the third does not, the office is fine and the address is the problem. If both fail, the office is the problem.
4. That is exactly what the steps below do, with addresses and ports.

Where this comparison breaks: you can watch a parcel leave. Here you must capture your own outgoing packets to know they left at all, which is why a packet capture appears in the list.

■ 29.13.3 PLAIN – a worked example

1. The ordered list.
2. Step 1. Confirm the failure is still happening and is not a memory. Run `curl -v --max-time 20 https://github.com` and keep the output.
3. Step 2. Test the same destination on a different port. Try 22 and 80. If one port opens and 443 does not, the destination is reachable and the fault is port-specific, which is a very different report.
4. Step 3. Test another Microsoft-hosted address, so that most of the path is the same but the machine at the end is not. If that works, the fault is specific to the one address rather than to the whole path.
5. Step 4. Run a TCP traceroute to port 443, which follows the same kind of packet as your real connection instead of the different kind ordinary traceroute uses.
6. Step 5. Capture packets while trying, to prove your machine really sent the connection request and really received nothing.
7. Step 6. If your router allows it, run a ping and a trace from the router itself. That removes your Mac and your Wi-Fi from the experiment entirely.
8. Step 7. Log for an hour with the script from the previous section, so you know whether it is constant or comes and goes.
9. Step 8. Contact the provider with the template from earlier in this chapter, and ask the specific questions listed below.

■ 29.13.4 PLAIN – what is really happening inside

1. Why step 2 matters. A filter usually targets a port. If port 22 opens to the same host and 443 does not, you have proved the route works and a policy is selecting on port.
2. Why step 3 matters. It separates "this path is broken" from "this one machine is not accepting me". Both look identical when you only test one address.

3. Why step 4 matters. Ordinary traceroute on a Mac sends a different kind of packet to unusual port numbers. Some networks treat that differently from real web traffic, which is why a trace can die where your connection would have succeeded, or the reverse.
4. Why step 5 matters. Until you have seen the outgoing packets, “nothing happened” also covers “my own machine never sent anything”, which a local firewall or a tunnel interface could cause.
5. Why step 6 matters. If the router fails the same way, every question about your Mac is closed, and the provider cannot blame your device.
6. Why step 7 matters. Providers respond very differently to “it failed once” and “it failed for 47 of 60 consecutive minute-by-minute probes”.

■ 29.13.5 TECHNICAL – the engineer’s version

1. The commands, in order.

```
# 2. different ports, same host
nc -vz -G 5 github.com 22
nc -vz -G 5 github.com 80
nc -vz -G 5 github.com 443

# 3. different address, same network path
OTHER=$(dig +short www.microsoft.com | tail -1)
nc -vz -G 5 "$OTHER" 443
curl -sSI --max-time 10 https://www.microsoft.com

# 4. TCP traceroute to the port that matters
sudo traceroute -n -P TCP -p 443 20.207.73.82
sudo nmap -Pn -p 443 --traceroute --reason 20.207.73.82
sudo mtr -n --tcp --port 443 --report-cycles 50 20.207.73.82

# 5. prove what left the machine
sudo tcpdump -i en0 -n host 20.207.73.82 and port 443
```

2. On the TCP traceroute: macOS traceroute -P TCP exists but its handling of the final hop is unreliable, because a SYN-ACK or RST from the destination is not always recognized as arrival. Prefer nmap --traceroute or mtr --tcp and treat the macOS built-in as a second opinion.
3. Expected capture signature for a silent drop: repeated outgoing SYN packets to 20.207.73.82:443 with increasing gaps of roughly 1, 2, 4, 8 and 16 seconds under the retransmission rules of RFC 6298, and zero inbound packets from that address. That is the artefact to attach to a ticket.
4. Compare against a working target in the same capture window by adding a second curl to a host that works; a capture containing both a successful and a failed handshake side by side is very hard to dismiss.
5. From the router, if it offers a diagnostics page or a shell, run its own ping and traceroute to 20.207.73.82. Many consumer routers expose exactly these two tools. A failure there removes the Mac, the utun interfaces and the local firewall from consideration in one step.
6. Questions to put to the provider, in this wording:

Ask	Why
Route to 20.207.73.82?	Checks forward path
Return route to my prefix?	Checks the invisible half
Any filter on TCP 443?	Checks policy
Known issue with AS8075?	Checks peering
Can you test from your core?	Removes your line

7. Also ask whether your line is behind carrier-grade network address translation and, if so, which public address you were mapped to at the timestamp you supply. Without that mapping, the far end cannot look you up at all.
8. If the provider confirms nothing and the fault persists, the remaining practical mitigations are a different resolver that returns a different front-end address, an IPv6 path if one can be obtained, or a tunnel that moves your egress to another network. Note that these are workarounds that avoid the fault; none of them fixes it or identifies it.

■ 29.13.6 WORDS – remember these

1. TCP traceroute — a trace that uses real connection packets — probes carrying TCP SYN to a chosen port, so filters that select on port are visible.
2. Egress — where your traffic leaves for the internet — the point and address from which your packets enter the wider network.
3. Capture signature — the recognizable shape in a packet capture — here, retransmitted SYNs with no inbound reply.
4. Carrier-grade NAT — many customers sharing one public address — provider-side address translation, which hides which subscriber a session belonged to.
5. Workaround — avoids the problem without fixing it — a change that restores service by routing around the fault, leaving the cause in place.

29.98 Common wrong ideas

1. Wrong: the site did not load, so the site is down. Right: you proved only that you could not reach it. “Down” is a claim about everybody, and one success anywhere refutes it in ten seconds.
2. Wrong: it worked on mobile data, so my Wi-Fi is the problem. Right: switching to mobile data changes about thirteen variables at once. It proves the destination is alive and nothing about which variable mattered.
3. Wrong: ping works, so the service is fine. Right: ping tests one message type to one address. The service uses a different protocol on a specific port and may be down while ping answers.
4. Wrong: ping fails, so the host is down. Right: very many networks drop ICMP on purpose, and routers deprioritize it. Confirm with a TCP test before concluding anything.
5. Wrong: the stars at the end of my traceroute show where the fault is. Right: silence at the end of a trace is normal. Routers commonly rate-limit or suppress the replies traceroute depends on.
6. Wrong: loss shown at hop 6 means the path is losing packets. Right: read the last hop. Intermediate loss that does not persist to the destination is control-plane rate limiting on that router.
7. Wrong: DNS resolved, so DNS is not involved. Right: DNS resolved to one address out of many possible ones. A different resolver may hand you a different front end that works.
8. Wrong: the status page says all systems operational, so it is my fault. Right: a status page reports service health, not your reachability, and it updates only after a human decides an incident exists.
9. Wrong: a timeout and a connection refused are both just failures. Right: refused means a machine answered and said no. A timeout means nothing answered at all. They point at completely different causes.
10. Wrong: I tested it three times, so I have three pieces of evidence. Right: three identical tests from one vantage point are one observation repeated. Evidence comes from changing a variable, not from repetition.

29.99 Chapter summary in 20 lines

1. Reachability is a property of a pair, a source and a destination, never of a destination alone. That one sentence prevents most wasted hours.
2. “The site is down” is a universal claim; “I cannot reach it” is about you. One success from anywhere refutes the first; your own failure proves the second.
3. The one-line test that separates them is a request to the same destination from a network that shares nothing with yours.
4. The reader ran that test with mobile data and the site loaded instantly, so GitHub was up and the fault was on the reader’s own path.
5. That switch changed at least thirteen things: provider, public address, autonomous system, forward path, return path, peering point, resolver, resolved address, address family, middleboxes, path MTU, device stack, time.
6. Only one of them needed to be the cause, so the test had huge power over one question and none at all over which difference mattered.
7. Good diagnosis is a controlled experiment: change one variable, state the hypothesis, and know the eliminating power of the test before running it.
8. The ladder orders the chain: interface, router, public address by number, name resolution, TCP port, full request, path discovery.
9. For every rung, know three things: what a pass proves, what a pass does not prove, and what a failure means. Most mistakes are over-reading a pass.
10. On the reader’s ladder, rungs 1 to 4 passed, rung 5 failed with silence, rung 6 never ran, and rung 7 stopped at hop 12 inside Microsoft’s Pune site.
11. DNS passing while TCP fails is a very informative pair, because the two used different destinations, protocols and ports; only path, address and port remain as candidates.
12. Silence is the least informative failure mode. No RST, no ICMP, nothing. Consistent with a silent drop on that path, not with the server being down.
13. The trace’s * * * from hop 13 proves nothing; ICMP time-exceeded is routinely rate-limited, as with the Cisco default of one per 500 ms.
14. Five cross-checks cover every component: other network, other device, other destination, remote shell, and a tunnel. Each eliminates a different family.
15. Ping is ICMP echo, RFC 792 from 1981, written by Mike Muuss in December 1983. It is the most over-trusted tool in networking; use it to measure.
16. TTL in a ping reply lets you count hops; time is a round trip and cannot be halved reliably; loss must be read at the final hop only.
17. On macOS know these flags: `ping -c -i -s -t -W, traceroute -n -q -w -m -P, dig +short +trace @server, nc -vz -G, curl -v -I -sS --resolve --max-time -w`. On Linux, `ping -t` means TTL instead, and `-W` is seconds.
18. A fault report needs what you did, what you expected, what happened and what you ruled out, with exact output, a UTC timestamp and a control test.
19. Intermittent faults need logging, not testing: a minute-by-minute log, `mttr` in report mode, and correlation with time of day and load.
20. The reader’s remaining steps are TCP traceroute to port 443, a second Microsoft address, a test from the router itself, a packet capture showing retransmitted SYNs with no reply, and a precise question to the provider about the route and the return path towards their own prefix.

How the Internet Was Built

30.0 What this chapter gives you

1. You will be able to tell the story of long-distance messaging from signal fires to fibre, in order, with correct names and years.
2. You will be able to explain packet switching, and say honestly who invented it, when, and why three people arrived at it separately.
3. You will be able to kill the most popular myth in computing: that the ARPANET was built to survive a nuclear war.
4. You will be able to say exactly what happened on 29 October 1969.
5. You will be able to explain what TCP/IP is, why it was split into two layers, and what happened on 1 January 1983.
6. You will be able to state the difference between the internet and the web clearly enough to correct a newspaper.
7. You will be able to explain the dot-com boom and bust with real dates, real numbers, and an honest lesson rather than a moral.
8. You will be able to describe the physical internet: submarine cables, satellites, exchange points, and where India sits in that picture.
9. You will be able to say who actually runs the internet, and why the answer is uncomfortable and interesting at the same time.
10. You will be able to read a modern network path, like the reader's own traceroute through Mumbai and Pune, and see the whole history in it.

30.1 Before electricity: fire, flags and towers

■ 30.1.1 PLAIN – in simple words

1. For most of human history a message travelled at the speed of a horse.
2. There was one exception, found very early: light travels much faster than a person, and a fire on a hill can be seen from the next hill.
3. A chain of fires can carry one fact across a country in minutes.
4. But a fire says only one thing: the agreed event has happened. It cannot say what, or how many, or when.
5. So the next step was to give the signal parts, so it could spell anything.
6. In France in the 1790s, Claude Chappe built towers with two big movable wooden arms on top. Each position of the arms meant a code.
7. A watcher in the next tower copied the shape, and the shape moved down the line. This was the first working long-distance message network in the world, and it ran for over sixty years.

■ 30.1.2 PLAIN – a picture in your head

1. Imagine a line of people on hilltops, each just able to see the next one through a telescope.
2. Each person holds two long sticks. Each stick can be held in several positions, so two arms give many different shapes.
3. Each shape is a number, and both ends of the line hold the same code book. Number 4,102 might mean “the enemy has retreated”.
4. Nobody in the middle knows what the message means. They only copy shapes. That is an idea we still use: relaying without understanding.

Where this comparison breaks: there is no second path. A single sleepy operator stops the whole country, and fog, rain and darkness stop everything. A modern network routes around a failure in milliseconds. This one could not route around anything at all.

■ 30.1.3 PLAIN – a worked example

1. Take the Paris to Lille line, the first real Chappe line, opened in 1794.
2. It ran roughly 230 km with about 15 stations, so about 15 km between towers. Sources differ a little on the distance, so treat it as approximate.
3. One symbol took about nine minutes to reach Lille. A full message took about 32 minutes in good weather.

Method	Paris to Lille	Note
Rider on horseback	About 2 days	Roads, weather
Chappe tower line	About 32 minutes	Daylight only
First symbol only	About 9 minutes	Head of message

4. In 1794 the line told Paris that Condé-sur-l’Escaut had been retaken from the Austrians, less than an hour after it happened.

■ 30.1.4 PLAIN – what is really happening inside

1. Strip away the wood and three ideas are already present.
2. **Encoding.** The message becomes symbols the channel can carry. The channel carries shapes, not sentences.
3. **Relaying.** No station talks to the far end. Distance is built out of many short hops between neighbours.
4. **A shared code book.** Both ends must agree in advance what the symbols mean, or the shapes are noise.
5. All three survive today. Encoding is now modulation, relaying is now routing, and the code book is now a protocol specification.
6. The fatal weaknesses were that it was a chain rather than a mesh, and that it carried one message at a time. Packet switching fixes exactly those two things, one hundred and seventy years later.

■ 30.1.5 TECHNICAL – the engineer’s version

1. Chappe’s first public demonstration was on 2 March 1791 between Brûlon and Parc , over 16 km, using synchronized pendulum clocks rather than arms.
2. The French Legislative Assembly backed a Paris to Lille relay line in 1792. The line entered service in 1794.
3. Throughput was about 2 to 3 symbols per minute, roughly 0.4 to 0.6 words per minute of plain text.

- The Chappe code book used 92 basic symbols taken two at a time, giving 8,464 coded words and phrases. That is compression: one symbol pair stands for a whole phrase.
- At its peak the French network had 556 stations covering about 4,800 km. It stayed in service until 1855, and one of its last messages reported the fall of Sebastopol. Claude Chappe died on 23 January 1805.

Property	Chappe line	Modern comparison
Raw rate	About 0.5 bit/s	1 Gbit/s home fibre
Hop distance	About 15 km	80 to 100 km per span
Availability	Daylight, clear	99.99 percent target
Topology	Single chain	Meshed, multipath

- The modern term for the tower operator's job is **regenerative repeater**: a device that decides what the symbol was and transmits a clean copy, rather than amplifying the noise along with the signal.

■ 30.1.6 WORDS – remember these

- Semaphore — signalling by the position of arms — visual signalling using a defined symbol alphabet on a mechanical indicator.
- Code book — a shared list of what each signal means — a codec or protocol specification agreed by both endpoints.
- Regenerative repeater — a station that redraws the signal cleanly — a repeater that decides the symbol and retransmits, avoiding noise build-up.

30.2 The electric telegraph

■ 30.2.1 PLAIN – in simple words

- Electricity moves along a wire almost instantly and does not care about fog, night or hills.
- In Britain, William Fothergill Cooke and Charles Wheatstone patented a telegraph in May 1837, using needles that swung to point at letters.
- In the United States, Samuel Morse patented a recording telegraph in 1837. His assistant Alfred Vail built much of the practical machinery.
- Morse and Vail also produced a code of short and long pulses, which we call Morse code.
- On 24 May 1844 Morse sent "WHAT HATH GOD WROUGHT" from the Capitol in Washington to Baltimore, over a line paid for by the United States Congress.
- Wires then spread fast, because railways needed them and newspapers wanted them. The hard part left was the ocean.
- The first Atlantic cable worked for about three weeks in 1858 and died. A working cable was completed on 27 July 1866, and from that day a message crossed the Atlantic in minutes instead of ten days.

■ 30.2.2 PLAIN – a picture in your head

- Think of a very long garden hose full of water, with a tap at one end.
- Open and close the tap quickly and a pattern of pulses comes out the far end. Agree what the patterns mean and you have a telegraph.
- Now make the hose two thousand miles long and lay it on the sea floor.
- Water pushed in at one end no longer arrives as a sharp pulse. It arrives smeared out and weak, because the hose absorbs and delays it.

5. Push the pulses closer together and they blur into each other, and the far end cannot tell them apart. That blurring sets how fast you can go.
6. And if you turn the tap on harder to fight the weakness, you can burst the hose. That is roughly what killed the 1858 cable.

Where this comparison breaks: water pressure is not voltage. A long insulated conductor sitting in seawater behaves like a chain of capacitors and resistors, which is why William Thomson, later Lord Kelvin, had to work out the mathematics before long cables could be made to work at all.

■ 30.2.3 PLAIN – a worked example

1. Morse code is not random. Common letters got short codes.

E	.
T	-
A	. -
S	. . .
O	- - -
SOS	. . . - - - . . .

2. A skilled operator ran at roughly 20 to 25 words per minute, so that famous four-word phrase took about 10 to 12 seconds to send.
3. Now compare that with the ocean.

Link	Year	Speed
Land line, good operator	1860s	20 to 25 words/min
First Atlantic cable	1858	About 0.1 words/min
Working Atlantic cable	1866	About 8 words/min

4. Queen Victoria's message to President Buchanan on 16 August 1858 was 98 words and took about 16 hours to send. Even that beat a ship by a week.

■ 30.2.4 PLAIN – what is really happening inside

1. A telegraph key is a switch. Pressing it connects a battery to the wire.
2. At the far end an electromagnet pulls a lever whenever current flows. Morse's design pressed that lever against moving paper, so the message was recorded.
3. On a long undersea cable the current does not switch cleanly. The cable stores charge along its whole length, like a very long capacitor.
4. A sharp pulse arrives as a slow rounded hump, and closely spaced humps overlap. That overlap is called **intersymbol interference**, and it is still the limiting problem in every fast link today, including fibre.
5. The 1858 cable failed because the chief electrician, Wildman Whitehouse, applied very high voltage to force a stronger signal and the insulation broke down.
6. The 1866 cable succeeded through better insulation and manufacture and, crucially, instruments sensitive enough to detect a faint signal. On a long link, improving the receiver beats shouting louder.

■ 30.2.5 TECHNICAL – the engineer's version

1. The Cooke and Wheatstone patent of May 1837 recommended five needles, coding 20 of the 26 letters directly.

2. A four-needle system was installed on the Euston to Camden Town section of the London and Birmingham Railway in 1837 and rejected in favour of pneumatic whistles. Their first commercial success was on the Great Western Railway in 1838, over the 21 km from Paddington to West Drayton.
3. Multiplexing arrived because copper and poles were the expensive part.

Technique	Roughly	What it gives
Duplex (Stearns)	1870s	2 messages, both ways
Quadruplex (Edison)	1874	4 messages, 2 each way
Baudot TDM	1874	Several ops share a wire

4. Émile Baudot's 1874 printing telegraph used **time-division multiplexing**: a rotating distributor gave each operator a slot on the same wire. Typical working speed was 30 words per minute.
5. The Baudot code was a fixed-length 5-bit code, unlike Morse's variable length code. Fixed length is what a machine wants; variable length suits a human ear.
6. Cable dates precisely: the 1858 cable was completed on 5 August 1858, first official message 16 August, failed in September after about three weeks. The 1865 attempt broke after 1,062 nautical miles had been paid out from the SS Great Eastern. The successful cable landed on 27 July 1866, from Foilhommerum in Ireland to Heart's Content in Newfoundland.

■ 30.2.6 WORDS – remember these

1. Morse code — dots and dashes for letters — a variable-length code optimized for letter frequency in English.
2. Multiplexing — sharing one wire between several messages — combining channels onto one medium by time, frequency or code.
3. Intersymbol interference — pulses smearing into each other — spreading of symbol energy into adjacent symbol periods, limiting symbol rate.

30.3 The telephone network

■ 30.3.1 PLAIN – in simple words

1. The telegraph sent letters. The next question was whether a wire could carry a voice.
2. Alexander Graham Bell was granted US patent 174,465 on 7 March 1876, and on 10 March 1876 made the first successful call, saying "Mr. Watson, come here, I want to see you".
3. A voice is not dots and dashes. It is a continuously changing signal, so the wire must carry a smooth copy of the sound.
4. At first every telephone needed a direct wire to every other one. Ten phones need forty-five wires. A hundred need 4,950.
5. The fix was an **exchange**: one wire from each phone to a central room, where someone joins two wires together.
6. Almon Strowger, an undertaker, believed operators were sending his calls to a rival, and built a machine to replace them. His patent was granted in 1891 and the first automatic exchange opened in La Porte, Indiana on 3 November 1892, for about 75 subscribers.
7. From that grew the largest machine humans had built, working in a way that could never have become the internet.

■ 30.3.2 PLAIN – a picture in your head

1. Think of a railway where the whole route is booked for one train.
2. Every section of track from your station to your destination is reserved before you leave, and nobody else may use any part of it.
3. That is **circuit switching**. It is wonderful for a phone call, because once booked the quality never wavers mid-sentence.
4. It is terrible for computers, which send a burst, think for four seconds, then send another burst.

Where this comparison breaks: from the 1960s, long-distance calls were carried digitally over shared links in time slots, so there was no continuous copper path. But the reservation model stayed: a slot was booked for the whole call whether you spoke or not. The reservation, not the copper, is the point.

■ 30.3.3 PLAIN – a worked example

1. Suppose you type at 5 characters per second to a remote machine. At 8 bits per character that is 40 bits per second of real information.
2. The circuit reserved for you is a digital voice channel, 64,000 bits per second.
3. You are using about 0.06 percent of it. The rest is reserved and wasted.
4. Now put 100 such users side by side.

Model	Reserved for 100	Actually used
Circuit switching	6,400,000 bit/s	4,000 bit/s
Packet switching	64,000 bit/s	4,000 bit/s

5. That table is the entire economic argument for the internet.
6. It is also why the telephone companies, who were excellent engineers, resisted the idea. Their business was selling reserved time.

■ 30.3.4 PLAIN – what is really happening inside

1. Lift a handset and you close a loop, so direct current flows from the exchange through your phone and back.
2. A rotary dial breaks that current once per unit of the digit. Dial 5 and it breaks the loop 5 times.
3. A Strowger switch counts the breaks with an electromagnet, stepping the wiper up five levels. The next digit rotates it to one of ten contacts, so two digits select one of a hundred outlets.
4. Nothing in that path stores your voice or looks at it. The network is a wire-joining machine, not a message-handling machine, and that is the deep reason it could not become the internet.

■ 30.3.5 TECHNICAL – the engineer's version

1. Elisha Gray filed a caveat for a liquid-transmitter telephone on 14 February 1876, the day Bell's application was filed. Gray did not contest priority, and the examiner approved Bell's patent on 3 March 1876.
2. The Bell Telephone Company was formed in 1877. By 1886 over 150,000 Americans had telephones.
3. Strowger conceived his switch in 1888 and received US patent 447,918 in 1891. The prototype used a collar box and pins. The two-motion selector gives 10 levels by 10 contacts, so 100 outlets.
4. Digital voice standardized on 8,000 samples per second at 8 bits, giving the 64 kbit/s DS0 channel. That sampling rate follows from the Nyquist rule applied to a 300 to 3,400 Hz voice band.

Level	Region	Rate
DS0	Worldwide	64 kbit/s
T1 (DS1)	North America	1.544 Mbit/s
E1	Europe, India	2.048 Mbit/s
T3 (DS3)	North America	44.736 Mbit/s

5. Signalling System No. 7 moved call setup onto a separate packet network while voice used reserved circuits. Note the detail: SS7 is itself packet based.
6. The Bell System was broken up by consent decree effective 1 January 1984, separating AT&T from the regional operating companies.
7. Stated technically, the mismatch is this: circuit switching gives constant bit rate, constant delay and zero statistical multiplexing gain. Computer traffic is bursty, often with a peak-to-mean ratio above 100 to 1, and gains enormously from statistical multiplexing.

■ 30.3.6 WORDS – remember these

1. Exchange — the building that joins calls — a switching centre terminating subscriber loops and interconnecting them.
2. Circuit switching — booking a path for the whole call — dedicating a fixed channel end to end for a session's duration.
3. DS0 — one digital voice channel — 64 kbit/s, 8,000 samples per second at 8 bits, the base unit of the digital telephone hierarchy.

30.4 Packet switching, invented three times

■ 30.4.1 PLAIN – in simple words

1. Packet switching is the idea everything on the internet rests on.
2. Instead of booking a path and sending a stream, you chop the message into small pieces.
3. Each piece carries the full destination address, like an envelope with a complete postal address on it.
4. Each piece travels on its own. Machines along the way read the address and pass it towards the destination.
5. Different pieces may take different routes, arrive out of order, or be lost. The receiving end reorders them and asks for anything missing.
6. Nobody books anything. The link is shared moment by moment between whoever has a piece ready to send.
7. Three people worked this out separately in the early 1960s, in two countries, for two completely different reasons. When an idea is genuinely ready, it tends to be found more than once.

■ 30.4.2 PLAIN – a picture in your head

1. You want to send a long book to a friend in another city.
2. Circuit switching is hiring a private van that drives non-stop from your door to theirs with the whole book inside.
3. Packet switching is tearing the book into single pages, putting each page in its own addressed envelope with a page number, and posting them all.
4. The envelopes mix with everyone else's post. Each sorting office moves each one closer. Some go one way, some another, and they arrive out of order.

5. Your friend sorts them by page number. If page 74 never turns up they ask for page 74 only, not the whole book.

Where this comparison breaks: the post has no congestion feedback. The internet does. When routers fill up and drop packets, senders slow down. That feedback loop is why your download speed changes second by second, and nothing in the postal picture explains it.

■ 30.4.3 PLAIN – a worked example

1. Take a 10,000-byte file on a network whose maximum packet size is 1,500 bytes, the common Ethernet limit.
2. It splits into 7 packets: six of 1,500 bytes and one of 1,000.
3. Each gets a header. A minimal IPv4 header is 20 bytes and a minimal TCP header another 20, so 40 bytes of addressing and control per packet.

```
File 10,000 bytes
+--> pkt 1..6 [hdr 40][data 1500]
+--> pkt 7    [hdr 40][data 1000]
On the wire: 10,000 + 7*40 = 10,280 bytes
Overhead:    280 / 10,280 = 2.7 percent
```

4. Overhead is 2.7 percent. That is the price of the whole idea, and it is small.
5. Now lose packet 4. Circuit switching has no answer, because it assumed the path was reliable.
6. Packet switching resends 1,540 bytes instead of 10,280. A 15 percent cost instead of a 100 percent cost.

■ 30.4.4 PLAIN – what is really happening inside

1. The three inventors reached the same design from different starting points, and it helps to keep them apart.
2. **Paul Baran**, at the RAND Corporation, worked from around 1960 on survivable communications. He proposed a distributed mesh with many alternative paths, messages cut into fixed **message blocks**, and store-and-forward switching with what he called hot potato routing.
3. **Donald Davies**, at the National Physical Laboratory in Britain, started in 1965 from a different problem: interactive computing was painful because telephone lines were slow to set up and wasteful when idle.
4. Davies coined the word **packet**, choosing a short word that would survive translation into other languages.
5. **Leonard Kleinrock**, at MIT and then UCLA, did the mathematics of queues: what happens to delay when messages wait in line at a busy node.
6. The credit question got heated later. The honest position is that historians and primary sources credit Baran and Davies with independently inventing packet switching.
7. Kleinrock's early published work was about message switching and queueing delay, and in 2023 he acknowledged that. Donald Davies wrote that he could find no evidence Kleinrock understood the principles of packet switching from that work, and Alex McKenzie of BBN said the 1964 book contains nothing analysing packetization.
8. That is a live disagreement between people who were there. State it as a disagreement, not as a settled insult in either direction.

■ 30.4.5 TECHNICAL – the engineer’s version

1. Baran’s work at RAND was first briefed to the US Air Force in summer 1961 as briefing B-265, published as RAND paper P-2626 in 1962 and as the report series RM-3420 in 1964. The full series, “On Distributed Communications”, ran to eleven volumes.
2. His three core ideas were a decentralized network with multiple paths between any two points, division of user messages into message blocks, and delivery by store-and-forward switching.
3. Davies conceived his design in 1965, proposed a national commercial data network publicly in 1966, and his colleague Roger Scantlebury presented the work at the Symposium on Operating System Principles in October 1967. The NPL Data Communications Network piloted in early 1969 and entered service in 1970.

Person	Where	Contribution
Paul Baran	RAND, USA	Distributed mesh, blocks
Donald Davies	NPL, UK	The packet, network design
Leonard Kleinrock	MIT, UCLA	Queueing analysis

4. Now the nuclear myth, stated precisely, because it needs care.
5. It is **true** that Baran’s funding context was US Air Force interest in communications that could survive attack, and his reports address that.
6. It is **false** that the ARPANET was built for that reason. ARPA director Charles Herzfeld stated plainly that the ARPANET was not started to create a command and control system that would survive a nuclear attack.
7. The ARPANET’s real motivation was resource sharing: expensive research computers were scarce, incompatible and scattered.
8. So the correct sentence is that one of the three independent inventions of packet switching had a survivability motive, and the network that first deployed it did not.
9. The model that matters in practice is Little’s Law: L equals λ times W , relating mean number in system, arrival rate and mean time in system. It is why a link driven near 100 percent utilization has unbounded queueing delay.

■ 30.4.6 WORDS – remember these

1. Packet — a small piece of a message with an address on it — a self-contained protocol data unit with header and payload.
2. Store and forward — receive it fully, then send it on — a node buffers the whole unit before retransmitting on the next hop.
3. Little’s Law — long queues mean long waits — L equals λ times W .

30.5 ARPANET

■ 30.5.1 PLAIN – in simple words

1. ARPA was a research funding agency inside the United States Department of Defense.
2. In October 1962 J.C.R. Licklider was put in charge of its computing office. He had already published “Man-Computer Symbiosis” in 1960.
3. On 23 April 1963 he sent a memo addressed to “Members and Affiliates of the Intergalactic Computer Network”. The joke name carried a serious idea: these separate machines ought to be able to talk to each other.

4. In 1966 Bob Taylor, then running that office, started the actual project. He was irritated by having three terminals on his desk for three remote computers.
5. He hired Larry Roberts to run it, and the work went out to tender in 1968.
6. A small company in Cambridge, Massachusetts called Bolt Beranek and Newman won the contract in January 1969.
7. They built a box called an Interface Message Processor, which sat between a university's computer and the network. On 29 October 1969 the first message was sent, and the receiving machine crashed after two letters.

■ 30.5.2 PLAIN – a picture in your head

1. Imagine four universities, each with one expensive and completely different computer, with different operating systems and tape formats.
2. So instead you put a small identical box next to each computer.
3. Every box is the same model from the same company running the same software, and the boxes talk to each other over leased telephone lines.
4. Each box talks to its own local computer in that computer's own language.
5. The universities never had to agree with each other. Each only had to agree with the box in its own machine room.

Where this comparison breaks: the box was not merely a translator. It did the routing, buffering, error checking and retransmission. It was the first router, and it did the hard part so the host computers could stay ignorant of it.

■ 30.5.3 PLAIN – a worked example

1. On 29 October 1969, at about 10:30 pm Pacific time, Charley Kline at UCLA tried to log in to the SRI machine over the new network.
2. To log in he had to type LOGIN, and the remote system echoed each character.

```

UCLA (SDS Sigma 7)           SRI (SDS 940)
| -- 'L' -----> |
| <----- echo 'L' |
| -- 'O' -----> |
| <----- echo 'O' |
| -- 'G' -----> |
|                                     [crash]

```

3. So the first message ever sent between two computers on the ARPANET was “LO”. The crash was in SRI's host software, not in the network, and a full login worked about an hour later.
4. The four original nodes, installed through the last months of 1969, were these.

Node	Host computer
UCLA	SDS Sigma 7
SRI (Stanford Research)	SDS 940
UC Santa Barbara	IBM 360/75
University of Utah	DEC PDP-10

■ 30.5.4 PLAIN – what is really happening inside

1. The Interface Message Processor, always called the IMP, was a ruggedized Honeywell DDP-516 minicomputer with 24 kilobytes of magnetic core memory.

2. Each IMP connected to its local host on one side and to two or more other IMPs on the other, over leased telephone lines.
3. Each IMP kept a table saying which neighbour was the best next hop for each destination, updating it as conditions changed. That is exactly what a router does today; the word did not exist yet.
4. Above the IMP layer, host computers spoke the Network Control Program, or NCP, which assumed the network underneath was reliable.
5. That assumption is the key difference from TCP/IP and is what eventually had to be thrown away. Note the split, though: identical switches doing the networking, wildly different hosts doing the applications. That is still the shape of the internet.

■ 30.5.5 TECHNICAL – the engineer’s version

1. The honest version of the dates: Licklider arrived at the Information Processing Techniques Office in October 1962, and the famous Intergalactic Computer Network memo is dated 23 April 1963. Use 1962 for his arrival and 1963 for the document.
2. The request for quotation went to 140 potential bidders in 1968 and 12 bid. BBN won, and the contract was awarded in January 1969.
3. The host-to-IMP interface was specified in BBN Report 1822, which is why engineers still call it 1822. Lines between IMPs ran at 50 kbit/s.
4. The public demonstration was at the International Conference on Computer Communications in Washington in October 1972, organized largely by Robert Kahn, with about 40 machines connected live in a hotel.
5. That demonstration matters more than it sounds. Until then, telephone engineers widely considered packet switching doubtful. Afterwards it was considered obvious.

Date	Rough scale
Dec 1969	4 nodes
1971	About 15 nodes
1973	First links outside USA
1981	213 hosts

6. In 1973 the ARPANET gained links to Norway, via NORSAR, and to University College London, making it international.
7. The tools that echo this era are still on your machine. `ping` measures round-trip time and `traceroute` shows the hop sequence, because a hop-by-hop store-and-forward network needs hop-level diagnostics.

■ 30.5.6 WORDS – remember these

1. ARPANET — the first packet switched network of this kind — the ARPA-funded research network, operational 1969 to 1990.
2. IMP — the box that did the networking — Interface Message Processor, a Honeywell DDP-516 acting as the first packet switch, specified in BBN Report 1822.
3. NCP — the early host-to-host rules — Network Control Program, the ARPANET host protocol before TCP/IP, which assumed a reliable subnet.

30.6 Email and the applications that came first

■ 30.6.1 PLAIN – in simple words

1. The ARPANET was funded to let people share expensive computers. What people actually did with it, almost immediately, was send each other messages.
2. Machines already had local mail: you could leave a note for another user of the same computer.
3. In 1971 Ray Tomlinson at BBN joined a local mail program to a file transfer program, so a note could be left on a different computer.
4. He needed a way to write “this user, on that machine”, and picked the @ sign because it was not used in user names and read naturally as “at”.
5. Email spread on its own, with nobody planning or funding it, and within a couple of years it was the bulk of the traffic.
6. That is the pattern to remember: the important application is usually not the one in the funding proposal.

■ 30.6.2 PLAIN – a picture in your head

1. Think of a research building where every office has a pigeonhole for internal notes. That works, but only inside the building.
2. Now someone realizes the courier who carries parcels between buildings could also carry notes.
3. So they write a note, put it in a parcel, and mark it for the pigeonhole of Dr Rao, in the Chennai building.

Where this comparison breaks: unlike a parcel, an email is copied rather than moved, so the sender still has it. That single difference is why email became a permanent archive, and why mailing lists were possible at all.

■ 30.6.3 PLAIN – a worked example

1. An email address has exactly two parts, split by the @ sign. In `tomlinson@bbn-tenexa`, the local part is `tomlinson` and the host part is `bbn-tenexa`.
2. Here is a modern message as it actually travels.

```
S: 220 mail.example.net ESMTP ready
C: EHLO client.example.org
S: 250 STARTTLS
C: MAIL FROM:<student@example.org>
S: 250 2.1.0 Ok
C: RCPT TO:<teacher@example.net>
S: 250 2.1.5 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: Subject: chapter 30
C: (blank line, then the body)
C: .
S: 250 2.0.0 Ok: queued as 4A2F1C
C: QUIT
```

3. Notice the envelope and the content are separate. `MAIL FROM` and `RCPT TO` are the envelope; the Subject line inside `DATA` is content.
4. Mail is delivered to the envelope address, not the `To:` header. That is why blind copies work at all.
5. That design is from RFC 821 in 1982, and the modern version is RFC 5321 from 2008. The shape has not changed.

■ 30.6.4 PLAIN – what is really happening inside

1. Three applications defined the first decade.
2. **Telnet** gave you a terminal session on a remote machine. **FTP**, the File Transfer Protocol, moved files; its first specification is RFC 114 from 1971. **Mail** began as a bolt-on to file transfer and then got its own rules.
3. MsgGroup started in 1975. Another early list, SF-Lovers, was about science fiction, and administrators argued about whether a defence-funded network should carry it. That argument about acceptable use returns in the 1990s and matters enormously.
4. In 1979 and 1980, Tom Truscott and Jim Ellis at Duke University, with Steve Bellovin, built something different: **Usenet**.
5. Usenet was not on the ARPANET. It ran over ordinary dial-up calls between Unix machines using UUCP, Unix-to-Unix Copy. Each machine phoned its neighbours and swapped new articles, so over days an article reached every participating machine in the world.
6. That is store-and-forward at the application level, with no central server, no live connection and no permission needed from anybody. It is where the words FAQ, flame war and spam entered common use.

■ 30.6.5 TECHNICAL – the engineer’s version

1. Tomlinson’s 1971 work combined SNDMSG, a local message program, with CPYNET, an experimental file copy program, on TENEX systems at BBN. The exact date is not recorded, only the year.

Function	Then	Now
Remote login	Telnet, RFC 854	SSH, RFC 4251
File transfer	FTP, RFC 959	HTTPS, SFTP
Mail transport	RFC 821 (1982)	RFC 5321 (2008)
Mail format	RFC 822 (1982)	RFC 5322 (2008)

2. Usenet was publicly established in 1980 at Duke University and the University of North Carolina at Chapel Hill. The first group was net.general.
3. The Usenet transport protocol over TCP is NNTP, RFC 977 from 1986, updated by RFC 3977 in 2006.
4. Email’s structural weakness was baked in early. SMTP has no authentication of the sender, and everything added since, SPF (RFC 7208), DKIM (RFC 6376) and DMARC (RFC 7489), is a patch over that original openness.
5. You can still watch the protocol directly with `openssl s_client -starttls smtp -connect mail.example.net:587`.

■ 30.6.6 WORDS – remember these

1. Email — a note left in a mailbox on another computer — a store-and-forward messaging system using SMTP for transport and RFC 5322 for format.
2. Envelope — the delivery instructions, separate from the letter — the SMTP MAIL FROM and RCPT TO commands, distinct from message headers.
3. Usenet — a worldwide notice board copied machine to machine — a distributed discussion system using flood-fill propagation over UUCP or NNTP.

30.7 TCP/IP and the flag day

■ 30.7.1 PLAIN – in simple words

1. By the early 1970s there was not one network but several, and they were all different.
2. The ARPANET ran over leased telephone lines, PRNET was a radio network in the San Francisco area, and SATNET went over satellite to Europe.
3. The question was how to join networks that disagree about everything.
4. Bob Kahn posed the problem and Vint Cerf worked on it with him from spring 1973. In May 1974 they published “A Protocol for Packet Network Intercommunication” in IEEE Transactions on Communications.
5. Their answer was not to make the networks agree. It was to put a common protocol on top of all of them, with boxes called gateways in between.
6. That is what the word internet means: internetworking, networks of networks. In 1978 the design was split into TCP and IP, and on 1 January 1983 the ARPANET switched to them in a single day.

■ 30.7.2 PLAIN – a picture in your head

1. Imagine shipping goods across a continent where each country has different trucks, roads and load limits.
2. The good solution is the shipping container. Agree only on the box.
3. IP is the shipping container: a standard box with an address, able to travel over any underlying network.
4. TCP is the paperwork saying how many boxes there are, in what order, and what to do if one is lost. Gateways are the border crossings.

Where this comparison breaks: a real container is never opened at the border, but an IP packet may be cut into smaller pieces if the next network cannot carry one that big. That is fragmentation, and it is one of the messier parts of the design. Also, no shipping firm would tolerate quietly losing containers. IP does, on purpose, leaving recovery to TCP.

■ 30.7.3 PLAIN – a worked example

1. Look at what happened on the reader’s own machine when it tried to reach `github.com`, which DNS resolved to `20.207.73.82`.

```

1 192.168.0.1          home router
2 172.31.0.17         ISP private core
3 137.97.29.249       ISP public edge
7 ...del01...msn.net  Microsoft, Delhi
8 ...bom01...msn.net  Microsoft, Mumbai
10 ...pnq21...msn.net Microsoft, Pune
13+ * * *             no reply
```

2. Hops 1 to 6 are inside the reader’s ISP. Hops 7 onwards are inside Microsoft’s own backbone. Two completely separate organizations.
3. Nothing there required them to run the same equipment, software or internal addressing. They only had to agree on the IP packet format and on how to exchange routes.
4. Then look at the failure. The connection to port 443 timed out after 15 seconds with no reply of any kind: no reset, no error, silence.
5. IP did its job by promising nothing. TCP did its job by retrying and then giving up. Correct layering does not mean success. It means each layer fails in a well-defined way.

■ 30.7.4 PLAIN – what is really happening inside

1. The original 1974 design was one protocol doing both jobs, addressing and reliability.
2. That was wrong for one important reason: not every application wants reliability. Digitized voice would rather lose a packet than wait for it.
3. So in 1978 the design was split. **IP** does addressing and forwarding and promises nothing. **TCP** sits on top and adds order, retransmission and flow control.
4. IP is deliberately **best effort**: it will try, it may fail, and it will not tell you.
5. This is the hourglass or thin waist model: many technologies below, many applications above, one narrow protocol in the middle. That narrow waist is why the internet survived every change underneath it, from 50 kbit/s leased lines to 400 Gbit/s optics, without rewriting the applications.
6. The companion idea is the **end-to-end principle**: put function in the endpoints, not the network, unless there is a strong reason. That is why routers are simple and your laptop is complicated. The telephone network did the opposite, and could not change.

■ 30.7.5 TECHNICAL – the engineer’s version

1. Cerf and Kahn, “A Protocol for Packet Network Intercommunication”, IEEE Transactions on Communications, volume 22 issue 5, pages 637 to 648, May 1974.
2. RFC 675, December 1974, by Cerf, Dalal and Sunshine, was the first detailed specification and is where “internet” appears as a contraction of internetwork.
3. Version 4, in 1978, split the Transmission Control Program into IP, connectionless, and TCP, connection-oriented. Jonathan Postel drove the split. That version number is why we have IPv4 and never deployed an IPv1, 2 or 3.
4. The core specifications settled as RFC 791 (IP) and RFC 793 (TCP), both September 1981, plus RFC 768 (UDP, 1980) and RFC 792 (ICMP, 1981).
5. A **flag day** is a cutover with no compatibility period: at a set instant everyone stops the old thing and starts the new one, with no fallback.
6. The NCP to TCP/IP flag day was 1 January 1983. It was announced years ahead, tested in stages during 1982, and enforced by turning NCP off. Sites that were not ready lost service.
7. It worked because the network had a few hundred hosts, one funding agency could compel every site, and every administrator was reachable by name.

Property	Jan 1983	Today
Hosts affected	A few hundred	Billions
Single authority	Yes, ARPA	No
Rollback plan	Turn NCP back on	None possible
Migration style	Flag day	Dual stack, decades

8. IPv6 proves the point. It was specified in RFC 1883 in 1995, updated as RFC 2460 in 1998 and RFC 8200 in 2017, and adoption is still partial in 2026, with Google’s measurement of IPv6-capable users in the range of roughly 40 to 50 percent in recent years.
9. The reader’s machine reported IPv6: (none), an ordinary result on many Indian home connections, which illustrates the point exactly.
10. Instead of flag days we use **dual stack**, running both protocols at once, with transition mechanisms such as NAT64 and 464XLAT. The cost is running two internets in parallel for thirty years.

■ 30.7.6 WORDS – remember these

1. Internetworking — joining different networks into one — providing a uniform service across heterogeneous networks via a common protocol.
2. IP — the addressed envelope — the connectionless best-effort internet layer protocol, RFC 791 for version 4 and RFC 8200 for version 6.
3. TCP — the part that makes sure everything arrives in order — a connection-oriented reliable byte stream with congestion control.
4. Flag day — everyone switches at the same instant — a cutover with no backward compatibility period.
5. End-to-end principle — keep the clever parts at the edges — place function in end hosts unless it cannot correctly be implemented there.

30.8 Other networks and the standards war

■ 30.8.1 PLAIN – in simple words

1. It is easy to think the internet was the only game in town. It was not.
2. Through the 1980s there were many networks, most of them bigger and better funded.
3. Telephone companies offered **X.25**, a public data network standard first published in 1976.
4. France gave away terminals for a service called **Minitel** and had millions of users while most of the world had none.
5. Universities that could not join the ARPANET built **BITNET** in 1981, and hobbyists built **FidoNet** from 1984, linking home bulletin boards by nightly phone calls.
6. CompuServe and America Online sold access to their own private worlds, with their own content and their own rules.
7. And an enormous international standards effort, **OSI**, was supposed to produce the one true networking suite. Governments mandated it, textbooks taught it, and TCP/IP beat it completely.

■ 30.8.2 PLAIN – a picture in your head

1. Imagine two teams designing a bicycle for the whole world.
2. Team B builds a rough bicycle, rides it, fixes the problems, and gives away both the plans and a working example for free.
3. Ten years later everyone rides Team B's bicycle, and Team A's document sits on a shelf being technically excellent.

Where this comparison breaks: OSI was not a failure of intelligence. Its vocabulary is still how we teach networking, and the seven-layer model is genuinely useful even though the seven-layer protocols lost. The failure was of process and timing, not of ability.

■ 30.8.3 PLAIN – a worked example

1. Compare the rivals of the 1980s side by side.

Network	Started	Fate
X.25 public networks	1976	Replaced, niche use
Minitel (France)	1982	Closed 30 June 2012
BITNET	1981	Support ended 1996
FidoNet	1984	Tiny, still alive

2. Minitel is the most instructive. France Telecom gave terminals away free from 1984, and by early 1986 there were 1.4 million connected.
3. By the end of 1999 there were close to 9 million terminals used by around 25 million people, with revenue of 620 million francs in 1985 and about 832 million euros in 1998.
4. The reason matters: Minitel was closed, with one operator deciding what services could exist. The web let anybody publish without permission.
5. FidoNet is the opposite lesson. With no money at all it peaked at around 39,000 systems in the mid 1990s and roughly 4 million users by 1993, using hierarchical addresses like 1:250/250.10.

■ 30.8.4 PLAIN – what is really happening inside

1. To understand why TCP/IP won, separate four advantages that people usually collapse into one.
2. **Running code.** TCP/IP existed and worked while OSI was still being written. In particular the 4.2BSD release of Berkeley Unix in 1983 shipped a usable TCP/IP stack with source, on the machines universities were already buying.
3. **No licence fees and no owner.** RFCs were free to read and free to implement. ISO standards had to be purchased, and still do.
4. **Government backing of the useful kind.** The US Department of Defense required TCP/IP on its networks, which created a large market immediately. OSI mandates promised a market later.
5. **Simplicity.** IP has one job. OSI's network layer had modes, options and variants, so two implementations could both be correct and still fail to interoperate.
6. The honest counter-argument, still made by experts: OSI had better answers for network management, addressing scale and formal description, and some of the pain of the 1990s came from TCP/IP's weaker versions of those. The better process beat the better committee, and we inherited both the benefits and the gaps.

■ 30.8.5 TECHNICAL – the engineer's version

1. X.25 was finalized by CCITT in 1976 as the Orange Book, revised in 1980, 1984, 1988, 1993 and 1996. It provided virtual circuits, typically at 2,400 bit/s to 64 kbit/s, rarely above.
2. BITNET was founded in 1981 by Ira Fuchs of CUNY and Greydon Freeman of Yale, first linking those two. It used IBM's RSCS and NJE protocols, peaked around 1991 at almost 500 organizations and about 3,000 nodes, and lost support in 1996. LISTSERV came out of that world.
3. FidoNet was created by Tom Jennings in San Francisco in early 1984, first released in June 1984. Zone Mail Hour was a reserved period, commonly around 04:00, when nodes exchanged mail and human callers were locked out.
4. OSI: ISO began the work in 1977; the reference model was published as ISO 7498 in 1984, after a merged CCITT and ISO text was agreed in May 1983.

OSI layer	Number	TCP/IP reality
Application	7	HTTP, SMTP, DNS
Presentation	6	Mostly in the app
Session	5	Mostly in the app
Transport	4	TCP, UDP
Network	3	IP
Data link	2	Ethernet, Wi-Fi, PPP
Physical	1	Copper, fibre, radio

5. The US Government OSI Profile, GOSIP, became a federal procurement requirement in 1990 and was effectively abandoned in 1994 and 1995 when agencies were allowed to buy TCP/IP instead.

6. ASN.1, an OSI-era notation, is everywhere in security: the X.509 certificates your browser checks on every HTTPS connection are defined in ASN.1 and encoded in DER.
7. So the accurate summary is not that OSI failed. It is that the OSI protocol stack lost and several OSI ideas won quietly.

■ 30.8.6 WORDS – remember these

1. X.25 — the phone companies' data network — a CCITT virtual-circuit packet network standard from 1976.
2. Walled garden — a service where one company decides what exists inside — a closed platform with controlled content and gated third-party access.
3. OSI — the official international networking suite — the ISO and CCITT protocol family and its seven-layer reference model, ISO 7498.

30.9 From research network to public internet

■ 30.9.1 PLAIN – in simple words

1. Until the late 1980s you could not simply buy internet access. There was nothing to buy.
2. You got on the internet by being at a university or research lab with a connection paid for by a government agency.
3. In 1986 the US National Science Foundation built **NSFNET** to link supercomputer centres and then regional university networks.
4. It came with an **acceptable use policy** saying the network was for research and education. Commercial traffic was not allowed.
5. So a national network existed and businesses were forbidden to use it. They built their own connections and their own exchange points instead.
6. Between 1991 and 1993 the restrictions were relaxed and then removed, and commercial providers appeared quickly.
7. On 30 April 1995 the NSFNET backbone was switched off. That date is a good candidate for the day the internet became a public utility rather than a research project.

■ 30.9.2 PLAIN – a picture in your head

1. Imagine a country with excellent motorways built only for university buses. Lorries are banned.
2. Eventually the private roads are better than the public one, the government closes its motorway, and everyone drives on the private roads.
3. The interchanges remain, and they are now the most important places on the map.

Where this comparison breaks: unlike a road interchange, two networks meeting at an exchange point may simply refuse to exchange traffic with each other. Peering is voluntary, and refusals happen, sometimes publicly and expensively.

■ 30.9.3 PLAIN – a worked example

1. Follow the NSFNET backbone as demand grew.

Year	Backbone	Speed
1986	6 sites, leased	56 kbit/s
July 1988	13 nodes	1.544 Mbit/s
Nov 1991	Upgraded	45 Mbit/s

Year	Backbone	Speed
Apr 1995	Decommissioned	Handed to industry

- Put a modern number beside it. DE-CIX Frankfurt, one exchange point, recorded a peak of 18.73 Tbit/s in December 2025.
- That is roughly 416,000 times the entire 1991 national backbone, in one building, in one city.
- That is why the acceptable use policy felt reasonable at the time. The capacity genuinely could not have absorbed commercial traffic.

■ 30.9.4 PLAIN – what is really happening inside

- The acceptable use policy was not a law. It was a condition attached to money, which is often stronger.
- Commercial networks that wanted to exchange traffic without touching NSFNET formed the Commercial Internet eXchange, CIX, in 1991.
- That is the first clear example of **settlement-free peering**, still how most large networks interconnect.
- In 1991 the NSF removed access restrictions, and on 23 October 1992 US legislation authorized the NSF to support networks used substantially for purposes beyond research and education.
- For the handover, NSF funded four **Network Access Points**, in New York, Washington DC, Chicago and San Jose. They were transitional and were replaced by hundreds of independent exchange points worldwide.
- The structural change is the important part. Before 1995 there was a backbone. After 1995 there is no backbone, only a mesh of networks that agree to carry each other's traffic.

■ 30.9.5 TECHNICAL – the engineer's version

- NSFNET began in 1986 with six backbone sites on leased 56 kbit/s links. The T-1 backbone, 1.544 Mbit/s across 13 nodes, was in place by July 1988, operated by Merit Network with IBM and MCI. The T-3 backbone at 44.736 Mbit/s was deployed in November 1991.
- The NSFNET Backbone Service transition completed and the fibre backbone was decommissioned on 30 April 1995.
- What replaced it is described by business relationship, not technology.

Relationship	Money flows	Routes given
Transit	Customer pays	Full table
Settlement-free peer	Nobody pays	Own customers only
Paid peering	One side pays	Own customers only

- Those relationships are enforced in BGP, the Border Gateway Protocol, currently RFC 4271. BGP decided the path the reader's packets took towards Microsoft.
- An autonomous system number, an ASN, identifies each independently routed network. Roughly 75,000 to 80,000 ASNs are visible in the global routing table in recent years, and the IPv4 table passed 1,000,000 routes in 2025.
- An exchange point is physically a large Ethernet switching fabric in a neutral building. Members take a port, commonly 10, 100 or 400 Gbit/s, and set up BGP sessions with whichever other members agree.
- DE-CIX Frankfurt had more than 4,300 connected networks by the end of 2025.

8. India's National Internet Exchange, NIXI, was registered on 19 June 2003 specifically so Indian-to-Indian traffic would stop being hauled via Singapore, London or the United States. It has reported peak traffic of about 393 Gbit/s inbound.
9. DE-CIX opened its first Indian exchange in Mumbai in 2018 and now operates in several Indian cities including Delhi, Chennai, Bengaluru, Hyderabad and Kolkata.

■ 30.9.6 WORDS - remember these

1. NSFNET — the American research backbone of the late 1980s — the NSF-funded TCP/IP backbone, 1986 to 30 April 1995.
2. Acceptable use policy — the rule saying what the network may be used for — a funding condition restricting traffic to research and education.
3. Peering — two networks swapping traffic directly — a BGP interconnection, usually settlement-free and carrying customer routes only.
4. Internet exchange point — a building where many networks meet — a shared layer 2 fabric enabling many-to-many BGP peering.
5. Autonomous system — one network under one routing policy — an independently administered routing domain identified by an ASN.

30.10 The World Wide Web

■ 30.10.1 PLAIN - in simple words

1. By 1989 the internet existed and worked, but there was no web.
2. To read a document on another machine you had to know it existed, know which machine held it, log in, find the directory and fetch the file.
3. Tim Berners-Lee was a software engineer at CERN, the particle physics laboratory near Geneva, where thousands of scientists came and went.
4. In March 1989 he wrote a proposal called "Information Management: A Proposal", suggesting a linked information system. He wrote a second in May 1990, and a formal one with Robert Cailliau on 12 November 1990.
5. By the end of 1990 he had written a server and a browser, both running on a NeXT computer, and the first website was live at `info.cern.ch`.
6. On 6 August 1991 he posted about the project publicly, and the software went out to the whole internet on 23 August 1991.
7. On 30 April 1993 CERN put the web software into the public domain, free for anyone to use with no fee and no permission. That may be the most valuable giveaway in the history of technology.

■ 30.10.2 PLAIN - a picture in your head

1. Think of a library with millions of books and no catalogue.
2. Now each book gains two things. First, a printed address on its spine that identifies it uniquely worldwide.
3. Second, wherever a book mentions another book, that mention becomes a button. Press it and the other book appears in your hands.

Where this comparison breaks: real links break. A book cannot vanish from your hands mid-sentence, but a web page can be deleted and the link becomes a dead end. Berners-Lee chose one-way links precisely because two-way links need permission from both ends, and requiring permission would have stopped the whole thing growing.

■ 30.10.3 PLAIN – a worked example

1. Take one address apart. Every part does one job.

```
https://info.cern.ch/hypertext/WWW/TheProject.html
  \____/  \_____/  \_____/
  scheme   host      path
```

```
GET /hypertext/WWW/TheProject.html HTTP/1.1
Host: info.cern.ch
```

```
HTTP/1.1 200 OK
Content-Type: text/html
```

2. The scheme says which protocol to speak, the host says which machine, and the path says which document on it.
3. The response starts with a number. 200 means here it is, 404 means no such document, 301 means it moved. The body is HTML: text with tags marking headings, paragraphs and links.

Invention	Job
URL	Name any document anywhere
HTTP	Ask for it and get an answer
HTML	Write it with links inside

4. None of the three was new on its own. The combination was: a global name, a trivial fetch, and a format that contains the next global name.

■ 30.10.4 PLAIN – what is really happening inside

1. The web is an **application** that runs on the internet. It is not the internet.
2. The internet is the delivery system: addresses, routing, packets. Email, video calls, games, software updates, DNS and messaging apps all use the internet without using the web at all.
3. Three design choices made the web spread faster than its rivals.
4. **Links are one way.** You may link to any page without telling the owner, so the graph grows with no coordination.
5. **Broken links are tolerated.** A 404 is a normal answer, not a failure of the system. Designs that guaranteed link integrity could not scale.
6. **It was free of ownership.** CERN's 1993 release meant no company could charge for the protocol or block a competing browser.
7. The honest version: the web was not the best hypertext system designed. It was much weaker than several research systems of the 1980s. It won because it was the one that required permission from nobody.

■ 30.10.5 TECHNICAL – the engineer's version

1. Protocol versions, with dates, because the differences matter in practice.

Version	Year	Key change
HTTP/0.9	1991	GET only, no headers
HTTP/1.0, RFC 1945	1996	Headers, status codes
HTTP/1.1, RFC 2068	1997	Persistent connections
HTTP/2, RFC 7540	2015	Multiplexing, binary

Version	Year	Key change
HTTP/3, RFC 9114	2022	Runs over QUIC and UDP

2. URLs were specified in RFC 1738 in 1994. The current generic URI syntax is RFC 3986 from 2005.
3. HTTP/3 uses QUIC (RFC 9000, 2021), which runs over UDP rather than TCP, because TCP's ordering guarantee causes head-of-line blocking when a single packet is lost.
4. To see the raw protocol yourself, run `curl -v` against any site. It prints the request and response headers exactly as described above.

■ 30.10.6 WORDS – remember these

1. Web — pages with links that you read in a browser — a distributed hypertext application layered on the internet.
2. URL — the full address of one document — a Uniform Resource Locator, syntax defined in RFC 3986.
3. HTTP — the request and answer language — the stateless application protocol for retrieving and submitting resources.
4. HTML — the way pages are written — a markup language describing structure and hyperlinks, now a living standard with no version number.

30.11 The browser wars

■ 30.11.1 PLAIN – in simple words

1. Early web browsers showed text and made you open images in a separate window.
2. In January 1993 Marc Andreessen and Eric Bina at the National Center for Supercomputing Applications released an alpha of a browser called Mosaic. Version 1.0 came on 21 April 1993.
3. Mosaic showed pictures inline with the text. That sounds small. It changed who wanted to use the web.
4. Andreessen left and, with Jim Clark, founded a company on 4 April 1994 that became Netscape. Its browser was released as Mosaic Netscape 0.9 on 13 October 1994 and took over the web almost immediately.
5. Netscape went public on 9 August 1995. Shares were offered at 14 dollars, doubled to 28 before trading opened, touched 75 on the first day and closed at 58.25, valuing a company with little revenue at about 2.9 billion dollars.
6. Microsoft responded by giving its browser away and putting it inside Windows. Netscape lost, was bought by AOL, and released its source code, which became Mozilla and eventually Firefox.
7. Then Google entered with Chrome in 2008 and won more completely than anyone before it.

■ 30.11.2 PLAIN – a picture in your head

1. Imagine two companies selling door locks for a new kind of house.
2. Company B also sells houses, and starts fitting its own lock to every house it builds, free and already installed.
3. Company A cannot compete with free, or with pre-installed, and goes out of business. Later, a third company that owns the road everyone drives on offers a new lock and mentions it on every journey.

Where this comparison breaks: locks do not improve when more people use them, but browsers do. Popularity made web authors test against a browser, which made it work better, which made it more

popular. That loop is why browser markets tip so hard toward one winner and are so difficult to reverse.

■ 30.11.3 PLAIN – a worked example

1. Follow the market share and notice how fast each turn happened.

Period	Leader	Rough share
1995 to 1997	Netscape	Majority
2001 to 2004	Internet Explorer	Over 90 percent
2010	Firefox and IE	Near parity
July 2026	Chrome	68.22 percent

2. The July 2026 worldwide figures from Statcounter page-view measurement were Chrome 68.22, Safari 16.47, Edge 5.37, Firefox 3.34, Samsung Internet 2.06 and Opera 1.88 percent.
3. Be careful with such numbers. They measure page views from participating sites, not installed copies, and different measurement firms disagree by several points.
4. Edge and Opera are built on the same engine as Chrome, so engine share is far more concentrated than browser share.

■ 30.11.4 PLAIN – what is really happening inside

1. A browser is several programs in one, and the important one is the **rendering engine**.
2. The rendering engine takes HTML, CSS and images and works out exactly what pixels go where. Beside it sit a JavaScript engine and a networking layer.
3. Building a rendering engine is enormously expensive. It must handle every badly written page on the web and match the leaders bug for bug. That is why the number of independent engines keeps falling.
4. Three engines survive in general use in 2026.
5. **Blink**, used by Chrome, Edge, Opera, Brave and Samsung Internet, forked from WebKit by Google in 2013.
6. **WebKit**, used by Safari, forked by Apple from KHTML, from the KDE project, in 2001.
7. **Gecko**, used by Firefox, descends directly from the Netscape code released in 1998. So the code Netscape published when it lost is still running today, which is a fair result for the loser of the first browser war.

■ 30.11.5 TECHNICAL – the engineer's version

1. NCSA Mosaic 0.5 alpha was announced on 23 January 1993, version 1.0 came on 21 April 1993, and NCSA ended support in January 1997.
2. Internet Explorer 1.0 shipped on 24 August 1995 in the Microsoft Plus pack for Windows 95. IE peaked above 90 percent usage share in 2001.
3. United States v. Microsoft was filed in 1998. The district court found a Sherman Act violation and in 2000 ordered a breakup; the appeals court reversed the breakup in 2001, and a settlement with conduct remedies rather than divestiture was approved in 2002.
4. Netscape announced on 22 January 1998 that its source would be free and developed openly as Mozilla. AOL announced the acquisition on 24 November 1998 at 4.2 billion dollars in stock, worth about 10 billion when it closed on 17 March 1999.
5. Firefox 1.0 was released 9 November 2004. Google Chrome was released 2 September 2008.

Engine	Origin	Used by
Gecko	Netscape code, 1998	Firefox
WebKit	KHTML fork, 2001	Safari
Blink	WebKit fork, 2013	Chrome, Edge, Opera

6. Microsoft retired its own Trident and EdgeHTML engines when Edge moved to Blink, released in January 2020. That took the world from four independent engines to three.
7. The concentration matters technically, not only commercially. When one engine has over two-thirds of usage, “it works in Chrome” starts to replace “it follows the specification” as the working definition of correct.
8. This is a live disagreement. One side argues consolidation makes the web more consistent and lets features ship faster. The other argues it hands one company effective control of a public standard. Serious people hold both positions.

■ 30.11.6 WORDS – remember these

1. Browser — the program that shows web pages — a user agent implementing HTTP, HTML, CSS and scripting.
2. Rendering engine — the part that decides what the page looks like — the layout and paint engine turning markup and styles into pixels.
3. Bundling — shipping it already installed — including a product with an operating system, the core issue in the Microsoft antitrust case.

30.12 The dot-com boom and bust

■ 30.12.1 PLAIN – in simple words

1. In 1995 three things happened at once and together they lit the fire.
2. Windows 95 put a usable computer in ordinary homes, Netscape made the web easy, and the NSFNET ban on commercial traffic was gone.
3. Suddenly anyone could sell anything to anyone, and nobody knew how big that would be.
4. Netscape’s listing on 9 August 1995 taught investors a lesson they took too far: a company with almost no profit could be worth billions.
5. Money poured in. The accepted wisdom was “get big fast”: collect users now and worry about profit later. Users were called eyeballs.
6. The NASDAQ Composite index peaked on 10 March 2000 at 5,048.62, then fell to about 1,114 on 9 October 2002, a drop of roughly 78 percent.
7. Roughly 5 trillion US dollars of market value was destroyed by the end of 2002. Some companies survived and became the largest in the world. Most did not.

■ 30.12.2 PLAIN – a picture in your head

1. Imagine a gold rush in a valley where gold has genuinely been found.
2. The gold is real. That matters. This was not a fantasy.
3. Eventually money stops arriving, and the diggers with no gold cannot pay wages and stop within weeks. The few who found real seams keep working and in twenty years own the valley.

Where this comparison breaks: gold is finite and the internet market was not. The survivors were building businesses whose costs fell as they grew, which is the opposite of mining. The bubble was in

the pricing, not in the underlying opportunity, and telling those two apart is the hardest judgement in investing.

■ 30.12.3 PLAIN – a worked example

1. Pets.com is the standard example. Founded November 1998, it ran a Super Bowl advertisement in January 2000 costing about 1.2 million dollars, went public in February 2000 at 11 dollars per share raising 82.5 million, and shut down on 9 November 2000, about nine months later.
2. The fatal detail: shipping heavy pet food cost more than the margin on it, so every sale lost money and growth made the loss bigger.
3. Webvan, founded 1996, raised 375 million dollars at its November 1999 listing and reached a valuation over 4.8 billion. In 2000 it had 178.5 million dollars of sales against 525.4 million of expenses, and filed for bankruptcy in July 2001 having lost over 800 million.
4. Boo.com, founded 1998 in London, spent 135 million dollars of venture capital in about 18 months and went into receivership on 18 May 2000.

Company	Founded	Outcome
Amazon	1994	Survived, now huge
eBay	1995	Survived, profitable early
Google	1998	Survived, dominant
Cisco	1984	Survived, lost 80 pct
Netscape	1994	Sold to AOL, 1999
Webvan	1996	Bankrupt July 2001
Boo.com	1998	Receivership May 2000
Pets.com	1998	Closed 9 Nov 2000

5. Amazon's share price fell around 90 percent from its peak and it survived anyway, largely because it had raised bond money before the market closed.
6. That is the uncomfortable truth: survival depended heavily on how much cash was in the bank when the funding stopped.

■ 30.12.4 PLAIN – what is really happening inside

1. Separate four things that get blended together in the retelling.
2. **The technology was real.** Internet use grew right through the crash and never stopped.
3. **The business models often were not.** Free delivery of heavy goods at a loss does not become profitable at scale.
4. **The valuations were unmoored.** Companies were priced on visitor counts because there were no earnings to price on.
5. **The financing was fragile.** Most needed a new funding round every year, so when funding stopped they died within months.
6. Alan Greenspan used the phrase “irrational exuberance” about asset prices in December 1996, more than three years before the peak. Being early is the same as being wrong, until it is not.
7. The honest lesson is not that the internet was overhyped. Amazon in 2026 is larger than any 1999 forecast. The forecasts were not too optimistic about the internet. They were too optimistic about who would capture the value, and how soon.

■ 30.12.5 TECHNICAL – the engineer's version

1. NASDAQ Composite closing peak: 5,048.62 on 10 March 2000. Trough: about 1,114 on 9 October 2002, roughly 78 percent below the peak, giving up all the gains of the bubble.
2. From 1995 to March 2000 the index rose about 600 percent. In 1999 alone it rose 85.6 percent, while more listed stocks fell than rose.
3. The index did not regain its March 2000 closing level until 2015.
4. The infrastructure overbuild had a long tail. Enormous amounts of optical fibre were laid and left unlit. That dark fibre was bought cheaply later and is part of why bandwidth prices fell so far in the 2000s.
5. WorldCom, a major carrier of the era, collapsed in an accounting fraud, was renamed MCI in 2003 and acquired by Verizon in 2006.
6. A useful measure for judging any such period is burn rate: cash spent per month, divided into cash on hand, gives runway in months. Most failed dot-coms had a runway under 12 months at all times.
7. The pattern generalizes. Real technology, unclear business models and stretched valuations appear in later cycles too, and reasonable people disagree in 2026 about whether current AI spending is a repeat, an overshoot, or neither. Treat that as an open argument, not a settled fact.

■ 30.12.6 WORDS – remember these

1. Eyeballs — a count of visitors used instead of profit — audience metrics used as a valuation proxy when there are no earnings.
2. Burn rate — how fast the money is spent — net cash outflow per month, which divided into reserves gives runway.
3. Dark fibre — cable laid but not switched on — installed optical fibre with no active transmission equipment attached.

30.13 Broadband and the physical internet

■ 30.13.1 PLAIN – in simple words

1. For most of the 1990s, home internet came through a modem over an ordinary phone line. A modem turns bits into sounds, because the phone network only carried sound.
2. That is why dial-up made a noise. The two modems were negotiating out loud: testing the line, agreeing a speed and measuring the echo.
3. The noise stopped once agreement was reached, because the speaker was switched off, not because the sounds stopped.
4. Then came ways to use the same copper without occupying the voice channel, which is what ADSL does. ADSL gives more download than upload speed on purpose, which is what the A means.
5. Cable television networks were adapted to carry data, and eventually fibre was run all the way into homes.
6. Underneath all of it sits the part almost nobody sees: over a million kilometres of cable on the sea floor, carrying about 99 percent of intercontinental data. Satellites carry a tiny fraction.

■ 30.13.2 PLAIN – a picture in your head

1. Think of the phone line into a house as a wide pipe that was only ever used at one end.
2. Voice needs the frequencies a human ear cares about, roughly 300 to 3,400 cycles per second. The copper can carry far more than that.

3. Dial-up squeezed data into the voice part of the pipe, so it competed with speech and was capped near 56 thousand bits per second.
4. ADSL puts data in the unused higher frequencies and leaves the voice part alone, with a small filter at each phone keeping them apart.
5. The download side gets the bigger share because that is what people mostly do, and because upstream signals are weakest exactly where interference is worst. Fibre replaces the pipe with glass, where the usable range is thousands of times wider.

Where this comparison breaks: a water pipe's capacity does not fall with length, but copper's does, badly. ADSL speed drops steadily with distance from the exchange and is usually unusable past about 4 to 5 km. Fibre does not care about a few kilometres, which is why the industry moved to it rather than squeezing copper further.

■ 30.13.3 PLAIN – a worked example

1. Follow one home connection through its generations.

Technology	Typical down	Typical up
V.34 modem, 1994	28.8 kbit/s	28.8 kbit/s
V.90 modem, 1998	56 kbit/s	33.6 kbit/s
ISDN basic rate	128 kbit/s	128 kbit/s
ADSL2+	Up to 24 Mbit/s	About 1 Mbit/s
GPON fibre	100 to 1000 Mbit/s	20 to 1000 Mbit/s

2. Take a 4-megabyte photograph, which is 32 million bits.
3. On a V.90 modem at 56 kbit/s that is about 570 seconds, so nearly ten minutes. On ADSL2+ at 20 Mbit/s it is about 1.6 seconds. On a 300 Mbit/s fibre link it is about 0.1 seconds.
4. Now the ocean. A single modern transatlantic cable such as MAREA is rated at about 224 terabits per second.
5. That one cable could move that photograph roughly 7 million times per second, which is why cables and not satellites carry the world's traffic.

■ 30.13.4 PLAIN – what is really happening inside

1. A submarine cable is thinner than you expect: about 25 mm across in deep water, weighing roughly 1.4 tonnes per kilometre.
2. Inside are a few pairs of glass fibres, a steel strength member, a copper conductor and layers of insulation and water barrier.
3. The copper is not for data. It carries direct current at several thousand volts to power repeaters on the sea floor, roughly every 60 to 100 km.
4. A cable ship lays it slowly, and in shallow water a plough buries it about a metre under the seabed to protect it from anchors and trawlers.
5. Cables break often. About 200 faults happen worldwide each year, and about two-thirds are caused by fishing gear and ships' anchors.
6. Repair means sending a ship, grappling the cable up, cutting it, splicing in a new length and lowering it back. That takes days to weeks depending on weather, permits and ship availability, which is why redundancy matters more than repair speed.
7. Satellites are different. A geostationary satellite sits about 35,786 km above the equator, so light takes about 240 milliseconds for the round trip before any processing. Low earth orbit constella-

tions fly around 550 km up, which cuts that delay enormously at the price of needing thousands of satellites.

■ 30.13.5 TECHNICAL – the engineer’s version

1. Modem standards, with years: Bell 103 at 300 bit/s in 1962; V.22bis at 2,400 in 1984; V.32 at 9,600 in 1984; V.32bis at 14,400 in 1991; V.34 at 28.8 then 33.6 kbit/s from 1994 to 1996; V.90 at 56 down and 33.6 up in 1998; V.92 in 2000, raising upload toward 48 kbit/s.
2. The dial-up handshake is a defined training sequence: answer tone, echo canceller disabling tone, probing signals to measure the channel, then rate negotiation. It is audible only because the monitor speaker is on by default, controlled by the ATM and ATL commands.
3. ADSL, ITU-T G.992.1, uses discrete multitone modulation. In Annex A, 26.075 to 137.825 kHz carries upstream and 138 to 1104 kHz downstream. ADSL2+ is G.992.5, reaching about 24 Mbit/s down. VDSL2 is G.993.2.
4. Asymmetry has two causes: crosstalk is worst at the exchange end where the upstream signal is already weakest, and demand is genuinely asymmetric. The first is engineering, the second is marketing. Both are real.
5. ISDN basic rate is 2B+D: two 64 kbit/s bearer channels plus a 16 kbit/s signalling channel, so 128 kbit/s bonded.
6. Submarine facts as of early 2026: more than 600 active and planned cables, over 1.5 million km in service, carrying about 99 percent of international data traffic. Fault rates fell from about 3.7 per 1,000 km per year between 1959 and 1979 to about 0.44 after burial became standard in 1985.

Orbit	Altitude	Typical round trip
Geostationary	35,786 km	About 240 to 600 ms
Low earth orbit	About 550 km	About 25 to 35 ms
Terrestrial fibre	Ground	About 1 ms per 100 km

7. Starlink reported approximately 10,400 satellites in orbit and more than 12 million subscribers as of June 2026, with typical latency around 25 to 35 ms.
8. India specifically. Public internet began on 15 August 1995, provided by Videsh Sanchar Nigam Limited. TRAI reported about 918 million internet subscribers as of 30 September 2023, of which about 881 million were wireless and about 37 million wired.
9. India is served by roughly 17 international submarine cables landing at about 5 main landing points, chiefly Mumbai and Chennai, including SEA-ME-WE 3, Bay of Bengal Gateway and Europe India Gateway. Reliance completed the IAX and IEX systems in 2024.
10. That concentration is the real chokepoint. A small number of landing stations in two cities serves 1.4 billion people, and cable damage in the Red Sea and near Egypt has repeatedly degraded Indian and Gulf connectivity because so many routes share one corridor.
11. The reader’s own traceroute shows this geography directly: Delhi, then Mumbai, then Pune, all inside Microsoft’s backbone, because that is where the capacity and the landing stations are.
12. Speedtest Global Index figures for India in November 2023 were about 94.62 Mbit/s mobile download and 58.62 Mbit/s fixed download. Mobile being faster than fixed is unusual worldwide and reflects how India built out.

■ 30.13.6 WORDS – remember these

1. Modem — the box that turned bits into sound — a modulator and demodulator converting digital data to and from an analogue channel.

2. ADSL — fast down, slower up, over phone copper — asymmetric digital subscriber line, ITU-T G.992 series, using discrete multitone modulation.
3. Repeater — the thing on the sea floor that keeps the light strong — an optical amplifier, usually erbium-doped, powered over the cable's copper.
4. Landing station — where a sea cable comes ashore — the terminal facility that powers the cable and connects it to terrestrial networks.
5. Chokepoint — a place where too much depends on too little — a corridor or facility whose failure degrades connectivity for a whole region.

30.14 Who actually runs the internet

■ 30.14.1 PLAIN - in simple words

1. Nobody owns the internet, and that is not a slogan. There is no company or government that can switch it off.
2. But some things must be agreed globally or nothing works.
3. Names must be unique, so two people cannot own the same domain. Addresses must be unique, so a packet has one destination. Protocols must be shared, so machines built by strangers can talk.
4. Three kinds of body handle those three jobs, and none controls the network.
5. ICANN coordinates names and, through IANA, the address and number pools.
6. The IETF writes the protocol specifications, published as RFCs. The W3C writes web standards, and five regional registries hand out address blocks in their own parts of the world.
7. Everything else is contracts between companies, and national law.

■ 30.14.2 PLAIN - a picture in your head

1. Think of the world's postal systems. No single body runs them.
2. Someone must agree country codes, or letters cannot be sorted. Someone must agree envelope sizes and address layout so machines can read them. Small, boring, essential jobs.
3. The internet has the same shape: a tiny agreed core and an enormous uncoordinated periphery.

Where this comparison breaks: postal agreements are made between governments in a treaty organization. Internet standards are made largely by engineers and companies, with governments as one voice among several. That is the multistakeholder model, and several governments have objected to it for decades, wanting state control instead. The argument is unresolved.

■ 30.14.3 PLAIN - a worked example

1. Here is how a proposal actually becomes an internet standard.
2. Someone writes an **Internet-Draft** and posts it. Anyone may do this, with no membership required.
3. A working group discusses it on a public mailing list, archived and readable by anyone. Implementations are expected, because working code carries more weight than argument.
4. When the group reaches **rough consensus**, meaning no serious unanswered objection rather than a majority vote, it goes to the steering group.
5. If approved it is published as an RFC with the status Proposed Standard. If it proves itself in deployment it can be raised to Internet Standard, and most useful protocols never bother.

```
Idea -> Internet-Draft -> working group list ->
running code -> rough consensus -> IESG review ->
RFC (Proposed Standard) -> deployment ->
Internet Standard
```

6. Meetings famously test consensus by humming rather than voting, so volume rather than head-count is judged and nobody is formally counted.

■ 30.14.4 PLAIN – what is really happening inside

1. Keep the layers of governance apart, because they are often confused.
2. **Unique identifiers.** ICANN coordinates the domain name root and the number pools, and decides which top-level domains exist.
3. **Protocol specifications.** The IETF decides how TCP, HTTP and DNS behave. It has no power over who uses them.
4. **Everything else.** Which cables exist, who peers with whom, what content is allowed and what is censored is decided by companies and states.
5. That is why “who runs the internet” has an uncomfortable answer: for the things people care about most, it is your government and your internet provider. Net neutrality is exactly the argument about that last point.

■ 30.14.5 TECHNICAL – the engineer’s version

1. ICANN was incorporated in California on 30 September 1998, taking over functions previously performed by Jon Postel at USC-ISI. It does not control content, spam, or what any network carries.
2. The IANA functions contract with the US National Telecommunications and Information Administration ended on 1 October 2016, transferring stewardship to the global multistakeholder community.
3. The IETF held its first meeting on 16 January 1986, with 21 US federal-government-funded researchers attending.
4. RFC 1 was published on 7 April 1969 by Steve Crocker of UCLA, titled “Host Software”. Jon Postel edited the series from 1969 until his death in 1998.
5. The process is defined in RFC 2026, “The Internet Standards Process, Revision 3”. RFC 6410, October 2011, cut the maturity ladder from three levels to two by retiring Draft Standard.
6. “Rough consensus and running code” comes from David Clark’s 1992 IETF presentation, where he said the group rejects kings, presidents and voting, and believes in rough consensus and running code.

Registry	Region	Since
RIPE NCC	Europe, West Asia	1992
APNIC	Asia Pacific, India	1993
ARIN	North America	1997
LACNIC	Latin America	1999
AFRINIC	Africa	2004

7. The IANA free pool of IPv4 addresses was exhausted on 3 February 2011, when the last five blocks went to the five registries. APNIC, which covers India, effectively ran out in April 2011, earliest of all.
8. The W3C was founded in October 1994 by Tim Berners-Lee at MIT. Since 2019 the HTML and DOM specifications are maintained by the WHATWG with W3C endorsement, after years of the two bodies publishing rival versions.
9. Net neutrality in the United States, in dates: the FCC voted for Title II common-carrier treatment on 26 February 2015, effective 12 June 2015; repealed it on 14 December 2017, effective 11 June 2018; voted to restore it on 25 April 2024; and the Sixth Circuit struck that down in January

2025. As of 2026 there is no federal rule, though state laws such as California's SB 822, signed 30 September 2018, remain.

10. Both sides, stated fairly. For: without a rule a provider can slow or charge for competing services, and small entrants cannot negotiate. Against: broadband needs heavy investment, traffic genuinely differs in requirements, and common-carrier rules designed for telephones discourage building. Serious economists hold both views.
11. India took a strong position. TRAI prohibited discriminatory pricing of data services in February 2016, ending zero-rating schemes, and recommended full net neutrality rules in November 2017.

■ 30.14.6 WORDS - remember these

1. ICANN — the body that coordinates names and numbers — the corporation overseeing the DNS root, gTLD policy and the IANA functions.
2. RFC — the document a standard is written in — a numbered publication of the RFC series, of which only some are standards-track.
3. Rough consensus — no serious unanswered objection — the IETF decision rule, explicitly not a majority vote.
4. Net neutrality — the rule that providers must not favour some traffic — the principle that carriers should not block, throttle or paid-prioritize.

30.15 Web 1.0 to now

■ 30.15.1 PLAIN - in simple words

1. The first web was files. Someone wrote an HTML file, put it on a server, and everyone who asked got the same file.
2. Then pages started being built when you asked for them, from a database, so a shop could show your basket rather than a fixed page.
3. From about 2004 the phrase Web 2.0 described sites where the users made the content: blogs, wikis, photos, video and comments.
4. Social networks turned that into the main use of the web, and smartphones changed it again. From around 2016 more web traffic came from phones than from computers.
5. Apps took a large share of that time back from the web, inside private platforms with their own rules.
6. Running your own server became rare, and most services now run on cloud infrastructure owned by a handful of companies.
7. So traffic and attention both concentrated, on a network originally designed to have no centre.

■ 30.15.2 PLAIN - a picture in your head

1. Imagine a town that starts as thousands of small shops, each owner-built.
2. Then a few large market halls open. They are cleaner, easier and free to enter, so almost everyone moves inside.
3. The streets still exist and the roads are still public, but nearly all the trade happens in five buildings, and each landlord sets the rules inside.
4. Nobody forced anyone. Each move was individually sensible.

Where this comparison breaks: a shop that leaves a market hall can reopen on the street. On the web, leaving usually means losing the audience, because the audience is a feature of the hall rather than of the shop. That lock-in has no clean equivalent in a market town.

■ 30.15.3 PLAIN – a worked example

1. Compare the same task across eras: publishing a photograph for friends.

Era	Method	Who controls it
1996	Edit HTML, upload	You
2003	Blog software	You, mostly
2010	Social network post	The platform
2026	App feed post	The platform

2. In 2026 anyone can do it in four seconds, and cannot control who sees it, how long it lasts, or whether it is shown at all.
3. That is the trade in one table: an enormous gain in access, and a real loss of control.

■ 30.15.4 PLAIN – what is really happening inside

1. Say plainly what was gained. Publishing became free and instant for billions of people who had no voice before. Search made the whole archive findable. Encryption became normal, so most traffic is private in transit.
2. Say plainly what was lost. Independence: most content now lives on servers the author does not control.
3. Durability: pages vanish when a platform closes, and nobody is obliged to keep them.
4. Interoperability: an early web page could be read by any client, and a modern app feed usually cannot.
5. Privacy in a different sense: the cost of free services is measurement of behaviour at a scale nobody imagined in 1993.
6. Resilience: a design meant to have no single point of failure now leans heavily on a few cloud and content-delivery providers, so single configuration errors have caused worldwide outages.
7. None of that was forced by the technology. It followed from economics: scale is cheaper, and defaults decide behaviour.

■ 30.15.5 TECHNICAL – the engineer's version

1. The phases in technical terms: static file serving; then server-side generation with CGI (RFC 3875) and later application servers; then asynchronous updates in the browser using XMLHttpRequest, which made single-page applications practical from around 2005.
2. The term Web 2.0 was popularized by the O'Reilly Media conference of that name held from 2004. It described a business and design pattern, not a protocol version. There was never a Web 1.0 specification.
3. Mobile passed desktop in worldwide web page views around late 2016 by Statcounter's measurement, and has stayed the majority since.
4. Encryption in transit became the default rather than the exception. Let's Encrypt began issuing free certificates in 2015, and the share of page loads over HTTPS in Firefox telemetry rose from under 30 percent in 2014 to above 80 percent by 2020.
5. Content delivery networks now terminate a large share of connections close to users, so your packets often never reach the origin server's country. That is exactly what the reader's trace shows, ending inside Microsoft's own network rather than at a distant data centre.
6. The honest assessment: the internet's core design remains decentralized and works. The application layer above it has centralized heavily. Those are two separate facts, often wrongly merged into one complaint.

■ 30.15.6 WORDS – remember these

1. Web 2.0 — sites built from what users post — a design and business pattern from around 2004, not a technical standard.
2. Content delivery network — copies of a site kept near you — distributed caching and connection termination close to end users.
3. Lock-in — hard to leave without losing something — switching costs created by data, audience or formats that are not portable.

30.16 A master internet timeline

1. One table, from the semaphore to today. Every date here is discussed somewhere in this chapter.

Year	Event	Significance
1792	Chappe line approved	State message network
1794	Paris to Lille opens	Minutes instead of days
1837	Cooke-Wheatstone patent	Electric telegraph starts
1844	Washington to Baltimore	Morse line proves itself
1858	First Atlantic cable	Works 3 weeks, then dies
1866	Atlantic cable works	Continents linked for good
1874	Baudot and quadruplex	Multiplexing arrives
1876	Bell patent 174,465	Voice over wire
1891	Strowger patent 447,918	Automatic switching
1892	La Porte exchange opens	Operators replaced
1960	Baran begins at RAND	Distributed network idea
1962	Licklider heads IPTO	Networked computing aim
1963	Intergalactic memo	The aim written down
1964	Baran RM-3420 series	Message blocks published
1965	Davies designs at NPL	The word packet coined
1966	Taylor starts ARPANET	Project actually funded
1969	RFC 1 published	The RFC series begins
1969	First message, 29 Oct	Link goes live, then "LO"
1970	NPL network in service	UK packet network runs
1971	Tomlinson network email	The @ sign is chosen
1972	ICCC demonstration	Doubters convinced
1973	Norway and UK links	ARPANET goes global
1974	Cerf and Kahn paper	Internetworking defined
1976	X.25 published	The carriers' rival
1978	TCP and IP split	The thin waist appears
1980	Usenet established	Discussion with no centre
1981	BITNET founded	Universities outside ARPA
1982	Minitel commercial	France online early
1983	Flag day, 1 January	ARPANET moves to TCP/IP
1983	4.2BSD ships TCP/IP	Free running code spreads
1984	DNS specified	Names scale past a file
1984	FidoNet created	Hobbyist global network
1984	ISO 7498 published	OSI reference model
1986	NSFNET at 56 kbit/s	Research backbone starts
1986	First IETF meeting	Standards body forms
1988	NSFNET T-1 backbone	1.544 Mbit/s core

Year	Event	Significance
1989	Berners-Lee proposal	Web idea written down
1990	ARPANET decommissioned	The original retires
1990	First web server runs	info.cern.ch goes live
1991	Web released publicly	Anyone can join in
1991	CIX formed	Commercial peering begins
1991	NSFNET T-3 backbone	45 Mbit/s core
1993	Mosaic released	Images inline, web grows
1993	CERN public domain	No fee, no permission
1994	Netscape founded	Web becomes an industry
1994	W3C founded	Web standards body
1995	Netscape IPO, 9 Aug	Boom psychology begins
1995	NSFNET off, 30 April	Internet goes commercial
1995	India public internet	VSNL service, 15 August
1995	IE 1.0 with Windows 95	Browser war opens
1998	ICANN incorporated	Names and numbers body
1998	Google founded	Search is the front door
1998	Mozilla source released	Gecko lineage begins
2000	NASDAQ peak, 10 March	5,048.62, then the fall
2001	IE above 90 percent	First browser war ends
2002	NASDAQ low, October	About 78 percent down
2003	NIXI registered	Indian traffic stays local
2004	Firefox 1.0	Competition returns
2008	Chrome released	Second browser war
2011	IPv4 pool exhausted	3 February, IANA dry
2012	Minitel switched off	30 June, closed model ends
2015	HTTP/2 standardized	Multiplexed web transport
2016	IANA transition	US contract ends, 1 Oct
2016	Mobile passes desktop	Phone-first web
2020	Edge moves to Blink	Three engines left
2022	HTTP/3 standardized	Web moves onto QUIC
2025	DE-CIX peak 18.73 Tbit/s	Scale of one exchange
2026	Over 600 sea cables	1.5 million km in service

30.98 Common wrong ideas

1. Wrong: the internet was built to survive a nuclear war. Right: Paul Baran's RAND work had a survivability motive, but ARPA's director stated plainly that the ARPANET was not started for that. It was built for resource sharing among scarce research computers.
2. Wrong: Al Gore claimed he invented the internet. Right: in a 1999 interview he said he took the initiative in creating the internet, meaning legislative and funding support. He sponsored the High Performance Computing Act of 1991, and Vint Cerf and Bob Kahn publicly defended his record. He never claimed to have invented it.
3. Wrong: the internet and the web are the same thing. Right: the internet is the network. The web is one application on it, created eighteen years later. Email, video calls and app traffic use the internet without the web.
4. Wrong: the internet is wireless now. Right: your last few metres may be wireless, but traffic goes onto fibre almost immediately, and about 99 percent of international traffic runs through cables on the sea floor.

5. Wrong: satellites carry most long-distance internet traffic. Right: they carry a very small fraction. Cables are far cheaper, far faster and far lower in delay.
6. Wrong: one organization runs the internet and could switch it off. Right: ICANN coordinates names and numbers, the IETF writes specifications, and nobody operates the whole network. Governments can cut off their own country, which is a different thing.
7. Wrong: Tim Berners-Lee invented hypertext. Right: hypertext ideas date to Vannevar Bush in 1945 and Ted Nelson in the 1960s. He combined URLs, HTTP and HTML into something anyone could deploy without permission.
8. Wrong: the dot-com crash proved the internet was hype. Right: usage kept growing right through the crash. The pricing and the business models were wrong, not the direction.
9. Wrong: Leonard Kleinrock invented packet switching. Right: he did important queueing analysis and later ARPANET performance work, but primary sources and historians credit Baran and Davies with the independent invention, and Kleinrock acknowledged in 2023 that his early published work was about message switching.
10. Wrong: TCP/IP won because it was technically the best design. Right: it won because it existed, worked, cost nothing, shipped in Berkeley Unix, and had a guaranteed government market while OSI was still being drafted.

30.99 Chapter summary in 20 lines

1. Long-distance messaging began by making a signal visible from the next hill, then giving it parts so it could spell anything.
2. Chappe's towers, from 1792, cut Paris to Lille from two days to about 32 minutes, and introduced encoding, relaying and a shared code book.
3. Electricity removed the need for line of sight. Cooke and Wheatstone patented in May 1837 and Morse's line ran on 24 May 1844.
4. The 1858 Atlantic cable managed about a word every ten minutes and died in weeks. The 1866 cable worked and shrank the world permanently.
5. The telephone, patented by Bell on 7 March 1876, needed exchanges, and Strowger automated them from 1891.
6. Circuit switching books a path for a whole call. That suits speech and wastes almost everything on a bursty computer conversation.
7. Packet switching splits messages into addressed pieces that travel independently and are reassembled at the far end.
8. It was invented three times: Baran at RAND from 1960 with a survivability motive, Davies at NPL in 1965 who named the packet, and Kleinrock supplying queueing analysis.
9. The ARPANET was built for resource sharing, not nuclear survival. Its first message, on 29 October 1969, was "LO" before the far end crashed.
10. The IMP was the first router, and its key idea was that hosts never had to agree with each other, only with the box beside them.
11. Email was not in the plan. Tomlinson joined two programs in 1971, chose the @ sign, and messaging became the network's main use.
12. Cerf and Kahn's May 1974 paper solved internetworking by putting a common protocol over dissimilar networks instead of making them agree.
13. TCP and IP were split in 1978, giving a thin waist: many technologies below, many applications above, one narrow protocol between.
14. On 1 January 1983 the ARPANET switched to TCP/IP in a single flag day. Nothing that size can be done that way again, which is why IPv6 has taken thirty years.

15. TCP/IP beat X.25, Minitel, private online services and OSI because it had running code, no fees, US government demand and simplicity.
16. NSFNET carried the research internet from 1986, banned commercial traffic, then lifted the ban and shut down on 30 April 1995, leaving a mesh of commercial networks and exchange points.
17. The web was proposed in March 1989, working by the end of 1990, and put into the public domain on 30 April 1993. It is an application on the internet, not the internet.
18. Mosaic in 1993 and Netscape's 9 August 1995 listing started both the browser wars and the dot-com boom. The NASDAQ peaked on 10 March 2000 at 5,048.62 and fell about 78 percent.
19. Physically the internet is fibre: over 600 submarine cables and more than 1.5 million kilometres, carrying about 99 percent of international traffic, with India served by a handful of landing points.
20. The core stayed decentralized and the applications did not, which is the central tension of the internet in 2026.

Wireless – WiFi, Bluetooth, Cellular and GPS

31.0 What this chapter gives you

1. You will be able to explain what a radio wave is and why an antenna has the length it has, and work out the wavelength of any band in your head.
2. You will be able to explain how information is put onto a wave, from simple on-off keying up to the 4096-QAM used by WiFi 7.
3. You will be able to explain OFDM properly, and say why the same idea sits under WiFi, 4G LTE, 5G and digital television.
4. You will be able to name every WiFi standard from 1997 to 2024 with its year, its band and its top speed, and say what each one added.
5. You will be able to walk through what happens on the air and in software from the moment you click a network name to the moment data flows.
6. You will be able to explain Bluetooth classic and Bluetooth Low Energy as two different protocols that share a brand name, and say why.
7. You will be able to explain how a phone attaches to a cellular network, what the SIM card actually holds, and what each generation changed.
8. You will be able to explain how GPS works, including why four satellites are needed and why the satellites need a relativity correction.

31.1 Radio from first principles

■ 31.1.1 PLAIN – in simple words

1. Push electricity back and forth along a piece of metal and it makes a wave that flies away into space. That is all radio is.
2. Electricity in a wire has two side effects. It makes a magnetic push around the wire, and an electric push along it.
3. If the electricity is steady, those pushes just sit there. If it changes, they change too.
4. A changing magnetic push creates an electric push, and a changing electric push creates a magnetic push. Each one keeps making the other.
5. So the pair can let go of the wire and keep going on their own, forever, at the speed of light.
6. That travelling pair is a **radio wave**. The metal that launches it is an **antenna**.
7. A second piece of metal far away feels the wave arrive, and the wave pushes electricity back and forth in it, very weakly. That wiggle is the signal.
8. **Frequency** is how many times per second the push reverses, measured in hertz. WiFi at 2.4 GHz reverses 2,400,000,000 times per second.

9. **Wavelength** is how far the wave travels during one full reversal. At 2.4 GHz that is about 12.3 centimetres.

■ 31.1.2 PLAIN – a picture in your head

1. Think of a long rope tied to a post at the far end of a field.
2. You hold your end and flick your wrist. A bump travels to the post. You did not send the rope. You sent a disturbance in the rope.
3. Flick faster and the bumps are closer together. The bump's travel speed does not change, only the spacing.
4. That is exactly frequency and wavelength. Speed is fixed. Faster wiggling means shorter waves.

Where this comparison breaks: a rope needs to exist. A radio wave needs nothing. It crosses empty space with no material carrying it, and the Michelson-Morley experiment of 1887 found no carrying substance at all. Also, a rope wave is one wiggle. A radio wave is two wiggles at right angles, electric and magnetic, locked together.

■ 31.1.3 PLAIN – a worked example

1. The rule is: speed equals frequency times wavelength, so $\lambda = c / f$.
2. The speed of light, written c , is 299,792,458 metres per second.
3. WiFi channel 6 sits at 2.437 GHz. $\lambda = 299,792,458 / 2,437,000,000 = 0.1230$ metres, about 12.3 cm.
4. WiFi channel 36 sits at 5.180 GHz. $\lambda = 0.0579$ metres, about 5.8 cm.

Band centre	Wavelength	Quarter wave	Used by
900 MHz	33.3 cm	8.3 cm	LoRa, old cordless
2.437 GHz	12.3 cm	3.1 cm	WiFi, Bluetooth
5.180 GHz	5.8 cm	1.4 cm	WiFi 5 GHz
6.175 GHz	4.9 cm	1.2 cm	WiFi 6E and 7
28 GHz	1.07 cm	2.7 mm	5G millimetre wave

5. A wire radiates best when it is a simple fraction of the wavelength, usually one half or one quarter.
6. At 2.4 GHz a quarter wave is 3.1 cm, which is why the metal strip inside a router aerial is about three centimetres long.

■ 31.1.4 PLAIN – what is really happening inside

1. Feed alternating current into the middle of a straight wire. Electrons pile up at one end, rush to the other, and back, at the frequency you feed in.
2. Very close to the wire the fields slosh back and forth and return their energy to the wire. This is the **near field**.
3. Further out, the field cannot get back before the wire has reversed. The loop pinches off and is stranded.
4. A stranded loop is a wave, and it must travel outwards at the speed of light. This is the **far field**.
5. The boundary is roughly one sixth of a wavelength, about 2 cm at 2.4 GHz. Inside, energy is stored. Outside, energy is radiated.
6. Length matters because the current wave bounces off each end of the wire.
7. If the wire is a half wavelength long, the reflection arrives back in step with the next push, and the wire rings like a tuning fork.

8. If the length is wrong, the reflection fights the next push, most power bounces back into the transmitter, and little is radiated.
9. That is why a broken aerial makes a radio go quiet and the transmitter run hot.

■ 31.1.5 TECHNICAL – the engineer's version

1. Radio propagation is described by Maxwell's equations, published by James Clerk Maxwell in 1865. Heinrich Hertz demonstrated generation and detection of the waves between 1886 and 1888, which is why the unit is the hertz.
2. Radio waves and visible light are the same phenomenon at different frequencies. There is no physical boundary in the spectrum, only naming.
3. Photon energy is $E = h \times f$, with $h = 6.626 \times 10^{-34}$ joule seconds. At 2.4 GHz that is 1.59×10^{-24} J, or 9.9 microelectronvolts.
4. Visible light photons carry about 2 electronvolts, roughly 200,000 times more, and breaking a chemical bond needs several electronvolts.
5. That is the physics behind calling radio **non-ionizing**: a 2.4 GHz photon cannot break a bond at any power level, only heat.

Region	Boundary	At 2.4 GHz
Reactive near field	$r < \lambda / 2\pi$	under 2.0 cm
Radiating near field	to $2 D^2 / \lambda$	depends on size
Far field	$r > 2 D^2 / \lambda$	typically over 10 cm

6. Gain is focusing, not amplification. A 3 dBi antenna sends twice the power one way by sending less elsewhere.
7. Tools: a vector network analyser measures VSWR and impedance. In software, `iw dev wlan0 link` on Linux and `wdutil info` on macOS 13 and later report the received level of a live link.

■ 31.1.6 WORDS – remember these

1. Wavelength — how long one wiggle is in space — $\lambda = c / f$ in metres.
2. Near field — the zone next to the antenna where energy sloshes — the reactive region within $\lambda / 2\pi$.
3. Far field — the zone where the wave has left and is travelling — the radiating region beyond $2 D^2 / \lambda$.
4. Non-ionizing — too weak per photon to break molecules — photon energy far below chemical bond energies, so only thermal effects.

31.2 Carrying information on a wave

■ 31.2.1 PLAIN – in simple words

1. A plain wave with nothing done to it carries no information. It is a tone.
2. To carry information you must change the wave in a way the receiver can notice and undo. Changing it is called **modulation**.
3. There are only three things about a wave you can change: its height, called **amplitude**; its wiggle rate, called **frequency**; and its timing offset, called **phase**.
4. Each jump lands the wave in one of a fixed set of states. One state is a **symbol**.
5. Four possible states means each symbol carries two bits. Sixteen states means four bits, but sixteen states are harder to tell apart.

6. That trade, more bits per symbol against needing a cleaner signal, is the single most important idea in modern wireless.

■ 31.2.2 PLAIN – a picture in your head

1. Imagine signalling to a friend across a valley with a lamp at night.
2. Brighter and dimmer is amplitude modulation. Swapping a red lamp for a green one is frequency modulation, because colour is the frequency of light.
3. Now agree four brightness levels: off, dim, medium, bright. Each flash carries two bits.

Where this comparison breaks: a lamp has one dimension, brightness. A radio symbol has two dimensions at once, because the in-phase and out-of-phase parts vary independently. That two-dimensional space is what makes QAM possible. Also, real receivers use error-correcting codes that fix a certain number of wrong guesses per block, so failure is a cliff, not a slow fade.

■ 31.2.3 PLAIN – a worked example

1. Here are the three classic modulations drawn as waves, reading the bits 1, 0, 1 from left to right.

Carrier alone (no information)

```

  /\ /\ /\ /\ /\ /\ /\ /\
 /  \ /  \ /  \ /  \ /  \ /  \

```

AM / ASK : height changes, spacing fixed

```

  /\ /\  _ _  /\ /\
 /  \ /  \ /  \ /  \
  bit 1   bit 0   bit 1

```

FM / FSK : spacing changes, height fixed

```

 /\ /\ /\ / \ / \ /\ /\ /\
 \ \ \ \ \ / \ / \ \ \ \ \
  bit 1   bit 0   bit 1

```

PSK : height and spacing fixed, wave jumps in time

```

 /\ /\ | \ / \ | \ /\
 /  \ |  \ / \ | \ \
      ^ flip   ^ flip

```

2. **ASK**, amplitude shift keying: two brightness levels. Morse code by radio is ASK.
3. **FSK**, frequency shift keying: two tones. Old dial-up modems and the Bluetooth radio use it.
4. **PSK**, phase shift keying: in step or half a cycle out of step. Two states, one bit per symbol. That case is **BPSK**.
5. **QPSK** uses four phase positions a quarter turn apart, two bits per symbol.
6. **QAM** varies phase and amplitude together, giving a grid of states.

Scheme	States	Bits per symbol	First WiFi use
BPSK	2	1	802.11a, 1999
QPSK	4	2	802.11a, 1999
16-QAM	16	4	802.11a, 1999
64-QAM	64	6	802.11a, 1999
256-QAM	256	8	802.11ac, 2013
1024-QAM	1024	10	802.11ax, 2019
4096-QAM	4096	12	802.11be, 2024

7. A **constellation diagram** pictures those states. The sideways axis is the in-phase part, the up axis is the out-of-phase part, and each allowed state is a dot.
8. BPSK is two dots, left and right. QPSK is four, one per corner. 16-QAM is a four-by-four grid, 256-QAM is sixteen by sixteen, 4096-QAM is sixty-four by sixty-four.
9. With noise, each received dot lands in a fuzzy cloud instead of exactly on its spot.
10. On a four-by-four grid the clouds have room. On a sixty-four by sixty-four grid they overlap and the receiver picks the wrong dot.

■ 31.2.4 PLAIN – what is really happening inside

1. **Symbol rate** is how many states per second you send, limited by how much spectrum you may occupy.
2. **Bit rate** is symbol rate times bits per symbol, reduced by the error-correcting overhead.
3. So there are only two ways to go faster: more symbols per second, needing more spectrum, or more bits per symbol, needing a cleaner signal. There is no third way.
4. Spread spectrum looks like the opposite of going faster.
5. In **frequency hopping spread spectrum**, FHSS, the transmitter jumps between small channels many times a second in a pattern the receiver knows. Jamming one channel costs only the hops that landed there.
6. In **direct sequence spread spectrum**, DSSS, each bit is replaced by a fixed longer pattern of faster pulses called chips.
7. Sending eleven chips per bit spreads the signal over eleven times the bandwidth, so it looks weaker and wider.
8. The receiver multiplies by the same known pattern. The wanted signal adds up; noise, being unrelated, cancels. That gain is why GPS can be received from orbit despite arriving weaker than the background noise.
9. Now **OFDM**, which solves echo. Indoors a signal arrives several times: direct, off a wall, off the floor, late by tens of nanoseconds.
10. OFDM makes each symbol much longer, four microseconds instead of fifty nanoseconds, so an echo overlaps only a tiny fraction of it.
11. Long symbols mean a slow symbol rate, so OFDM sends many symbols at once, side by side, on many narrow subcarriers.
12. The subcarriers are spaced so that at the exact centre of each, every other one passes through zero. That is what **orthogonal** means here, and it lets them overlap in frequency without interfering.
13. Each symbol is sent with a copy of its own tail glued to the front, the guard interval or cyclic prefix. Echoes land in the copy, which is thrown away.

■ 31.2.5 TECHNICAL – the engineer’s version

1. Frequency hopping was patented by Hedy Lamarr, filed under her legal name Hedy Kiesler Markey, with the composer George Antheil. US Patent 2,292,387, “Secret Communication System”, was granted on 11 August 1942.
2. The mechanism used a punched roll like a player piano to step a torpedo guidance link through frequencies so it could not be jammed.
3. The US Navy rejected it as too bulky, and the patent expired before similar systems entered service around 1962.
4. It is honest to say Lamarr and Antheil did not invent WiFi. They patented one idea, decades before the electronics existed to do it cheaply.
5. Shannon’s capacity theorem, published by Claude Shannon in 1948, sets the ceiling: $C = B \times \log_2(1 + S/N)$.

6. Worked example: a 20 MHz channel at 25 dB SNR. 25 dB is a ratio of 316. $C = 20,000,000 \times \log_2(316) = 166 \text{ Mbit/s}$.
7. That is why one 20 MHz stream cannot exceed roughly 170 Mbit/s; real 802.11ax tops out at 143.4 Mbit/s.
8. 802.11b DSSS: 1 and 2 Mbit/s use an 11-chip Barker sequence at 11 Mchip/s. 5.5 and 11 Mbit/s use Complementary Code Keying with 8-chip codewords.

Parameter	802.11a/g value
FFT size	64 points
Subcarrier spacing	312.5 kHz
Data subcarriers	48
Pilot subcarriers	4
Symbol time	3.2 us plus 0.8 us guard

9. Check the top rate: 48 subcarriers $\times$ 6 bits (64-QAM) $\times$ 3/4 coding = 216 bits per symbol, divided by 4.0 microseconds = 54 Mbit/s exactly.
10. 802.11ax quadrupled the symbol: 78.125 kHz spacing, 256-point FFT for 20 MHz, 234 data subcarriers, 12.8 us symbol plus a 0.8, 1.6 or 3.2 us guard.
11. The longer symbol tolerates longer echoes, which is why 802.11ax works better outdoors and in large halls.

Modulation and coding	Rate, 1 stream	Needed SNR
BPSK 1/2 (MCS0)	8.6 Mbit/s	about 2 dB
16-QAM 1/2 (MCS3)	34.4 Mbit/s	about 11 dB
64-QAM 3/4 (MCS6)	68.8 Mbit/s	about 22 dB
1024-QAM 5/6 (MCS11)	143.4 Mbit/s	about 35 dB

12. Those SNR values are typical receiver sensitivity figures published by chip vendors, not values written into the standard.

■ 31.2.6 WORDS – remember these

1. Modulation — changing a wave so it carries information — varying amplitude, frequency or phase of a carrier according to a message.
2. QAM — packing bits by changing both strength and timing — quadrature amplitude modulation with independent I and Q levels.
3. Spread spectrum — deliberately using more bandwidth than needed — trading bandwidth for processing gain and jam resistance.
4. DSSS — every bit sent as a longer known pattern — direct sequence spreading by a chip code such as the 11-chip Barker sequence.
5. FHSS — jumping between channels on a shared schedule — frequency hopping, 1600 hops per second in Bluetooth.
6. OFDM — many slow narrow signals instead of one fast wide one — orthogonal frequency division multiplexing with IFFT/FFT and a cyclic prefix.

31.3 The realities of radio

■ 31.3.1 PLAIN – in simple words

1. A radio link is a shouting match across a noisy room. Three things decide whether it works: how loud you shout, how much the journey swallows, and how noisy the room already is.
2. Radio power is measured in **dBm**. 0 dBm means one milliwatt. Every 10 dBm added multiplies power by ten; every 10 dBm removed divides by ten.
3. A router transmits at about +20 dBm, which is 100 milliwatts. Your laptop hears it at about -50 dBm, one hundred billionth of a milliwatt.
4. The room is never silent. There is always a hiss from heat in the electronics and from the universe. That hiss is the **noise floor**.
5. What matters is not how loud your signal is, but how far above the hiss it is. That gap is the **signal-to-noise ratio**.
6. Signal also arrives more than once, bouncing off walls. The copies can add up or cancel out. That is **multipath**.
7. Lower frequencies bend around obstacles and pass through walls better, while higher ones carry more data but are stopped more easily.
8. That is the whole 2.4 versus 5 GHz story: 2.4 goes further, 5 goes faster.

■ 31.3.2 PLAIN – a picture in your head

1. Imagine a party. Your voice loudness is the transmit power. The distance and the bodies between you are the path loss. Everyone else talking is the noise floor.
2. Add echo. Your voice arrives directly and again off the wall behind them. A short echo is unnoticed; a long one smears words together.
3. Now imagine you and a friend speaking at once from different places, each saying half a sentence. The listener has two ears and can separate you by direction. That is MIMO.

Where this comparison breaks: at a party, two people talking at once is always worse. In radio, two transmitters and two receivers with enough echo in the room can genuinely double the data, because the echoes make the paths mathematically separable. Multipath, the enemy in older systems, became the resource MIMO exploits.

■ 31.3.3 PLAIN – a worked example

1. A real link budget for a laptop five metres from a router.
2. Router transmit power +20 dBm, plus 3 dBi antenna gain, gives +23 dBm radiated.
3. Free space loss at 5 metres and 2.4 GHz is 54 dB. One internal wall costs 6 dB. Laptop antenna gain 0 dBi.
4. Received signal: $23 - 54 - 6 = -37$ dBm. Very strong.
5. The same at 5 GHz, which loses about 6.4 dB more and where walls cost roughly double: $23 - 72.4 - 24 = -73.4$ dBm. Marginal.

Signal (dBm)	Quality	What works
-30 to -50	Excellent	Everything, top rates
-50 to -60	Very good	Video calls, 4K streaming
-60 to -67	Good	Reliable, voice calls fine
-67 to -70	Fair	Streaming may stutter
-70 to -80	Weak	Browsing only, drops likely
below -80	Unusable	Association fails or flaps

6. See it yourself. On macOS: `wdutil info`. On Linux: `iw dev wlan0 link`. On Windows: `netsh wlan show interfaces`.

■ 31.3.4 PLAIN – what is really happening inside

1. Power spreads over the surface of an expanding sphere. Double the distance and the same power covers four times the area, so a fixed antenna catches a quarter as much. That is exactly 6 dB per doubling.
2. Higher frequency loses more for a subtle reason. The spreading is identical, but an antenna sized for a shorter wavelength has a smaller catching area.
3. Multipath means copies arrive at different times. Where two copies arrive half a wavelength apart, they cancel.
4. Half a wavelength at 2.4 GHz is 6 centimetres, which is why moving a laptop by a hand's width can change signal by 10 dB.
5. That cancellation is **fading**, and it is frequency-dependent: one subcarrier may sit in a null while its neighbour is fine. OFDM survives by spreading data across many subcarriers plus error correction.
6. MIMO uses several antennas at each end. Each pair sees a slightly different mix of echoes.
7. With enough difference, the receiver solves a small set of simultaneous equations and separates two or more independent streams sent at the same time on the same channel. Each is a **spatial stream**.
8. **Beamforming** is different. It sends the same data from several antennas with small timing offsets so the copies add at the receiver and cancel elsewhere. It improves signal, not stream count.

■ 31.3.5 TECHNICAL – the engineer's version

1. dBm is decibels relative to one milliwatt: $P(\text{dBm}) = 10 \times \log_{10}(P \text{ in mW})$.
2. Free space path loss in decibel form: $\text{FSPL}(\text{dB}) = 20 \log_{10}(d_{\text{km}}) + 20 \log_{10}(f_{\text{MHz}}) + 32.44$.
3. Worked: $d = 0.005 \text{ km}$, $f = 2437 \text{ MHz}$. $20 \log_{10}(0.005) = -46.0$. $20 \log_{10}(2437) = 67.7$. Total = 54.1 dB.
4. Same at 5180 MHz: $20 \log_{10}(5180) = 74.3$, giving 60.7 dB. The 6.6 dB difference is the 2.4 versus 5 GHz penalty.
5. Thermal noise is $N = k \times T \times B$ with $k = 1.38 \times 10^{-23} \text{ J/K}$. At 290 K the noise density is -174 dBm/Hz.
6. For 20 MHz: $-174 + 10 \log_{10}(20,000,000) = -101 \text{ dBm}$. Add a typical 5 dB receiver noise figure and the practical floor is about -96 dBm.

Material	Loss at 2.4 GHz	Loss at 5 GHz
Plasterboard wall	3 to 5 dB	4 to 7 dB
Brick wall	6 to 12 dB	10 to 20 dB
Concrete floor	12 to 20 dB	20 to 30 dB
Clear glass	2 to 4 dB	3 to 6 dB
Human body	3 to 5 dB	5 to 8 dB

7. The honest version of RSSI: in the standard it is a vendor-defined unitless integer with no required scale. Drivers convert it to dBm using their own table, so two adapters in the same spot can legitimately disagree.
8. Tools: `iperf3` for throughput, `iw dev wlan0 station dump` for per-station rate and signal, and Wireshark with a monitor-mode adapter for the radiotap header.

■ 31.3.6 WORDS – remember these

1. Noise floor — the background hiss you must shout over — kTB thermal noise plus receiver noise figure, about -96 dBm in a 20 MHz channel.
2. SNR — the gap between your signal and the hiss — signal-to-noise ratio in decibels, the direct determinant of usable modulation order.
3. Multipath — the same signal arriving by several routes — delayed replicas causing frequency-selective fading.
4. MIMO — several antennas sending different data at once — multiple input multiple output spatial multiplexing.
5. Beamforming — aiming the signal by timing several antennas — applying a steering matrix so transmissions add constructively at the receiver.

31.4 Spectrum and regulation

■ 31.4.1 PLAIN – in simple words

1. Radio is shared. If everyone transmits on the same frequency, nobody gets through, so governments divide the spectrum into bands and say who may use each one, for what, and at what power.
2. Most bands are **licensed**. One company pays for exclusive use. Mobile phone bands work this way.
3. A few bands are **unlicensed**. Anyone may transmit if they obey power limits and interference rules, and may not complain about interference.
4. The main ones are the **ISM bands**, short for industrial, scientific and medical. They were not created for communication. They were set aside for machines that leak radio while doing something else.
5. A microwave oven is the classic example. It cooks with radio at 2.45 GHz. Nobody wanted that band because ovens were already polluting it.
6. The rules differ by country. The rule set in force where you are is called a **regulatory domain**, and your router turns off channels not allowed there.
7. Part of the 5 GHz band is shared with weather and military radar. There, a WiFi device must listen first and leave immediately if it hears a radar. That rule is called DFS.

■ 31.4.2 PLAIN – a picture in your head

1. Think of the spectrum as a very long street of shops. Most of it is rented, one tenant per shop with a locked door. That is licensed spectrum.
2. A few stretches are public squares. Anyone may set up a stall, under rules about how loud you may shout and how big your stall may be.
3. One square sits next to a factory that hums loudly. Rents there are zero. That square is 2.4 GHz.

Where this comparison breaks: a market stall takes a fixed amount of floor. Radio devices share time and frequency together, and a well-behaved device can share a channel with dozens of others by taking short turns. Also the square is public per country, not globally: the same square is open in India and closed in Japan, on the same street.

■ 31.4.3 PLAIN – a worked example

1. The 2.4 GHz band runs from 2400 to 2483.5 MHz in most countries.
2. WiFi defines 14 channels in it, spaced 5 MHz apart, but each channel is about 20 MHz wide, so neighbours overlap heavily.

- Only channels 1, 6 and 11 are far enough apart not to overlap. That is the origin of the famous advice to use 1, 6 or 11.

Channel	Centre	Overlaps with
1	2412 MHz	2, 3, 4, 5
6	2437 MHz	2 to 5 and 7 to 10
11	2462 MHz	7 to 10, 12, 13
14	2484 MHz	Japan only, 11b only

- Country differences: the United States allows channels 1 to 11, most of Europe and India allow 1 to 13, and Japan allows 1 to 14 with channel 14 restricted to the old 802.11b modulation.

Sub-band	Range (MHz)	DFS needed
U-NII-1	5150 to 5250	No
U-NII-2A	5250 to 5350	Yes
U-NII-2C	5470 to 5725	Yes
U-NII-3	5725 to 5850	No

- India is a useful case because it moved recently.
- The Department of Telecommunications published draft rules in May 2025 and notified the final delicensing on 22 January 2026, opening 5925 to 6425 MHz for indoor low-power and very-low-power wireless access.
- That is half the band the United States opened in April 2020. India's rules also forbid use on oil platforms, in cars, trains, boats and drones, and on aircraft below 10,000 feet.

■ 31.4.4 PLAIN – what is really happening inside

- Access points advertise their country in a **Country information element** inside every beacon frame, and a client supporting 802.11d adopts it.
- DFS works in three steps. Before using a radar channel, the access point listens silently for 60 seconds. That is the **channel availability check**.
- While using it, the access point keeps listening. On detecting a radar it must stop transmitting within 200 milliseconds and vacate within 10 seconds.
- It must then avoid that channel for 30 minutes, the **non-occupancy period**.
- This is why a 5 GHz network sometimes vanishes for a minute and returns on a different channel. It is not a fault. It is the law working.

■ 31.4.5 TECHNICAL – the engineer's version

- The ISM bands are defined in the ITU Radio Regulations, article 5.
- The key ones are 902 to 928 MHz (Region 2 only), 2400 to 2500 MHz (global), 5725 to 5875 MHz (global) and 24 to 24.25 GHz.
- Domestic microwave ovens operate at 2450 MHz plus or minus 50 MHz, which is exactly why that band was designated ISM and why WiFi shares it.
- The founding regulatory act for WiFi was the United States FCC Report and Order in Docket 81-413, adopted in May 1985, permitting unlicensed spread spectrum in the ISM bands up to 1 watt.
- Michael Marcus, then an FCC engineer, drove that rulemaking.

Band and region	EIRP limit	Note
2.4 GHz, Europe	20 dBm (100 mW)	Whole band
2.4 GHz, USA	36 dBm point-to-point	30 dBm typical
5150–5250, Europe	23 dBm	Indoor only
6 GHz LPI, India	30 dBm total	Indoor, no outdoor antenna

- India's regulator for spectrum is the Wireless Planning and Coordination Wing of the Ministry of Communications, written WPC. Equipment approval runs through Equipment Type Approval, ETA.
- Inspect the applied domain on Linux with `iw reg get`:


```
country IN: DFS-UNSET
(2402 - 2482 @ 40), (N/A, 20), (N/A)
(5170 - 5330 @ 160), (N/A, 23), (N/A)
(5735 - 5835 @ 80), (N/A, 30), (N/A)
```
- The honest version of “1, 6 and 11 do not overlap”: they do not overlap at the -30 dB level used in the spectral mask, but real transmitters leak outside the mask and cheap receivers reject adjacent channels imperfectly.
- It is a standard that 2.4 GHz channels are 5 MHz apart. It is a convention, not a standard, that everyone uses 1, 6 and 11.

■ 31.4.6 WORDS – remember these

- ISM band — bands set aside for machines that leak radio — allocations under ITU Radio Regulations article 5.
- Regulatory domain — the country rule set your radio obeys — a country code mapping to permitted channels, power and DFS requirements.
- DFS — get off the channel if radar is heard — dynamic frequency selection per ETSI EN 301 893 or FCC Part 15.407.
- WPC — India's spectrum authority — Wireless Planning and Coordination Wing, Ministry of Communications.

31.5 The WiFi hardware

■ 31.5.1 PLAIN – in simple words

- A WiFi radio is four things in a row: a digital part, a converter, an analogue radio part, and an antenna.
- The digital part works out what wave shape should be sent. That is the **baseband** processor.
- It also runs the rules of taking turns on the air. That part is the **MAC**, short for medium access control.
- The analogue radio part shifts that wiggle up to 2.4 or 5 GHz, amplifies it, and pushes it to the antenna. Coming in, its most important job is to amplify an extremely weak signal without adding noise.
- In a modern device the first three are one silicon chip. Around it sit a power amplifier, a low-noise amplifier, switches and filters, together called the **RF front end**.
- The chip runs its own small program, stored as **firmware**. Your operating system does not do the split-second radio timing. The chip does.

■ 31.5.2 PLAIN – a picture in your head

1. Think of a small radio station with four rooms.
2. A producer writes the script and decides when the station goes on air. That is the MAC and the baseband.
3. A machine turns the script into sound, and a transmitter lifts that sound onto the broadcast frequency where an amplifier makes it loud. That is the converter, the mixer and the power amplifier.
4. Incoming signals run the same rooms backwards, and the most delicate part is the first receive stage, because whatever noise it adds is amplified along with the tiny signal forever after. That stage is the low-noise amplifier.

Where this comparison breaks: a radio station transmits or receives. A WiFi chip must do both, alternating thousands of times per second, sharing one antenna through a switch that must settle in under a few microseconds.

■ 31.5.3 PLAIN – a worked example

1. Open a laptop from around 2020 and you find a small card in an M.2 2230 slot, about 22 mm by 30 mm.
2. Two thin coaxial wires run from that card up through the hinge into the lid, ending at flat metal shapes printed just above the screen.
3. Those are the antennas, placed high and away from the metal chassis and your hands.

Part	Job	Typical figure
MAC and baseband	Frames, timing, OFDM	2x2 spatial streams
RF transceiver	Up and down conversion	2.4, 5 and 6 GHz
Power amplifier	Boost transmit signal	+18 to +20 dBm
Low-noise amplifier	Boost received signal	2 to 3 dB noise figure
Antenna switch	Share antenna Tx/Rx	Under 1 us switching

4. In a phone there is no card. WiFi, Bluetooth and often GPS sit in one **combo** chip soldered to the main board.
5. Phone antennas are usually part of the metal frame, split into segments by the thin plastic gaps you can see on the outside edge.
6. Real examples: Apple used Broadcom BCM4387-family parts in iPhone 13 and 14 era devices, and Qualcomm's FastConnect 7800, announced in 2022, was the first widely shipped WiFi 7 combo.

■ 31.5.4 PLAIN – what is really happening inside

1. Follow one packet out of the chip. The MAC has a frame of bytes and permission to transmit.
2. The baseband adds error correction, maps groups of bits onto constellation points, and lays those points across the OFDM subcarriers.
3. An inverse fast Fourier transform turns that set of subcarriers into a stream of numbers describing the wave shape over time.
4. Converters turn those numbers into two voltages, I and Q, and a mixer multiplies them by a pure tone from a **local oscillator** to shift everything up to 2.4 or 5 GHz.
5. The power amplifier raises it from a few milliwatts to 60 or 100 milliwatts, a switch connects the antenna, and the wave leaves.
6. Coming in, the low-noise amplifier boosts the tiny signal first, before anything can add noise.

7. A mixer shifts it down, converts digitize it, an FFT splits it into subcarriers, an equalizer undoes the channel distortion using known training symbols at the start of the frame, and a decoder recovers bits.
8. The firmware handles anything faster than the operating system can be trusted to react: acknowledgements within 16 microseconds, backoff countdown, rate selection and power save timing.

■ 31.5.5 TECHNICAL – the engineer’s version

Interface	Typical use	Note
PCIe	Laptop M.2 cards	Highest throughput
SDIO	Embedded, older phones	Limited to ~200 Mbit/s
USB	Dongles	Latency and jitter issues
CNVi	Intel laptops since 2018	MAC moved into the chipset

1. CNVi is an instructive implementation detail: Intel moved the MAC and baseband into the platform controller hub, leaving only RF on the M.2 card. This is why Intel CNVi cards do not work in AMD laptops.
2. Firmware is loaded at driver initialization from files on the host. On Linux they live under `/lib/firmware`:

```

/lib/firmware/iwlwifi-ty-a0-gf-a0-89.ucode      Intel AX210
/lib/firmware/brcm/brcmfmac43455-sdio.bin      Broadcom in Pi 4
/lib/firmware/ath11k/QCA6390/hw2.0/amss.bin    Qualcomm
/lib/firmware/mediatek/WIFI_RAM_CODE_MT7922_1.bin

```

3. Check what is loaded with `dmesg | grep -i firmware` or `ethtool -i wlan0`, which prints driver, firmware version and bus address.
4. Nearly all WiFi firmware is a closed binary. That is a real limitation for auditability, and it has hosted serious vulnerabilities, notably Google Project Zero’s 2017 findings against Broadcom WiFi firmware.
5. Two design styles exist. **FullMAC** puts scanning, association and much of the MAC in firmware. **SoftMAC** puts them in the host driver.
6. Most Broadcom and Qualcomm mobile parts are FullMAC; most Atheros and Intel parts on Linux are SoftMAC, which is why they support monitor mode better.

■ 31.5.6 WORDS – remember these

1. Baseband — the part that shapes the wave from bits — PHY layer coding, mapping, IFFT and FFT.
2. MAC — the part that decides when to talk — medium access control, the 802.11 state machine and timing.
3. Low-noise amplifier — the delicate first receive stage — LNA whose noise figure sets the receiver’s sensitivity.
4. Firmware — the program running on the radio chip itself — closed binary loaded from the host, handling microsecond timing.
5. SoftMAC — the computer handles connecting — MAC state machine in the host driver, as with Linux mac80211.

31.6 WiFi history and standards

■ 31.6.1 PLAIN – in simple words

1. WiFi did not start as a product idea. It started as a regulator giving away spectrum nobody wanted.
2. In 1985 the United States regulator allowed unlicensed devices to use the ISM bands if they used spread spectrum. That made it legal to build a wireless network without buying spectrum.
3. The way devices take turns was borrowed from ALOHAnet, built at the University of Hawaii and operating from June 1971.
4. ALOHAnet connected islands by radio at 9600 bits per second. Its rule was simple: transmit when you have something to send, and if no acknowledgement comes, wait a random time and try again.
5. That idea, random access with random backoff, became Ethernet and then WiFi.
6. The first WiFi standard, IEEE 802.11, arrived in 1997 at 1 or 2 megabits per second. It was too slow to matter.
7. In 1999 two versions appeared: 802.11b at 11 megabits in 2.4 GHz, and 802.11a at 54 megabits in 5 GHz. 802.11b won, because 2.4 GHz radios were cheaper and the range was better.
8. The name WiFi was invented in 1999 by a branding company. It does not stand for anything.

■ 31.6.2 PLAIN – a picture in your head

1. Think of WiFi generations as widening a road and adding lanes.
2. 802.11b was one narrow lane. 802.11a and g repaved it with a better surface, OFDM instead of DSSS, for five times the traffic.
3. 802.11n added lanes, with several antennas sending different data at once, and 802.11ac widened each lane from 20 MHz to 80 and then 160.
4. 802.11ax neither widened nor added lanes. It fixed the junction, letting several small vehicles cross together instead of one at a time.
5. 802.11be opened a second road, the 6 GHz band, and lets a device use two roads at once.

Where this comparison breaks: roads do not interfere with each other. Radio channels do, and adding width in a crowded area can make things worse. A 160 MHz channel in a block of flats often performs worse than an 80 MHz one.

■ 31.6.3 PLAIN – a worked example

1. Here is the complete family. Read the rate column as a theoretical ceiling no real device reaches.

Standard	Name and year	Band	Max PHY rate
802.11-1997	none, 1997	2.4 GHz	2 Mbit/s
802.11b	WiFi 1, 1999	2.4 GHz	11 Mbit/s
802.11a	none, 1999	5 GHz	54 Mbit/s
802.11g	WiFi 3, 2003	2.4 GHz	54 Mbit/s
802.11n	WiFi 4, 2009	2.4 and 5	600 Mbit/s
802.11ac	WiFi 5, 2013	5 GHz	6.93 Gbit/s
802.11ax	WiFi 6, 2019/21	2.4/5/6	9.6 Gbit/s
802.11be	WiFi 7, 2024	2.4/5/6	about 46 Gbit/s

Standard	Key feature added
802.11b	DSSS and CCK, first mass adoption
802.11a	OFDM, 5 GHz band
802.11g	OFDM at 2.4 GHz, b compatible

Standard	Key feature added
802.11n	MIMO, 40 MHz, frame aggregation
802.11ac	80/160 MHz, 256-QAM, MU-MIMO down
802.11ax	OFDMA, 1024-QAM, TWT, BSS colour
802.11be	320 MHz, 4096-QAM, multi-link

- Note the numbering gap. The Wi-Fi Alliance introduced generation numbers in October 2018 and applied them backwards only to n, ac and ax. “WiFi 1” through “WiFi 3” for b, a and g are retrospective informal labels.
- WiFi 6E is not a new standard. It is 802.11ax operating in the 6 GHz band, branded separately from 2021.

■ 31.6.4 PLAIN – what is really happening inside

- Now the Australian claim, told honestly, because it is often told badly.
- In the early 1990s a team at CSIRO, Australia’s national science agency, was working on a radio astronomy problem: detecting a weak signal smeared by echoes.
- The team was led by John O’Sullivan, with Graham Daniels, Terence Percival, Diethelm Ostry and John Deane.
- Their solution combined forward error correction, frequency-domain interleaving and multi-carrier modulation, done with fast Fourier transforms in a single chip.
- That combination was granted as US Patent 5,487,069 on 23 January 1996.
- CSIRO settled with Hewlett-Packard in April 2009, then Microsoft, Intel, Dell, Toshiba, Asus, Nintendo and others in 2009 and 2010, and with AT&T, Verizon and T-Mobile in 2012 for about 229 million dollars.
- Total income from the patent was roughly 430 million dollars, and the inventors received the European Inventor Award in 2012.
- What is not true is that CSIRO invented WiFi.
- OFDM itself was described by Chang at Bell Labs in 1966, and the 802.11 committee was chaired by Vic Hayes of NCR in the Netherlands, whose team built the first products.
- The honest version: WiFi has no single inventor.
- The FCC opened the band, NCR and Lucent built the first radios, the IEEE committee wrote the rules, CSIRO contributed a key patented signal processing idea, and Apple shipping AirPort in the 1999 iBook made it mainstream.

■ 31.6.5 TECHNICAL – the engineer’s version

- IEEE 802.11 formed as a working group in 1990. Vic Hayes of NCR chaired it for ten years and is commonly called the father of WiFi.
- The 1997 base standard defined three physical layers: FHSS at 2.4 GHz, DSSS at 2.4 GHz, and diffuse infrared at 1 and 2 Mbit/s. The infrared PHY was never commercially implemented.
- 802.11b, ratified September 1999, added CCK for 5.5 and 11 Mbit/s. 802.11a, ratified the same month, used OFDM at 5 GHz with rates 6 to 54 Mbit/s.
- The Wireless Ethernet Compatibility Alliance was founded in 1999 and renamed the Wi-Fi Alliance in 2002. Interbrand coined the name WiFi in 1999 from a list of ten candidates.
- The “Wireless Fidelity” expansion is a myth. Phil Belanger, a founding member, has stated the name has no expanded meaning; early Alliance marketing briefly used the phrase and created the confusion.

6. 802.11ax was published as IEEE 802.11ax-2021, with Wi-Fi Alliance certification from September 2019. It introduced OFDMA, uplink MU-MIMO, 1024-QAM, target wake time and BSS colouring.
7. 802.11be certification as Wi-Fi 7 began on 8 January 2024, with the IEEE standard published 22 July 2025. Key additions: 320 MHz channels, 4096-QAM, multi-link operation, multiple resource units and preamble puncturing.
8. Rate arithmetic for 802.11be: 8 spatial streams, 320 MHz, 4096-QAM, 5/6 coding and a 0.8 us guard gives about 23 Gbit/s. The headline 46 Gbit/s assumes the specification maximum of 16 spatial streams, which no product implements.
9. Wi-Fi 8 is IEEE 802.11bn, under the banner Ultra High Reliability, projected to be finalized around May 2028.
10. Its stated goal is not higher peak rate but better worst-case behaviour: lower latency variance, seamless roaming and coordinated access points. Treat published figures as active work, not established fact.

■ 31.6.6 WORDS – remember these

1. IEEE 802.11 — the committee and document that defines WiFi — the working group and its standard, amended by letter suffixes.
2. Wi-Fi Alliance — the industry group that tests and brands — the certifier owning the WiFi trademark and generation numbers.
3. 802.11ax — the standard behind WiFi 6 and 6E — high efficiency WLAN with OFDMA and 1024-QAM, published 2021.
4. 802.11be — the standard behind WiFi 7 — extremely high throughput with 320 MHz, 4096-QAM and multi-link operation, published July 2025.
5. CSIRO patent — the Australian signal processing patent WiFi uses — US 5,487,069 of January 1996, covering FEC with interleaved multicarrier modulation.

31.7 How WiFi actually works on the air

■ 31.7.1 PLAIN – in simple words

1. An access point announces itself about ten times a second with a short message called a **beacon**, carrying the network name, supported speeds, security type and a timestamp.
2. Your device can sit quietly and listen for beacons, called **passive scanning**, or shout a question and wait for answers, called **active scanning**. The shout is a probe request.
3. The network name you see is the **SSID**. The unique hardware address of the access point radio is the **BSSID**.
4. To join, your device sends an authentication frame, then an association request. The access point replies and gives it a small ID number. Then the security handshake runs, and only then can real data flow.
5. The rule is: listen first. If the channel is busy, wait. If it is free, wait a random extra amount, then transmit.
6. Every unicast frame must be acknowledged. No acknowledgement means it was lost and is sent again.
7. All that listening, waiting and acknowledging costs time. Roughly half the advertised speed disappears into it.

■ 31.7.2 PLAIN – a picture in your head

1. Think of a polite conference call with no chairperson.
2. Nobody can hear themselves while speaking, because their own voice drowns everything out, so nobody can tell if they talked over someone.
3. That is exactly the radio situation: a transmitter cannot listen on its own channel while transmitting, because its own signal is a billion times stronger than anything arriving.
4. The etiquette is: wait for silence, count silently for a random number, and if it is still silent when you reach zero, start talking.
5. Now a problem: two people at opposite ends of a long table can both hear the chairperson but not each other, and will talk over each other forever.
6. The fix is to ask the chairperson “may I speak”, and have the chairperson say out loud “yes, everyone else be quiet for two minutes”.

Where this comparison breaks: humans hear a clash afterwards and apologize. Radios never learn about the clash directly, only from the missing acknowledgement. Also the “random count” is not seconds, it is a count of 9 microsecond slots, and the range doubles after every failure.

■ 31.7.3 PLAIN – a worked example

1. Here is a full join sequence, in order.

Client	Access point
<----- Beacon (every 102.4 ms) ---	
----- Probe Request (broadcast) ----->	
<----- Probe Response -----	
----- Authentication (open) ----->	
<----- Authentication (success) -----	
----- Association Request ----->	
<----- Association Response (AID 3) ---	
<----- EAPOL 4-way handshake msg 1 ----	
----- EAPOL msg 2 ----->	
<----- EAPOL msg 3 -----	
----- EAPOL msg 4 ----->	
(keys installed, port open)	
----- DHCP Discover ----->	
<----- DHCP Offer / Ack -----	
data flows	

2. Typical times: probe response within 10 milliseconds, authentication and association a few milliseconds, four-way handshake 20 to 60 milliseconds, DHCP 10 to 100 milliseconds. Total join time is usually 100 to 300 milliseconds.
3. Now the taking-turns arithmetic for one 1500 byte packet on a 5 GHz 802.11ac link advertising 866 Mbit/s.

Step	Time
DIFS wait	34 us
Average random backoff	67 us
Preamble and header	20 us
1500 bytes at 866 Mbit/s	14 us
SIFS gap	16 us
Acknowledgement frame	24 us

Step	Time
Total	175 us

- Useful data moved: 12,000 bits in 175 microseconds is 68.5 Mbit/s, against an advertised 866.
- That is why frame aggregation exists. Send 64 frames in one burst with one preamble and one block acknowledgement, and the fixed costs are shared.
- With aggregation the same link delivers roughly 500 to 600 Mbit/s of real throughput. Still not 866, but far better than 68.

■ 31.7.4 PLAIN – what is really happening inside

- SIFS**, short interframe space, is the shortest gap, used for replies that must not be interrupted, such as an acknowledgement.
- DIFS**, distributed interframe space, is longer, and is what a device must see idle before starting a fresh transmission.
- After DIFS, the device picks a random number between 0 and the contention window and counts down one slot at a time.
- If the channel goes busy mid-count, the count freezes and resumes after the next idle DIFS. It does not restart, which gives waiting devices fairness.
- On failure the contention window doubles: 15, 31, 63, 127, up to 1023. That is **exponential backoff**, the same idea Ethernet uses.
- The hidden node problem is common: two clients on opposite sides of an access point may be unable to hear each other.
- The fix is **RTS/CTS**. The sender asks with a request to send, and the access point answers with a clear to send that every client hears.
- The clear to send carries a duration. Everyone who hears it sets a countdown called the **network allocation vector** and stays silent for that long, even without hearing the channel busy. That is **virtual carrier sense**.
- Every 802.11 frame starts with a 2 byte frame control field saying what kind it is, then a duration, then up to four addresses.

■ 31.7.5 TECHNICAL – the engineer's version

Parameter	2.4 GHz DSSS	5 GHz OFDM
Slot time	20 us	9 us
SIFS	10 us	16 us
DIFS	50 us	34 us
CWmin	31	15
CWmax	1023	1023

- The 802.11 MAC header layout, in bytes:

```

+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| FC | Dur | Addr1 | Addr2 | Addr3 | Seq | Addr4 | QoS | Body |
| 2 | 2 | 6 | 6 | 6 | 2 | 6* | 2* | ... |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

then FCS: 4

* present only in some frame types

2. Aggregation: A-MSDU packs several IP packets under one MAC header, up to 11,454 bytes in 802.11n. A-MPDU packs several MAC frames under one PHY preamble, up to 1 megabyte in 802.11ac and ax, with a block ACK.
3. As a rule of thumb, TCP throughput lands at 45 to 60 percent of the PHY rate for 802.11n and ac, and 60 to 70 percent for 802.11ax with OFDMA, because OFDMA removes much of the per-station contention.
4. **OFDMA** in 802.11ax divides the channel into resource units measured in subcarriers, called tones.

RU size	Tones	Users in 20 MHz
26-tone	26	up to 9
52-tone	52	up to 4
106-tone	106	up to 2
242-tone	242	1 (whole channel)

5. The access point schedules who uses which resource unit with a trigger frame, so uplink transmissions from several clients arrive simultaneously and must be time-aligned within 0.4 microseconds.
6. **Target wake time** lets a client and access point agree a future window for traffic so the client sleeps in between. It is the single biggest battery feature in 802.11ax and matters most for sensors.
7. Band steering is a vendor behaviour, not a standard: the access point withholds or delays probe responses on 2.4 GHz to push dual-band clients to 5 GHz.
8. The related real standards are 802.11k for neighbour reports, 802.11v for BSS transition management and 802.11r for fast roaming.
9. Capture and inspect: `iw dev wlan0 set type monitor`, then `tcpdump -i wlan0 -e -s0` or Wireshark with the display filter `wlan.fc.type_subtype == 0x08` for beacons.

■ 31.7.6 WORDS – remember these

1. Beacon — the regular announcement an access point sends — a management frame every 100 TU carrying SSID, capabilities and timestamp.
2. BSSID — the identity of one specific access point radio — the MAC address of that radio, unique per band per access point.
3. CSMA/CA — listen, wait a random time, then talk — carrier sense multiple access with collision avoidance and exponential backoff.
4. SIFS — the tiny gap before a reply — short interframe space, 16 us on 5 GHz OFDM, giving replies priority over new transmissions.
5. RTS/CTS — asking permission out loud so hidden devices hear — request to send and clear to send, setting the network allocation vector.
6. OFDMA — several devices sharing one transmission by frequency — scheduled resource units within a channel.
7. TWT — an agreed appointment so a device can sleep — target wake time, introduced in 802.11ax.

31.8 WiFi security

■ 31.8.1 PLAIN – in simple words

1. On radio, everything is in the air and anyone in range receives every bit, so WiFi must encrypt by default and must prove who is who.
2. The first attempt, **WEP** from 1997, was broken beyond repair within four years.

3. **WPA** in 2003 was a patch that ran on the same old hardware. It bought time.
4. **WPA2** in 2004 was the real fix, using the AES cipher. It is still everywhere.
5. **WPA3**, announced in 2018, fixed the last big weakness: an attacker who captured your handshake could guess your password offline, forever.
6. There is also open, with no password, where every byte is readable by anyone in range unless the site itself uses HTTPS.

■ 31.8.2 PLAIN – a picture in your head

1. Think of a locked diary with a code wheel on the front.
2. WEP used one short code and wrote part of it on the outside of every page. After watching enough pages, you could work out the whole code.
3. WPA2 uses a proper safe. The password is not the key: the password plus a fresh random number is stirred into a key used only for this session.
4. WPA3 changes the opening ceremony so the recording is useless. Each guess must be made live, against the real router, one at a time.

Where this comparison breaks: a diary code is one secret. WiFi has a hierarchy of keys, one master key from the password, one pairwise key per client, one group key for broadcasts. Also, encryption stops at the access point. Everything past it is in the clear unless the application encrypts too.

■ 31.8.3 PLAIN – a worked example

1. The absolute rule of a stream cipher is that a keystream must never be used twice. If it is, adding the two ciphertexts cancels the keystream.
2. To vary the keystream, WEP prepends a 24-bit **initialization vector** to the shared key and sends that vector in the clear with every packet.
3. 24 bits gives 16,777,216 values. By the birthday problem a repeat becomes likely after about 5,000 packets, which a busy access point sends in seconds.
4. Worse, RC4 has a related-key weakness, published as the FMS attack by Scott Fluhrer, Itsik Mantin and Adi Shamir in August 2001.
5. The 2007 PTW attack by Erik Tews, Andrei Pyshkin and Ralf-Philipp Weinmann recovers a 104-bit WEP key from about 40,000 packets in under a minute.
6. Now the WPA2 four-way handshake. Both sides already know a master key derived from the password and the network name.

AP	Client
--- 1. ANonce (AP random number) ----->	
client now has both nonces and both	
MACs, so it computes the PTK	
<-- 2. SNonce + MIC (proves it has PTK) ---	
AP computes the same PTK and checks	
--- 3. GTK (encrypted) + MIC ----->	
<-- 4. Acknowledgement + MIC -----	
both install keys, data port opens	

7. Nothing secret crosses the air. Both sides derive the same session key from things they already share plus two random numbers.

■ 31.8.4 PLAIN – what is really happening inside

1. The password and the SSID are stretched into a 256-bit **pairwise master key** by 4096 rounds of hashing. That slowness is deliberate, to make guessing expensive.

2. The four-way handshake mixes that master key with two fresh random numbers and both MAC addresses to make a **pairwise transient key** for this session only.
3. The KRACK attack of October 2017, found by Mathy Vanhoef and Frank Piessens at KU Leuven in Belgium, attacked this handshake.
4. It blocks message four so the access point resends message three. The client reinstalls the same key and resets its packet counter.
5. Reusing a counter with a stream-like cipher exposes the keystream, which is exactly the WEP failure in a new place.
6. On Android and Linux the wpa_supplicant implementation was worse: it installed an all-zero key on reinstallation, removing encryption entirely. That was fixed in wpa_supplicant 2.7.
7. WPA3 replaces the password-to-key step with **SAE**, simultaneous authentication of equals, a password-authenticated key exchange.
8. With SAE a captured handshake cannot be used for offline guessing, and each session gets forward secrecy: learning the password later does not decrypt old recorded traffic.

■ 31.8.5 TECHNICAL – the engineer’s version

Scheme	Year	Cipher	Status
WEP	1997	RC4, 24-bit IV	Broken, deprecated 2004
WPA	2003	RC4 with TKIP	Deprecated 2012
WPA2	2004	AES-CCMP	Still standard
WPA3	2018	AES-CCMP, SAE	Mandatory for new certs

1. WPA2 is IEEE 802.11i-2004. CCMP is AES in counter mode with CBC-MAC, 128-bit key, 48-bit packet number and an 8-byte message integrity code.
2. TKIP was a stopgap that kept RC4 but added per-packet key mixing, a 48-bit sequence counter and the Michael integrity check. The Wi-Fi Alliance barred TKIP-only devices from certification in 2012.
3. Key derivation is $PMK = PBKDF2-HMAC-SHA1(\text{passphrase}, \text{SSID}, 4096 \text{ iterations}, 256 \text{ bits})$. Because the SSID is the salt, common names such as “linksys” have precomputed rainbow tables.
4. KRACK carries ten CVEs, CVE-2017-13077 through CVE-2017-13088.
5. **OWE**, opportunistic wireless encryption, branded Enhanced Open, encrypts open networks with an unauthenticated Diffie-Hellman exchange. It stops passive sniffing but not an active impostor access point.
6. **WPS** is dangerous. Its 8-digit PIN is validated in two halves and the last digit is a checksum, so the search space is 11,000, not 100 million.
7. Stefan Viehbock published this in December 2011, and Dominique Bongard’s 2014 Pixie Dust attack breaks many chipsets offline in seconds. Turn it off.
8. Enterprise uses IEEE 802.1X: the client is the supplicant, the access point the authenticator, and a RADIUS server the authentication server.
9. EAP carries the method, with EAP-TLS using client certificates and PEAP-MSCHAPv2 or EAP-TTLS tunnelling a password inside TLS.
10. Tools: `wpa_cli status`, `hostapd_cli`, `aircrack-ng` for auditing your own network, and Wireshark with the display filter `eapol` to watch the four-way handshake.

■ 31.8.6 WORDS – remember these

1. WEP — the broken first attempt — RC4 with a 24-bit IV, key recoverable from 40,000 packets by the PTW attack.

2. WPA2 — the long-standing standard — IEEE 802.11i with AES-CCMP and the four-way handshake.
3. Four-way handshake — the exchange proving both sides know the password — EAPOL-Key frames deriving the PTK from PMK, two nonces and two MACs.
4. KRACK — the 2017 attack that replayed a handshake message — key reinstallation causing nonce reuse.
5. WPA3 — the current standard — SAE replacing the PSK exchange, giving forward secrecy and no offline dictionary attack.
6. WPS — the push-button feature you should disable — Wi-Fi Protected Setup, whose PIN has an effective search space of 11,000.

31.9 The WiFi software stack

■ 31.9.1 PLAIN – in simple words

1. Five layers of software sit between your click and the air.
2. **Firmware** runs on the WiFi chip itself and handles anything measured in microseconds.
3. The **driver** runs inside the operating system kernel and talks to the chip over PCIe or SDIO.
4. The **802.11 stack** implements the rules of scanning, joining and framing for drivers that do not do it in firmware.
5. The **supplicant** is a background program that does the security handshake and holds your passwords.
6. The **network manager** is the part with the menu you click. It decides which network to join and asks the supplicant to do it.
7. Your password is not sent to the router. It is used locally to compute a key. This is worth repeating because almost everyone assumes otherwise.

■ 31.9.2 PLAIN – a picture in your head

1. Think of joining a members-only club.
2. The network manager is you, reading the list of clubs and picking one.
3. The supplicant is your solicitor, who holds your documents and handles the paperwork at the door.
4. The 802.11 stack is the club's standard entry procedure: queue here, show this, sign that.
5. The driver is the doorman who opens the door, and the firmware is the door mechanism, which must latch in microseconds and cannot wait for anyone.
6. Once inside, DHCP is the receptionist who assigns you a table number.

Where this comparison breaks: in a club these are separate people. On a FullMAC phone chip the procedure, the doorman and the mechanism are one closed piece of firmware, and the operating system only sends high-level requests such as “join this network”.

■ 31.9.3 PLAIN – a worked example

1. Here is exactly what happens in software when you click a network name and type a password. This is the Linux path, the one you can read.

1. You click. The GUI calls NetworkManager over D-Bus.
2. NetworkManager writes a connection profile file.
3. NetworkManager tells wpa_supplicant over D-Bus:
"network={ssid=..., psk=...}, select it".
4. wpa_supplicant asks the kernel to scan, via nl80211.
5. cfg80211 passes the request to the driver; the driver

- programs the chip to sweep channels.
- 6. Scan results come back up the same path as a list of BSSIDs with signal, channel and security info.
- 7. wpa_supplicant picks the best BSSID for that SSID.
- 8. It computes $PMK = PBKDF2(\text{password}, \text{SSID}, 4096, 256)$.
- 9. It asks the kernel to authenticate and associate.
- 10. mac80211 sends the auth and assoc frames on the air.
- 11. The AP replies; the association succeeds.
- 12. The AP sends EAPOL message 1. The driver passes it up.
- 13. wpa_supplicant computes the PTK and sends message 2.
- 14. Messages 3 and 4 complete. Keys go to the kernel.
- 15. The kernel installs keys into the chip's key table.
- 16. The netdev is marked carrier-up. NetworkManager sees it.
- 17. NetworkManager runs its DHCP client. An address arrives.
- 18. Routes and DNS servers are installed. You are online.

- 2. Steps 1 to 3 take milliseconds. The scan is the slow part: a full sweep of all channels takes 1 to 4 seconds, because it must dwell on each channel long enough to catch a beacon.
- 3. Steps 8 to 15 take 20 to 100 milliseconds, and step 17 takes 10 to 100.

Platform	Where the password is stored
Linux, NetworkManager	/etc/NetworkManager/
Linux, plain supplicant	/etc/wpa_supplicant.conf
macOS	login and System keychains
Windows	DPAPI-protected profile XML
Android	/data/misc/wifi, keystore-backed

- 4. A minimal supplicant configuration looks like this:

```
ctrl_interface=/run/wpa_supplicant
update_config=1

network={
    ssid="HomeNet"
    psk="a-long-passphrase"
    key_mgmt=WPA-PSK
    priority=10
}
```

■ 31.9.4 PLAIN – what is really happening inside

- 1. Reconnecting later skips most of that, because the profile is already stored: no typing and no dialogue.
- 2. The device keeps a list of known networks and scans for them on waking.
- 3. Historically it also sent probe requests naming each known SSID, which leaked your home and office network names to anyone listening. Modern devices prefer passive scanning and randomized MAC addresses.
- 4. If the master key has not expired, **PMK caching** lets a device reassociate to a previously used access point without the expensive derivation. Standards 802.11r and opportunistic key caching extend this between access points in one network.
- 5. Roaming decisions live in the client, not the network. The client watches signal and loss, and jumps when a better BSSID crosses a threshold.
- 6. That is why a laptop sometimes clings to a distant access point: the thresholds are a vendor implementation detail, not a standard, and vary widely between chipsets.

- On macOS the same pieces exist with different names: the driver family is IO80211Family, network selection lives in `airportd`, applications use the CoreWLAN framework, and passwords sit in the keychain as items of kind “AirPort network password”.

■ 31.9.5 TECHNICAL – the engineer’s version

- The Linux layering, top to bottom:

```

GUI -> NetworkManager or iwd  (userspace policy)
      | D-Bus
      wpa_supplicant          (802.1X, WPA, SAE)
      | nl80211 over netlink
      cfg80211                (config API, regulatory)
      |
      mac80211                (SoftMAC 802.11 state machine)
      |
      driver: iwlwifi / ath11k / mt76 / brcmfmac
      | PCIe, SDIO, USB
      firmware on the chip

```

- `cfg80211` is the configuration layer and holds the regulatory database. `mac80211` is the SoftMAC implementation, used only by drivers that need it; FullMAC drivers such as `brcmfmac` register directly with `cfg80211`.

Command	Shows
<code>iw dev wlan0 link</code>	BSSID, signal, current bitrate
<code>iw dev wlan0 scan</code>	Beacon contents of neighbours
<code>iw reg get</code>	Active regulatory domain
<code>wpa_cli status</code>	Supplicant state, key management
<code>nmcli device wifi list</code>	NetworkManager’s view
<code>ethtool -i wlan0</code>	Driver and firmware versions

- On macOS 13 and later, `sudo wdistil info` prints RSSI, noise, channel, security, country code and the MAC address in use. The older `airport` utility inside `Apple80211.framework` was removed in macOS 14.4.
- The security boundary is worth stating plainly: your operating system trusts the WiFi firmware completely, and on PCIe that firmware has DMA access unless an IOMMU is configured. That is an established weakness, not a theoretical one.
- Debugging: `dmesg -w` while connecting shows driver and firmware messages, and `wpa_supplicant -dd -i wlan0 -c file.conf` in the foreground prints every state transition, which is the fastest way to see where a join fails.

■ 31.9.6 WORDS – remember these

- Supplicant — the program that does the security handshake — 802.1X supplicant, typically `wpa_supplicant` or `iwd` on Linux.
- `mac80211` — the kernel’s 802.11 rules engine — the SoftMAC framework implementing scanning, association and rate control.
- Network manager — the part with the menu — the userspace policy daemon choosing networks and running DHCP.
- PMK caching — remembering a key so reconnecting is fast — caching the pairwise master key security association per BSSID.

31.10 Bluetooth

■ 31.10.1 PLAIN – in simple words

1. Bluetooth solves a different problem to WiFi. WiFi connects you to a network. Bluetooth connects two devices to each other.
2. It uses the same 2.4 GHz band as WiFi but very differently: it hops between narrow channels 1600 times a second.
3. A small group of connected devices is a **piconet**. One device is in charge of timing and the others follow.
4. Because the purposes vary so much, Bluetooth defines **profiles**: agreed rules for one job, such as stereo audio, a keyboard, or a phone call.
5. **Pairing** is the one-time ceremony creating a shared secret, so the two devices recognize each other later without asking again.

■ 31.10.2 PLAIN – a picture in your head

1. Think of a busy hall where many pairs of people hold private conversations.
2. Each pair agrees a rule: after every sentence, move to a different spot, following a shuffle order only they know.
3. Pairing is the moment two people first exchange a private codeword, in person, so they can recognize each other later in the dark.

Where this comparison breaks: people would collide constantly while moving. The hopping pattern is derived from the leader's address and clock, so every follower knows exactly where the leader will be next. Also, since Bluetooth 1.2 in 2003, devices drop persistently busy channels from the hop list, which no wandering person could do.

■ 31.10.3 PLAIN – a worked example

Version	Year	What it added
1.0	1999	First release, 723 kbit/s
1.2	2003	Adaptive hopping, faster connect
2.0 + EDR	2004	Enhanced data rate, 2.1 Mbit/s
2.1 + EDR	2007	Secure Simple Pairing
3.0 + HS	2009	WiFi used for bulk transfer
4.0	2010	Bluetooth Low Energy, GATT
4.2	2014	Better privacy, IPv6 over BLE
5.0	2016	BLE 2 Mbit/s or 4x range
5.1	2019	Direction finding, AoA and AoD
5.2	2019	LE Audio, LC3 codec, isochronous
5.3	2021	Connection subrating, efficiency
5.4	2023	Encrypted advertising, PAWR
6.0	2024	Channel sounding for distance

1. The radio numbers: 79 channels of 1 MHz each from 2402 to 2480 MHz, hopped 1600 times a second, so each hop lasts 625 microseconds.

Profile	Job
A2DP	Stereo music to a speaker

Profile	Job
AVRCP	Play, pause, skip, volume
HFP	Phone calls, two-way voice
HID	Keyboard, mouse, game pad
SPP	A plain serial cable replacement
PBAP	Sharing contacts with a car

■ 31.10.4 PLAIN – what is really happening inside

1. One device in a piconet is the **central**, historically called master, and up to seven others are actively connected **peripherals**.
2. The central's clock and address generate the hop sequence. Time is divided into 625 microsecond slots and the central decides who transmits in each.
3. Above the radio sit a link layer, a multiplexing layer called L2CAP, then the profiles. Profiles exist so a headset from one maker works with a phone from another with no shared code.
4. **Legacy pairing**, before 2007, used a short PIN, often 0000, and a passive listener who captured the exchange could recover the link key.
5. **Secure Simple Pairing**, in Bluetooth 2.1 from 2007, uses elliptic curve Diffie-Hellman so a passive listener learns nothing.
6. It offers four association models, chosen by what each device can display and what you can type into it.
7. **Numeric comparison**: both show a six-digit number and you confirm they match. This is the strongest common model.
8. **Passkey entry**: one shows a number and you type it into the other, used for keyboards.
9. **Just Works**: neither device can display or input, so nothing is checked. It stops passive eavesdropping but not an active impostor.
10. The result is a **link key** stored on both sides. Storing it is **bonding**, and it is why a headset reconnects silently forever after.

■ 31.10.5 TECHNICAL – the engineer's version

1. Development began at Ericsson Mobile in Lund, Sweden, where Nils Rydbeck started a short-link radio project in 1989. Design work began in 1994, led by Jaap Haartsen with Sven Mattisson.
2. The Bluetooth Special Interest Group launched in May 1998 with five founding members: Ericsson, IBM, Intel, Nokia and Toshiba. The specification was announced on 20 May 1998.
3. The name was proposed by Jim Kardach of Intel in 1997 after Harald "Bluetooth" Gormsson, the tenth-century Danish king who united the Danish tribes. It was meant as a code name and stuck.
4. The controller and host are separated by the **HCI**, host controller interface, over UART, USB or SDIO. That split is why one chip works with Linux BlueZ, Windows and Android alike.
5. Attacks worth naming: BlueBorne, published by Armis in September 2017, gave remote code execution with no pairing.
6. KNOB, by Antonioli, Tuppenhauer and Rasmussen in 2019, forced the encryption key down to one byte of entropy. BIAS in 2020 impersonated a previously bonded device.
7. The honest version of "Bluetooth is secure": the cryptography in Secure Connections mode, added in 4.2 using P-256 ECDH and AES-CCM, is sound.
8. The weaknesses are almost all in negotiation, downgrade paths and Just Works, plus a long tail of unpatched embedded devices.

■ 31.10.6 WORDS – remember these

1. Piconet — a small group sharing timing — one central and up to seven active peripherals on a common hop sequence.
2. Frequency hopping — jumping between narrow channels constantly — 1600 hops per second across 79 channels of 1 MHz.
3. Profile — an agreed rule set for one job — a specification such as A2DP or HFP guaranteeing cross-vendor interoperability.
4. Pairing — the one-time ceremony creating a shared secret — Secure Simple Pairing with ECDH and one of four association models.
5. HCI — the boundary between chip and operating system — host controller interface over UART, USB or SDIO.

31.11 Bluetooth Low Energy

■ 31.11.1 PLAIN – in simple words

1. Bluetooth Low Energy shares the name and the band with classic Bluetooth and almost nothing else. It is a separate protocol.
2. It was created for devices that send tiny amounts of data occasionally and must run for a year on a coin cell: a fitness band, a door sensor, a car key.
3. Instead of building a long-lived connection, a device may simply shout a tiny message every so often. That is **advertising**.
4. If more is needed, a scanner can ask to connect, and then the two exchange named pieces of data organized as **services** and **characteristics**.

■ 31.11.2 PLAIN – a picture in your head

1. Classic Bluetooth is a telephone call: you dial, you stay connected, and you pay for the whole time whether you speak or not.
2. BLE is a postcard on a noticeboard. The device pins a small card up every second, then goes back to sleep.
3. If you want more than the card says, you knock on the door, and the device opens a short conversation from a printed menu of what it can tell you.
4. That menu is the GATT database: sections are services, items are characteristics.

Where this comparison breaks: a noticeboard is passive. A BLE advertisement costs real energy each time, which is why the advertising interval is the main battery knob. Longer intervals mean longer life and slower discovery.

■ 31.11.3 PLAIN – a worked example

1. Take a heart rate monitor chest strap. Its GATT database looks like this.

```
Service: Heart Rate (UUID 0x180D)
  Characteristic: Heart Rate Measurement (0x2A37)
    properties: NOTIFY
    value: flags byte + rate, e.g. 00 4B -> 75 bpm
  Characteristic: Body Sensor Location (0x2A38)
    properties: READ
    value: 01 -> chest

Service: Battery Service (UUID 0x180F)
  Characteristic: Battery Level (0x2A19)
```

```
properties: READ, NOTIFY
value: 5B -> 91 percent
```

- The phone does not poll for the heart rate. It subscribes once, and the strap sends a notification each time the value changes.
- Power arithmetic: a CR2032 coin cell holds about 225 milliamp-hours. Advertising once per second at 10 milliamps for 2 milliseconds averages about 20 microamps, plus 2 microamps sleeping. That is over 1,000 days.

■ 31.11.4 PLAIN – what is really happening inside

- Here is a BLE device from power-on to data flowing.

- Power on. The chip configures its radio and loads its GATT table from flash into memory.
- It picks an advertising interval, say 1000 ms.
- Every interval it wakes and transmits the same short packet on channels 37, 38 and 39 in turn, then sleeps.
- A phone scanning hears one of those three channels.
- The phone may send a scan request; the device answers with a scan response holding extra bytes, e.g. its name.
- The user taps connect. The phone sends a connect request carrying an access address, a hop increment and a connection interval.
- Both sides move onto the 37 data channels, hopping by the agreed increment.
- At each connection interval the peripheral wakes, listens for the central's packet, replies, and sleeps again.
- The phone discovers services, then characteristics.
- The phone writes 0x0001 to the heart rate CCCD.
- The strap now sends a notification every beat, inside the normal connection events. No polling.

- Privacy is a real concern. A device advertising the same address forever is trackable, so BLE supports **resolvable private addresses** that change every fifteen minutes and can be linked back only by a device holding the identity resolving key.
- Beacons are advertisements with a fixed payload and no connection. Apple's iBeacon, from 2013, and Google's Eddystone, from 2015, are two formats.
- They enable indoor positioning and also silent presence tracking, which is why both Android and iOS gate scanning behind a location permission.

■ 31.11.5 TECHNICAL – the engineer's version

- BLE was developed by Nokia as Wibree from 2006, adopted by the Bluetooth SIG and published in the Bluetooth 4.0 core specification in June 2010.
- Radio: 40 channels of 2 MHz from 2402 to 2480 MHz, with GFSK modulation.
- Speeds are 1 Mbit/s at LE 1M and 2 Mbit/s at LE 2M, added in 5.0, plus a coded PHY at S=2 or S=8 giving four times the range at one eighth the rate.
- Stack, bottom to top: link layer, L2CAP, then ATT (attribute protocol) and GATT (generic attribute profile), with SMP (security manager protocol) alongside for pairing.
- ATT is the wire protocol: a flat table of handles, each with a UUID, permissions and a value, with operations read, write, write without response, notify and indicate.
- GATT is the convention layered on ATT that groups attributes into services and characteristics. It is a profile, not a separate protocol.

7. Since BLE 4.2, data length extension raises the link layer payload from 27 to 251 bytes, roughly tripling throughput.
8. Tools: `bluetoothctl` and `btmon` on Linux, the nRF Connect application on Android and iOS for exploring a GATT table, and a Nordic nRF52 sniffer with Wireshark for over-the-air capture.

■ 31.11.6 WORDS – remember these

1. Advertising — shouting a small packet periodically — unconnected broadcast on channels 37, 38 and 39.
2. GATT — the small named database on a device — generic attribute profile organizing ATT handles into services and characteristics.
3. Characteristic — one named value you can read or subscribe to — an ATT attribute with a UUID, properties and optional descriptors.
4. Resolvable private address — an address that changes to avoid tracking — a rotating address resolvable only with the identity resolving key.

31.12 Bluetooth audio

■ 31.12.1 PLAIN – in simple words

1. Bluetooth audio was poor for a long time, and the reasons explain most of what people complain about.
2. Every device must support one basic method so anything works with anything. That method, SBC, was designed in 2002 to be cheap, not good.
3. Squeezing and unsqueezing takes time. That delay is **latency**, and it is why lips can be out of step with sound when watching video.
4. Worst of all, when you use the microphone the link switches to a different, much narrower mode and quality collapses.
5. That is why music sounds fine until a call starts, and then everything sounds like a 1990s telephone.

■ 31.12.2 PLAIN – a picture in your head

1. Think of a narrow letterbox. Thin envelopes go through quickly.
2. Now you want a two-way conversation through the same letterbox. Half the space must go to your outgoing envelopes.
3. With half the space gone, the incoming book must be squashed far harder, into something barely readable. That is what happens when the microphone opens.

Where this comparison breaks: a letterbox is only about width. The old profile also switched to a completely different, much older codec, so the damage was larger than a simple halving of bandwidth. LE Audio fixes this by using one modern codec in both directions.

■ 31.12.3 PLAIN – a worked example

Codec	Owner	Bitrate	Notes
SBC	Bluetooth SIG	192–345 kbit/s	Mandatory, baseline
AAC	MPEG	128–256 kbit/s	Apple devices default
aptX	Qualcomm	352 kbit/s	Fixed rate, low latency
aptX HD	Qualcomm	576 kbit/s	24-bit capable
LDAC	Sony	330–990 kbit/s	Highest rate, adaptive
LC3	SIG, 2020	160–345 kbit/s	LE Audio, best per bit

1. Latency figures, approximate and highly device dependent: SBC 150 to 250 ms, AAC on Apple devices 100 to 160 ms.
2. aptX Low Latency is 32 to 40 ms, LC3 over LE Audio is 20 to 30 ms, and a wired 3.5 mm jack is under 1 ms.
3. Humans notice audio and video out of step at roughly 45 milliseconds of audio lag. That is why video players delay the picture when Bluetooth is in use, and why gaming over Bluetooth feels wrong.

■ 31.12.4 PLAIN – what is really happening inside

1. Two different profiles handle audio, and that is the root of the problem.
2. **A2DP**, the advanced audio distribution profile, is one-way, high quality and used for music.
3. **HFP**, the hands-free profile, is two-way and used for calls.
4. When a call starts the headset switches from A2DP to HFP. Classic HFP originally offered only the CVSD codec at 8 kHz sampling, telephone quality, later improved to mSBC at 16 kHz, called wideband speech.
5. **LE Audio**, introduced in Bluetooth 5.2 in 2019 and shipping in products from 2022, replaces both. It uses one codec, **LC3**, developed by Fraunhofer IIS and Ericsson, in both directions over new isochronous channels that guarantee timing.
6. LC3 at 160 kbit/s is judged better than SBC at 345 kbit/s in listening tests run by the Bluetooth SIG, so it gives better sound at half the bitrate, which also means longer battery life.
7. **Auracast** is LE Audio's broadcast mode. One transmitter serves unlimited receivers, so a television, an airport gate or a lecture hall can broadcast audio anyone nearby joins, optionally with a password.
8. The Bluetooth SIG announced the Auracast brand in 2022.

■ 31.12.5 TECHNICAL – the engineer's version

1. HFP uses a SCO or eSCO link, which is synchronous and reserves fixed slots.
2. CVSD is 64 kbit/s at 8 kHz; mSBC is a constrained SBC configuration at 16 kHz over eSCO, defined in HFP 1.6 in 2009.
3. There is genuine expert disagreement about whether LDAC and aptX HD are audibly better than well-implemented AAC. Blind listening tests generally fail to separate them at 256 kbit/s and above.
4. Treat "hi-res Bluetooth" as a marketing claim, and codec efficiency at low bitrate as the engineering fact.
5. LE Audio components: the LC3 codec, isochronous channels (CIS for connected, BIS for broadcast), the Basic Audio Profile, and the Public Broadcast Profile for Auracast.
6. Inspect on Linux with `pactl list sinks` or `bluetoothctl` plus PipeWire or BlueZ logs, which name the negotiated codec. On Android, Developer options shows the Bluetooth audio codec, sample rate and bit depth.

■ 31.12.6 WORDS – remember these

1. HFP — the profile for calls — hands-free profile over a synchronous SCO or eSCO link, historically CVSD then mSBC.
2. LC3 — the modern codec of LE Audio — low complexity communication codec, better quality per bit than SBC.
3. LE Audio — the redesign that fixes call quality — Bluetooth 5.2 isochronous channels with LC3 in both directions.

4. Auracast — broadcasting audio to many listeners — LE Audio broadcast isochronous streams, branded by the SIG in 2022.

31.13 Cellular

■ 31.13.1 PLAIN – in simple words

1. WiFi covers a room. Cellular covers a country with the same amount of spectrum, by reusing it over and over.
2. The trick is the **cell**. Divide the land into patches and give each patch a low-power transmitter that covers only that patch.
3. Two patches far enough apart can use the same frequency without hearing each other. That is **frequency reuse**, and it is the whole idea.
4. As you move you leave one cell and enter another, and the network moves your call across without dropping it. That is **handover**.
5. Unlike WiFi, cellular spectrum is licensed and expensive, so the operator controls exactly who transmits and when. There is no random backoff.

■ 31.13.2 PLAIN – a picture in your head

1. Imagine a country covered in hexagonal tiles, like a bathroom floor.
2. Seven colours are enough that no two touching tiles share a colour. Each colour is a set of frequencies.

Where this comparison breaks: real cells are not hexagons and not equal. Coverage is a messy blob shaped by hills and buildings, cells overlap heavily, and modern networks reuse every frequency in every cell, relying on coding and interference coordination instead of colour separation.

■ 31.13.3 PLAIN – a worked example

Gen	Years	Peak rate	What changed
1G	1979–1991	analogue voice	Cellular concept, AMPS/NMT
2G	1991	9.6 kbit/s	Digital voice, SMS, SIM
2.5G	2000–2003	56–384 kbit/s	Packet data: GPRS, EDGE
3G	2001	384 kbit/s–42 Mbit/s	UMTS, CDMA, mobile web
4G	2009	100 Mbit/s–1 Gbit/s	All-IP, OFDMA, no circuits
5G	2019	20 Gbit/s peak spec	mmWave, slicing, low latency

1. GSM, the dominant 2G system, was agreed by 15 representatives from 13 European countries in Copenhagen in 1987 and launched commercially in Finland in December 1991, with the first call on 1 July 1991.
2. GSM voice used a 13 kbit/s full-rate codec and 9.6 kbit/s circuit-switched data. GPRS services launched in 2000 and EDGE first ran in 2003.
3. 5G frequency ranges: FR1 is everything below 7.125 GHz with channels up to 100 MHz; FR2 is 24.25 to 71 GHz with channels up to 400 MHz.

■ 31.13.4 PLAIN – what is really happening inside

1. The SIM card is a small computer, not a memory card. It runs a program and holds a secret it will never reveal.
2. It holds the ICCID, the card's serial number; the IMSI, which identifies the subscriber; and Ki, a 128-bit secret key.

3. The **IMEI** is different and lives in the phone, not the SIM. It identifies the handset, and is what a stolen-phone blacklist uses.
4. Authentication never sends the key. The network sends a random challenge, the SIM computes a response using K_i , and returns it. The network computes the same response independently, and if they match you are in.
5. In 2G this was one-way: the phone proved itself to the network but not the reverse, which is why fake base stations, IMSI catchers, worked. 3G onwards added mutual authentication.
6. Roaming works because your home network holds K_i and the visited network does not. The visited network asks your home network for challenge and response pairs, and uses them without learning the key.

What happens when a phone attaches

1. Power on. Radio scans bands for the strongest cell.
 2. It reads broadcast information: operator, cell ID, tracking area, and how to send a request.
 3. It sends a random-access preamble on a shared channel.
 4. The network grants a small uplink allocation.
 5. The phone sends an attach request with an identity, ideally a temporary one, not the IMSI.
 6. Authentication challenge and response, both directions.
 7. Ciphering and integrity protection are switched on.
 8. The core network sets up a default data bearer.
 9. An IP address is assigned, usually behind carrier NAT.
 10. Data flows. The phone is paged in its tracking area.
7. Now tie this back to the reader's home broadband from Chapter 29. On home broadband your router gets an address and your packets go straight into the ISP's network.
 8. On mobile, packets are first carried inside a tunnel from the tower all the way to a gateway deep in the operator's core, and only there do they enter the internet.
 9. It is also why the reader's own session showed `github.com` loading instantly on mobile data while timing out on home broadband: the two paths into the internet were entirely different networks with different routing.

■ 31.13.5 TECHNICAL – the engineer's version

1. GSM was specified by ETSI, then everything from 3G onward by 3GPP, which publishes numbered Releases.
2. LTE is Release 8 of 2008; 5G New Radio is Release 15 of 2018, with Release 16 in 2020 adding ultra-reliable low latency and Release 17 in 2022 adding reduced-capability devices.
3. IMSI is 15 digits: a 3-digit mobile country code, a 2 or 3 digit network code, and the subscriber number. India's country codes are 404 and 405. IMEI is 15 digits with a Luhn check digit.
4. Authentication algorithms: 2G used COMP128 variants with A5/1 or the deliberately weakened A5/2 for ciphering, both broken. 3G, 4G and 5G use MILENAGE or TUAK with AES, giving mutual authentication.
5. 5G added SUCI, the subscription concealed identifier, which encrypts the subscriber identity with the operator's public key, closing the IMSI-catcher hole that persisted through 4G.
6. **Network slicing** is a Release 15 feature: several logical networks with separate quality guarantees run over one physical network, identified by an S-NSSAI.
7. Slicing is deployed commercially, but its widely promoted use cases remain partly a marketing claim rather than a mass reality as of 2026.

■ 31.13.6 WORDS – remember these

1. SIM — the small computer holding your identity — a UICC storing IMSI and Ki and computing authentication responses.
2. IMSI — who the subscriber is — 15-digit identity, concealed as SUCI in 5G.
3. IMEI — which handset it is — 15-digit equipment identity with a Luhn check.
4. VoLTE — voice carried as data — SIP over IMS on a dedicated LTE bearer.
5. Network slicing — several virtual networks on one physical one — 3GPP Release 15 S-NSSAI-identified logical networks.

31.14 GPS and positioning

■ 31.14.1 PLAIN – in simple words

1. GPS satellites do not know where you are and never hear from you. They only broadcast. Your receiver does all the work.
2. Each satellite broadcasts the same thing over and over: “I am satellite number seven, here is exactly where I am, and the time is now”.
3. Several arrive at slightly different times, because each travelled a different distance. From the delay, the receiver works out each distance.
4. Distance to one satellite puts you somewhere on a sphere around it. Two narrow it to a circle. Three narrow it to two points, one of them absurd, out in space.
5. So three would be enough if your clock were perfect. It is not.
6. Your phone’s clock is out by thousandths of a second, and light travels 300 kilometres in a thousandth of a second. That error is enormous.
7. So the receiver treats its own clock error as a fourth unknown and solves for it. Four unknowns need four measurements, so you need four satellites.

■ 31.14.2 PLAIN – a picture in your head

1. Imagine standing in fog with four lighthouses, each sounding a horn at exactly noon.
2. You hear them at slightly different moments, and since sound takes time, the delay tells you the distance to each. If your watch is right, three horns fix your position.
3. If your watch is wrong by a fixed amount, all four distances are wrong by the same amount. That common error is a fingerprint you can solve for.
4. You find the one position and one clock correction that make all four distances consistent. That is trilateration with a clock unknown.

Where this comparison breaks: sound is slow and the satellites move at about 3.9 km per second, so the receiver must also account for satellite motion, the rotation of the Earth during the signal’s travel time, and the slowing of radio by the ionosphere.

■ 31.14.3 PLAIN – a worked example

Property	Value
Satellites	24 baseline, 31 usually active
Orbital planes	6, inclined 55 degrees
Altitude	about 20,200 km
Orbital period	11 h 58 min
Civil signal L1	1575.42 MHz
C/A code rate	1.023 Mchip/s

1. Relativity: the satellite clock runs slow by about 7 microseconds a day due to its speed, and fast by about 45 microseconds a day due to weaker gravity. Net, it runs about 38 microseconds per day fast.
2. Uncorrected, that is 38 microseconds times 300 metres per microsecond, about 11 kilometres of drift per day. GPS is engineering that depends on general relativity to work at all.

Start type	What is known	Time to fix
Cold	Nothing	30 s to 15 min
Warm	Almanac, rough position	20 to 45 s
Hot	Recent ephemeris, time	1 to 5 s
Assisted	Data via mobile network	under 2 s

■ 31.14.4 PLAIN – what is really happening inside

1. The GPS signal arrives weaker than the background noise, around -130 dBm against a noise floor near -111 dBm in the same bandwidth. You cannot see it on a spectrum analyser.
2. It is recovered by correlation. Each satellite modulates its signal with a unique 1023-chip code that repeats every millisecond.
3. The receiver generates the same code locally and slides it against the received noise, looking for the offset where the two line up.
4. When they line up, the wanted signal adds coherently over 1023 chips and the noise does not, giving roughly 30 dB of processing gain and lifting the signal above the noise.
5. Once locked, the receiver reads the navigation message at 50 bits per second. A full frame takes 30 seconds; the complete almanac takes 12.5 minutes.
6. That is the real reason a cold start can take 15 minutes. It is waiting to download data at 50 bits per second, not struggling to hear.
7. **Assisted GPS** delivers that same data over the mobile network in milliseconds, plus a rough position and time, so the search space shrinks.
8. A phone rarely uses GPS alone. It fuses satellites, WiFi access point locations from a crowd-sourced database, cell tower positions, the accelerometer and gyroscope for dead reckoning, and the barometer for floor level.
9. Indoors the WiFi database usually dominates, which is why a phone knows which shop you are in with no satellite in sight.

■ 31.14.5 TECHNICAL – the engineer's version

1. The first GPS prototype launched in 1978. The full 24-satellite constellation became operational in 1993, and Full Operational Capability was declared in April 1995.
2. Selective Availability, the deliberate degradation of civil accuracy to about 100 metres, was switched off on 1 May 2000, immediately improving civilian accuracy to roughly 10 metres.

System	Country	Satellites	Notes
GLONASS	Russia	24	Operational since 1995
Galileo	EU	24 plus spares	Services from 2016
BeiDou-3	China	30	Global from 2020
NavIC	India	7	Regional, from 2018

3. NavIC, run by ISRO and formerly called IRNSS, covers India and about 1,500 km beyond, uses L5 and S-band, and is mandated for some vehicle tracking and emergency use in India.

■ 31.14.6 WORDS – remember these

1. Trilateration — finding a position from distances — solving for x, y, z from ranges to several known points.
2. Assisted GPS — getting satellite data over the network — A-GPS delivering ephemeris, time and rough position.
3. Processing gain — how a signal below the noise is recovered — about 30 dB from correlating against the 1023-chip C/A code.
4. GNSS — the general name for all such systems — GPS, GLONASS, Galileo, BeiDou and regional NavIC.

31.15 The other short-range radios

■ 31.15.1 PLAIN – in simple words

1. WiFi and Bluetooth are not the only radios in your life. Several others sit in niches where those two are wrong.
2. **NFC** works over four centimetres and can power the other device: a card with no battery works because the reader's field energizes it.
3. **RFID** is the same family, used for shop tags, warehouse pallets and animal microchips.
4. **Zigbee**, **Z-Wave** and **Thread** are for home sensors: tiny messages, years of battery, and a mesh so distant devices relay for each other.
5. **Matter** is not a radio at all. It is a common language spoken over Thread, WiFi and Ethernet so devices from different makers work together.
6. **LoRa** trades speed for distance: a few hundred bits per second, but kilometres of range through buildings.
7. **UWB** measures distance very precisely by timing a very short pulse, to within about ten centimetres.

■ 31.15.2 PLAIN – a picture in your head

1. Think of a toolbox. WiFi is the power drill: fast, hungry, plugged in. Bluetooth is the screwdriver: handy, personal, battery powered.
2. NFC is the key you touch to a lock: no power of its own, works only when almost touching.
3. LoRa is a foghorn: barely any information, but heard for miles. UWB is a tape measure.

Where this comparison breaks: tools do not interfere with each other. Zigbee, Thread, WiFi and Bluetooth all share 2.4 GHz and do interfere, which is why Zigbee channels are chosen to sit in the gaps between WiFi channels 1, 6 and 11, and why a busy WiFi network can make smart bulbs unreliable.

■ 31.15.3 PLAIN – a worked example

Technology	Band	Rate	Range
NFC	13.56 MHz	106–424 kbit/s	under 10 cm
Zigbee	2.4 GHz	250 kbit/s	10–100 m mesh
Z-Wave	865–926 MHz	100 kbit/s	30–100 m mesh
Thread	2.4 GHz	250 kbit/s	10–100 m mesh
LoRa	865–928 MHz	0.3–27 kbit/s	5–15 km
UWB	3.1–10.6 GHz	up to 27 Mbit/s	10–50 m
BLE	2.4 GHz	0.125–2 Mbit/s	10–100 m

1. NFC data rates are 106, 212 or 424 kbit/s under ISO/IEC 18092.

■ 31.15.4 PLAIN – what is really happening inside

1. NFC is not radiation. It is transformer action across a gap.
2. The reader drives a coil at 13.56 MHz, creating a strong alternating magnetic field in the near field, within a few centimetres.
3. A tag's coil sits in that field, and the changing field induces a current that powers the tag's tiny chip.
4. The tag replies not by transmitting but by changing how much energy it absorbs, which the reader detects as a change in its own coil's load. That is **load modulation**.
5. That is why an NFC card works with no battery, and why the range is physically limited: near-field magnetic strength falls off as the cube of distance, not the square.
6. Zigbee and Thread both run on the IEEE 802.15.4 radio, a standard from 2003 defining 250 kbit/s at 2.4 GHz using offset QPSK.
7. Zigbee, from the Zigbee Alliance formed in 2002 with its first specification in 2004, defines its own application layer above that radio.
8. Thread, launched by the Thread Group in July 2014, keeps 802.15.4 but runs IPv6 over it using 6LoWPAN header compression, so every device is a real IP host with no translating gateway.
9. Matter, from the Connectivity Standards Alliance, released version 1.0 on 4 October 2022. It began in December 2019 as Project Connected Home over IP, founded by Amazon, Apple and the Zigbee Alliance.
10. LoRa uses chirp spread spectrum: each symbol is a sweep across the band rather than a fixed tone, and a sweep is easy to detect far below the noise floor.
11. The **spreading factor**, 5 to 12, sets how long each chirp lasts, trading range against data rate.
12. UWB sends pulses under two nanoseconds long, spread over at least 500 MHz. Because the pulse is so short, its arrival time can be measured very precisely, which is what gives centimetre ranging.

■ 31.15.5 TECHNICAL – the engineer's version

1. NFC standards: ISO/IEC 18092 and ECMA-340 define NFCIP-1, ISO/IEC 14443 covers proximity cards, and ISO/IEC 15693 covers vicinity cards at about one metre. The NFC Forum was founded on 18 March 2004 by Nokia, Philips and Sony.
2. RFID spans several bands: 125 to 134 kHz for animal tags, 13.56 MHz for smart cards, and 860 to 960 MHz for retail and logistics, where passive tags read at up to about 10 metres by backscatter.
3. Z-Wave was created by Zensys in Denmark around 1999, acquired by Sigma Designs and then by Silicon Labs in 2018.
4. Its band is region-specific: 908.42 MHz in the United States, 868.42 MHz in Europe, 865.2 MHz in India. Being sub-gigahertz, it avoids 2.4 GHz congestion entirely.
5. UWB was authorized for unlicensed use by the FCC in February 2002 from 3.1 to 10.6 GHz, with a power spectral density limit of -41.3 dBm/MHz.
6. The standards are IEEE 802.15.4a and the newer 802.15.4z, which added secure ranging with scrambled timestamp sequences.
7. UWB ranging uses two-way time of flight, so a clock offset between the two devices cancels. Ten centimetre accuracy is routine.
8. Car makers use UWB for digital keys under the Car Connectivity Consortium Digital Key 3.0 specification, because ranging defeats the relay attacks that break Bluetooth and radio key fobs.
9. LPWAN alternatives to LoRa: NB-IoT and LTE-M are 3GPP standards from Release 13 in 2016, running in licensed cellular spectrum, with NB-IoT using a 180 kHz carrier.
10. Sigfox was a proprietary ultra-narrowband system from a French company founded in 2010; it entered receivership in 2022, which is a useful caution about proprietary networks.

■ 31.15.6 WORDS – remember these

1. NFC — a tap-range link that powers the other device — 13.56 MHz inductive coupling with load modulation, ISO/IEC 18092.
2. RFID — tagging things with a chip and no battery — passive backscatter tags at 125 kHz, 13.56 MHz or 860-960 MHz.
3. Zigbee — a mesh for home sensors with its own application layer — IEEE 802.15.4 radio plus the Zigbee stack, from 2004.
4. Thread — an IPv6 mesh for home sensors — IEEE 802.15.4 plus 6LoWPAN, from the Thread Group in 2014.
5. Matter — one language so smart devices interoperate — a CSA application layer over IP, version 1.0 released October 2022.
6. LoRa — very long range at very low data rates — chirp spread spectrum with spreading factors 5 to 12.
7. UWB — measuring distance to a few centimetres — sub-nanosecond pulses over 500 MHz or more, IEEE 802.15.4z two-way ranging.

31.98 Common wrong ideas

1. Wrong: WiFi and the internet are the same thing. Right: WiFi is only the last few metres of radio. A working WiFi link with a broken uplink gives full signal bars and no internet at all.
2. Wrong: more bars means faster. Right: bars show received signal strength, only one input. A strong signal on a congested channel is slower than a weaker signal on a clear one.
3. Wrong: the advertised router speed is what you get. Right: an “AX5400” figure adds every band and stream together. Real single-client throughput is typically 40 to 60 percent of one band’s headline number.
4. Wrong: 5 GHz is always better than 2.4 GHz. Right: 5 GHz carries more data but loses about 6.5 dB more over distance and roughly twice as much through each wall. Through two brick walls, 2.4 GHz usually wins.
5. Wrong: your WiFi password is sent to the router. Right: it never leaves your device. Both sides derive the same key from it, and only random numbers and proofs cross the air.
6. Wrong: Bluetooth and Bluetooth Low Energy are versions of one protocol. Right: they share a brand, a band and a chip, and are otherwise different protocols with different radios, packet formats and stacks.
7. Wrong: Hedy Lamarr invented WiFi. Right: she and George Antheil patented frequency hopping in 1942, an idea Bluetooth uses and WiFi mostly does not.
8. Wrong: GPS satellites track your phone. Right: satellites only broadcast, and a GPS receiver transmits nothing. Anything tracking you does so through the mobile or WiFi network, not through GPS.

31.99 Chapter summary in 20 lines

1. A radio wave is made when alternating current in a conductor sheds a self-sustaining pair of electric and magnetic fields into space.
2. Wavelength is the speed of light divided by frequency: 12.3 cm at 2.4 GHz and 5.8 cm at 5 GHz, which is why antennas are the sizes they are.
3. Radio is light at a frequency you cannot see, and its photons are far too weak to break chemical bonds.

4. Information rides on a wave by changing amplitude, frequency or phase, and modern systems change all of them at once as QAM.
5. More bits per symbol needs a cleaner signal: 1024-QAM needs about 35 dB signal-to-noise ratio where BPSK needs 2 dB.
6. Spread spectrum trades bandwidth for robustness, by hopping in Bluetooth or by spreading with a code in 802.11b and GPS.
7. OFDM sends many slow narrow subcarriers instead of one fast wide one, and its guard interval swallows echoes. It underlies WiFi, LTE and 5G.
8. Signal is measured in dBm against a noise floor near -96 dBm in a 20 MHz channel, and the gap between them decides your data rate.
9. WiFi exists because the FCC opened the ISM bands to unlicensed spread spectrum in 1985, and because ALOHAnet in 1971 had shown how to share a channel by random access.
10. The standards run from 802.11-1997 at 2 Mbit/s to 802.11be in 2024, with Wi-Fi 8 as 802.11bn expected around 2028 aiming at reliability, not speed.
11. CSIRO's US patent 5,487,069 of 1996 covers a real technique WiFi uses and earned about 430 million dollars, but WiFi has no single inventor.
12. On the air, devices listen, wait a random backoff, transmit and wait for an acknowledgement, because a transmitter cannot hear a collision.
13. Overheads mean real throughput is roughly half the advertised rate; aggregation, OFDMA and MU-MIMO exist to claw that back.
14. WEP failed because a 24-bit initialization vector repeats within minutes, WPA2 fixed it with AES-CCMP, KRACK broke the handshake's assumptions in 2017, and WPA3 with SAE removed offline password guessing.
15. In software, firmware, driver, `cfg80211` and `mac80211`, a supplicant and a network manager cooperate, and your password becomes a key locally.
16. Bluetooth began at Ericsson in 1994 with Jaap Haartsen, was published by a five-company SIG in 1998, and hops 1600 times a second across 79 channels.
17. Bluetooth Low Energy is a separate protocol using advertising, connections and a small named database called GATT, designed for coin cells.
18. Cellular reuses the same spectrum in every cell, keeps tight central control, and authenticates with a secret the SIM never reveals.
19. GPS needs four satellites because the receiver's clock is a fourth unknown, and it depends on a relativistic correction of 38 microseconds per day.
20. NFC, RFID, Zigbee, Thread, Matter, LoRa and UWB each exist because WiFi and Bluetooth are the wrong tool for some job.

TLS, HTTPS and the Web Stack

32.0 What this chapter gives you

1. You will be able to write a complete HTTP request by hand, byte by byte, and read a real response header block without guessing at any line.
2. You will be able to name the three separate problems that TLS solves, and say why solving only the first one is close to useless.
3. You will be able to explain symmetric encryption, hashing, message authentication, public keys and key exchange in plain words, and then work a Diffie-Hellman exchange by hand with small numbers.
4. You will be able to walk through both the TLS 1.2 and the TLS 1.3 handshake message by message, and say exactly what TLS 1.3 removed.
5. You will be able to read an X.509 certificate field by field with `openssl`, and say what every field is for.
6. You will be able to explain the chain of trust from leaf to intermediate to root, say where the root store lives on each operating system, and say who decides what goes into it.
7. You will be able to state precisely what a browser checks before it shows the padlock, and, more importantly, the long list of things it does not check.
8. You will be able to explain why accepting a self-signed certificate on the reader's own router at `192.168.0.1` is a reasonable risk, and why accepting the identical warning on a public site destroys everything TLS was for.
9. You will be able to spot a company proxy or an antivirus product opening your encrypted traffic, using one command and one line of its output.
10. You will be able to describe HTTP/1.1, HTTP/2 and HTTP/3 and say what problem each new version was built to fix.

32.1 HTTP first, because TLS wraps it

■ 32.1.1 PLAIN – in simple words

1. HTTP is the language a browser and a web server use to talk.
2. It is text. Real, readable, typed-out text. Not a secret binary format.
3. The conversation has exactly two parts: you ask, the server answers.
4. The thing you send is called a **request**. The thing that comes back is called a **response**.
5. A request says: which verb, which path, which version, then some labelled facts, then maybe some data.
6. A response says: which version, a three-digit number saying how it went, then some labelled facts, then maybe some data.

7. The labelled facts are called **headers**. Each is one line, of the form **Name: value**.
8. The data at the end is called the **body**. A request often has no body. A response usually has one.
9. So to understand HTTPS you must first be able to read HTTP. That is what this section does.

■ 32.1.2 PLAIN – a picture in your head

1. Think of posting a form to a large government office.
2. On the envelope you write the department and the reference number. That is the request line: the verb and the path.
3. Inside, the first page is a cover sheet of small boxes. Your name, your language, the date, whether you want a reply by post. Those are the headers.
4. After the cover sheet come the actual pages you are submitting. That is the body.
5. The office writes back. Their first line is a stamp: APPROVED, REJECTED, MOVED TO ANOTHER OFFICE. That is the status code.
6. Under the stamp is their own cover sheet of boxes, then their pages. Where this comparison breaks: a real office keeps a file on you. HTTP keeps nothing. The server may keep a file, but the protocol itself has no memory at all, and that single fact is the reason cookies had to be invented. Also, a government office handles one form at a time. A modern browser fires off dozens of these exchanges at once over a single connection.

■ 32.1.3 PLAIN – a worked example

1. Here is a real request, typed by hand and sent to `github.com`. The command that sent it is shown further down.
2. Every line ends with two invisible characters, carriage return and line feed, written `\r\n`. The header block ends with one empty line.

```
GET /robots.txt HTTP/1.1\r\n
Host: github.com\r\n
User-Agent: kedbyte/1.0\r\n
Accept: */*\r\n
Connection: close\r\n
\r\n
```

3. Line 1 is the **request line**: verb GET, path `/robots.txt`, version HTTP/1.1.
4. Host says which site you want. One server may host a thousand sites on one address, so this line is compulsory in HTTP/1.1.
5. User-Agent says which program is asking. Accept says which formats you can read. Connection: close asks the server to hang up after replying.
6. The empty line is the end of the headers. Anything after it would be body. A GET has no body, so there is nothing after it.
7. Here is the real response that came back, trimmed only where marked.

```
HTTP/1.1 200 OK
date: Thu, 13 Aug 2026 02:28:27 GMT
content-type: text/plain
last-modified: Thu, 13 Aug 2026 02:17:47 GMT
etag: W/"6a7d294b-8e2"
vary: Accept-Encoding, Accept, X-Requested-With
server: github.com
x-frame-options: DENY
strict-transport-security: max-age=31536000;
    includeSubDomains; preload
accept-ranges: bytes
```

```

set-cookie: _octo=GH1.1.1717559084.1786588114;
  expires=Fri, 13 Aug 2027 02:28:34 GMT; domain=.github.com;
  path=/; secure; SameSite=Lax
content-length: 2274
x-github-request-id: 4003:3C7715:235C0C:2CA82C:6A7D2BCD
x-github-edge-region: iad

# If you would like to crawl GitHub contact us via ...
User-agent: bingbot
Disallow: /ekansa/Open-Context-Data

```

8. Line 1 is the **status line**: version HTTP/1.1, code 200, and the human word OK.
9. content-length: 2274 says the body is exactly 2274 bytes long. The reader knows when to stop reading.

■ 32.1.4 PLAIN – what is really happening inside

1. The browser has already opened a TCP connection, which is a two-way stream of bytes to the server. Chapter 30 covers how.
2. HTTP is only an agreement about what to write into that stream.
3. The client writes the request bytes and then waits.
4. The server reads until it sees the blank line, works out what is wanted, and writes the response bytes back.
5. The server must tell the client where the body ends. There are only two ways: give a Content-Length, or use Transfer-Encoding: chunked, where each piece is preceded by its own size in hexadecimal.
6. Nothing in HTTP is encrypted. Anyone who can read the TCP stream can read every byte shown above, including any password in the body.
7. HTTPS changes exactly one thing: those same bytes are handed to a TLS layer, which encrypts them before they reach TCP, and decrypts them at the far end.
8. The verb, the path, the headers, the cookies and the body are all inside the encryption. The IP address and the port number are not, because the network needs them to deliver anything at all.

■ 32.1.5 TECHNICAL – the engineer's version

1. HTTP/1.0 is RFC 1945, May 1996. HTTP/1.1 was first RFC 2068 in January 1997, then RFC 2616 in June 1999, then split into RFC 7230 to RFC 7235 in June 2014.
2. The current specification is the June 2022 trio: RFC 9110 HTTP Semantics, RFC 9111 HTTP Caching, RFC 9112 HTTP/1.1. Quote these, not RFC 2616.
3. A method is **safe** if it is not meant to change anything on the server. A method is **idempotent** if doing it five times leaves the same state as doing it once.

Method	Safe	Idempotent	Body
GET	yes	yes	no
HEAD	yes	yes	no
OPTIONS	yes	yes	rare
PUT	no	yes	yes
DELETE	no	yes	rare
POST	no	no	yes
PATCH	no	no	yes

4. GET fetches. HEAD fetches only the headers, so you can check size or freshness without downloading the body.
5. PUT replaces a resource at a known path with what you send. POST submits data to a handler and the server decides what to create.
6. PATCH applies a partial change and is defined in RFC 5789, March 2010. It is not idempotent in general because a patch may be relative.
7. OPTIONS asks what is allowed. Its main real use today is the CORS preflight, covered in section 32.11.
8. Status codes come in five families.

Family	Meaning	Example
1xx	Interim, keep waiting	101, 103
2xx	It worked	200, 201, 204
3xx	Go somewhere else	301, 304, 307
4xx	Your request is wrong	400, 403, 404
5xx	Our server is wrong	500, 502, 503

9. The codes actually worth memorizing: 200 OK, 201 Created, 204 No Content, 206 Partial Content, 301 Moved Permanently, 302 Found, 304 Not Modified, 307 Temporary Redirect, 308 Permanent Redirect, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 405 Method Not Allowed, 409 Conflict, 429 Too Many Requests, 500 Internal Server Error, 502 Bad Gateway, 503 Service Unavailable, 504 Gateway Timeout.
10. 401 means “you have not proved who you are”. 403 means “I know who you are and you still may not”. People mix these up constantly.
11. Request headers that matter: Host, User-Agent, Accept, Accept-Encoding, Authorization, Cookie, Referer (misspelled in the original 1996 specification and never fixed), If-None-Match, Range, Origin, Content-Type, Content-Length.
12. This is the exact command used to produce the real output above. It opens a TLS connection and then types HTTP into it by hand.

```
printf 'GET /robots.txt HTTP/1.1\r\nHost: github.com\r\n\
Connection: close\r\n\r\n' \
| openssl s_client -quiet -connect github.com:443 \
  -servername github.com
```

■ 32.1.6 WORDS – remember these

1. HTTP — the ask-and-answer language of the web — a stateless application protocol defined by RFC 9110 to RFC 9112, 2022.
2. Request line — the first line you send — method, request-target and HTTP-version separated by single spaces.
3. Header — one labelled fact per line — a case-insensitive field name, a colon, optional whitespace, and a field value.
4. Body — the actual data — the payload delimited by Content-Length or by chunked transfer encoding.
5. Safe method — does not change anything — a method with no intended state change on the origin server, so it may be cached or prefetched.

32.2 The three problems TLS solves

■ 32.2.1 PLAIN – in simple words

1. TLS stands for Transport Layer Security. It is the lock on HTTPS.
2. People think it solves one problem, secrecy. It actually solves three, and the third is the one that matters most.
3. **Confidentiality**: nobody in the middle can read what you send.
4. **Integrity**: nobody in the middle can change what you send without being caught.
5. **Authenticity**: you are really talking to the machine that owns the name you typed, and not to somebody pretending.
6. Confidentiality without authenticity is nearly worthless.
7. Here is why, in one line: if an attacker sits in the middle and you set up a perfectly encrypted tunnel to the attacker, your secrets are encrypted all the way into the attacker's hands.
8. So the first job of TLS is not encryption. It is proving who the other end is. Encryption only becomes useful after that proof.

■ 32.2.2 PLAIN – a picture in your head

1. Imagine sending a valuable parcel to a friend in another city.
2. Confidentiality is the sealed opaque box. Nobody along the way can see what is inside.
3. Integrity is the tamper-evident seal. If somebody opens the box and swaps the contents, the broken seal shows it.
4. Authenticity is checking the identity of the person who signs for it at the other end.
5. Now picture a courier who quietly redirects your parcel to his own flat, signs for it himself, opens it, and repacks it in an identical box with a fresh seal before sending it on.
6. The box was sealed. The seal was intact on arrival. And you were robbed anyway, because you never checked who signed for it.
7. That is exactly a man-in-the-middle attack, and it is why authenticity comes first.

Where this comparison breaks: a courier must physically hold your parcel. On a network an attacker only needs to sit on any link along a path that may be thousands of kilometres long. The reader's own traceroute crossed at least three organizations before Pune. Any one of them is a place a box could be opened.

■ 32.2.3 PLAIN – a worked example

1. Take three concrete attacks, one per property.
2. **Attack on confidentiality: passive sniffing.** The reader is on public cafe wireless. An attacker runs a laptop in monitor mode and records every frame in the air. Over plain HTTP the attacker reads the session cookie and logs in as the reader.
3. What defeats it: encryption. The recorded bytes are meaningless without the key.
4. **Attack on integrity: content injection.** An ISP or a hotel network rewrites a plain HTTP page as it passes, adding an advertising script, or swapping a download link for a modified installer. This has been done commercially and by governments.
5. What defeats it: a message authentication tag on every record, so any changed byte makes the receiver throw the whole record away.
6. **Attack on authenticity: impersonation.** An attacker answers the DNS query for `github.com` with his own address, or announces a false route, and your browser connects to his server. He offers you a perfectly good TLS connection to himself.

7. Now combine them. Encryption alone stops attack 2 from being read but not attack 6 from happening. Authentication alone stops attack 6 but leaves attack 2 wide open. You need all three, and TLS gives all three in one protocol.

■ 32.2.4 PLAIN – what is really happening inside

1. TLS sits between the application and TCP. HTTP hands it bytes; it hands TCP encrypted records; TCP hands the network packets.
2. During the handshake the two sides agree on which algorithms to use, prove identity, and jointly create secret keys.
3. Identity is proved by a certificate plus a signature. The server sends a certificate that says “this public key belongs to github.com”, and then signs a value derived from the live conversation with the matching private key.
4. That second step is what stops replay. Anybody can copy a certificate. Only the true owner can sign fresh data with the private key inside it.
5. Once identity is settled, the two sides derive symmetric keys and switch to fast bulk encryption for everything after.

■ 32.2.5 TECHNICAL – the engineer’s version

1. TLS provides confidentiality and integrity through an AEAD cipher, which means Authenticated Encryption with Associated Data, and authenticity through X.509 certificate-based server authentication.
2. In TLS 1.3, defined in RFC 8446 of August 2018, all cipher suites are AEAD. Non-authenticated modes were removed entirely.
3. Server authentication is mandatory in practice on the web. Client authentication with a client certificate is optional and rare outside enterprises and machine-to-machine APIs.
4. Anonymous key exchange suites exist in the specifications and are disabled everywhere. They give encryption with no authentication, which is the useless case described above.
5. A property TLS does not provide: hiding who you are talking to. The destination IP address is in the clear, and until Encrypted Client Hello the server name was in the clear too.
6. Encrypted Client Hello was published as RFC 9849 in March 2026. It encrypts the server name and the protocol list in the first message. Deployment is uneven; treat it as new rather than universal.

■ 32.2.6 WORDS – remember these

1. Confidentiality — nobody can read it — an attacker with the full ciphertext learns nothing about the plaintext beyond its length.
2. Integrity — nobody can change it undetected — any modification is rejected by an authentication tag check.
3. Authenticity — you are talking to the right machine — the peer proved possession of the private key matching a certified public key.
4. Man in the middle — someone sitting between you and the server — an active on-path attacker who terminates and re-originates the connection.

32.3 The crypto building blocks, taught gently

■ 32.3.1 PLAIN – in simple words

1. TLS is built from five simple ideas. None of them is hard on its own.
2. **Symmetric encryption:** one shared secret scrambles and unscrambles. Fast. Both sides need the same key.

3. **Hashing:** turn any amount of data into a short fixed fingerprint. Easy forwards, impossible backwards.
4. **Message authentication code:** a fingerprint that only somebody with the secret can produce, so it proves the data was not changed by an outsider.
5. **Asymmetric encryption:** two matching keys. One is public, one is private. What one does, only the other can undo.
6. **Key exchange:** a way for two strangers to end up holding the same secret number, even though everything they said was heard by everyone.
7. TLS uses asymmetric maths only at the start, to prove identity and agree a key, because it is slow. Then it uses symmetric encryption for all the real traffic, because it is fast.

■ 32.3.2 PLAIN – a picture in your head

1. For Diffie-Hellman key exchange, use paint.
2. You and a stranger stand at opposite ends of a crowded room. Everyone can see what you carry across.
3. First you both agree, out loud, on a common colour. Say yellow. Everybody hears this. That is fine.
4. You privately pick a secret colour, red. You keep the tin hidden. The stranger privately picks blue and hides it.
5. You mix your red into the yellow and get orange. He mixes his blue into the yellow and gets green.
6. You swap. You hand him orange in public. He hands you green in public. The crowd sees orange and green.
7. Now you add your hidden red to the green you received. He adds his hidden blue to the orange he received.
8. You both hold the same colour: yellow plus red plus blue. The crowd, who only saw yellow, orange and green, cannot make it.
9. The trick is that mixing paint is easy and unmixing it is very hard.

Where this comparison breaks: unmixing paint is only messy, not impossible. The real system uses modular exponentiation, and reversing it means solving the discrete logarithm problem, which nobody can do quickly. Also, the shared result is a number, not a key; it is passed through a key derivation function first. And on its own it tells you nothing about who the stranger is, which is why certificates must be bolted on.

■ 32.3.3 PLAIN – a worked example

1. Let us do the real arithmetic with numbers small enough to check by hand.
2. Public, agreed out loud: a prime $p = 23$ and a base $g = 5$. Everyone knows both.
3. Alice secretly picks $a = 6$. Bob secretly picks $b = 15$.
4. Alice computes $A = g^a \bmod p$, that is $5^6 \bmod 23$. Now $5^6 = 15625$, and $15625 = 679 * 23 + 8$, so $A = 8$.
5. Bob computes $B = g^b \bmod p = 5^{15} \bmod 23 = 19$.
6. They send 8 and 19 across the room. An eavesdropper now knows $p = 23$, $g = 5$, $A = 8$, $B = 19$.
7. Alice computes $B^a \bmod p = 19^6 \bmod 23 = 2$.
8. Bob computes $A^b \bmod p = 8^{15} \bmod 23 = 2$.
9. Both hold 2, and it was never sent. This is real output from python3:

```
p = 23  g = 5
Alice secret a = 6  -> A = g^a mod p = 8
Bob   secret b = 15 -> B = g^b mod p = 19
```

```
Alice computes  $B^a \bmod p = 2$ 
Bob   computes  $A^b \bmod p = 2$ 
```

10. Why it works: $B^a = (g^b)^a = g^{(ba)}$ and $A^b = (g^a)^b = g^{(ab)}$, and ab equals ba , so both are $g^{(ab)} \bmod p$.
11. Why the eavesdropper is stuck: to get a from $A = 8$ he must find which power of 5 gives 8 modulo 23. With $p = 23$ he can just try all of them. Here are the powers of 5 modulo 23, in order:

```
5, 2, 10, 4, 20, 8, 17, 16, 11, 9, 22, 18, 21, 13, 19, 3,
15, 6, 7, 12, 14, 1
```

12. The sixth entry is 8, so $a = 6$. Trivial at this size.
13. In real use p is 2048 or 3072 bits, so that list has more entries than there are atoms in the observable universe, and there is no known shortcut.
14. Now hashing, with real openssl output. Change one letter and the whole fingerprint changes:

```
$ echo -n "hello" | openssl dgst -sha256
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362
938b9824

$ echo -n "hellp" | openssl dgst -sha256
fdd7585e08c4e2afd71dcabdb4636c89d557a3f42db9e2040c8bbd17
08aa4ce7
```

15. Each output is 64 hexadecimal characters, which is 256 bits, which is why it is called SHA-256. Both were wrapped over two lines to fit this page.

■ 32.3.4 PLAIN – what is really happening inside

1. **Symmetric encryption.** AES takes a key of 128 or 256 bits and a block of 16 bytes, and shuffles the block using the key through 10 to 14 rounds of substitution and mixing. The same key run in reverse undoes it.
2. Modern TLS uses AES-GCM, which encrypts and also produces a 16-byte authentication tag over the ciphertext and the record header. One primitive, both properties.
3. Here is a real AES-256-GCM run. Note the tag, and note what happens when one single bit of the ciphertext is flipped:

```
plaintext = meet me at six (14 bytes)
ciphertext = aaacf7bb4bf766de898f5ea22006
tag        = 1cee6a97e098c053433578dfe18e3d22
flip one bit -> decrypt raises InvalidTag
honest decrypt -> meet me at six
```

4. **Hashing.** SHA-256 chews the input in 64-byte blocks, keeping eight 32-bit working values that get stirred by each block. The final eight values, joined, are the digest. There is no key and no way back.
5. The property we rely on is **collision resistance**: nobody can find two different inputs with the same digest. When that broke for MD5 in 2004 and for SHA-1 in 2017, both had to be removed from certificates.
6. **Asymmetric.** RSA works with a public pair (n, e) and a private exponent d . Encrypting is $c = m^e \bmod n$. Decrypting is $m = c^d \bmod n$. Signing is the same operation with the private key, and verifying uses the public key.

7. **Signatures in practice.** You never sign the message. You hash it, then sign the hash. That is why every certificate says something like `sha256WithRSAEncryption`: hash with SHA-256, sign with RSA.
8. Here is real signing and verifying, with a genuine 2048-bit RSA key made for this chapter:

```
$ openssl dgst -sha256 -sign ca.key -out msg.sig msg.txt
$ openssl dgst -sha256 -verify ca.pub -signature msg.sig \
  msg.txt
Verified OK

$ openssl dgst -sha256 -verify ca.pub -signature msg.sig \
  bad.txt
rsa routines:ossl_rsa_verify:bad signature
Verification failure
```

■ 32.3.5 TECHNICAL – the engineer’s version

1. AES is FIPS 197, published November 2001, from the Rijndael cipher by Joan Daemen and Vincent Rijmen. Block size 128 bits; key sizes 128, 192 and 256 bits; 10, 12 and 14 rounds.
2. HMAC is RFC 2104, February 1997. It is defined as $H((K \text{ xor } \text{opad}) \parallel H((K \text{ xor } \text{ipad}) \parallel M))$, which nests two hashes so that length-extension tricks do not work.
3. RSA was described by Ron Rivest, Adi Shamir and Leonard Adleman in 1977 and published in February 1978. Diffie-Hellman came first, in the November 1976 paper “New Directions in Cryptography” by Whitfield Diffie and Martin Hellman.
4. Elliptic curve cryptography was proposed independently by Neal Koblitz and Victor Miller in 1985. Curve25519 was published by Daniel J. Bernstein in 2006 and is specified for use in TLS by RFC 7748, January 2016.
5. Elliptic curves replaced RSA for new deployments because they give the same security with far smaller keys and far cheaper operations. Here are real key sizes measured on this machine:

Key	Private PEM	Public DER
RSA 2048	1708 bytes	294 bytes
RSA 3072	2488 bytes	422 bytes
P-256	227 bytes	91 bytes
P-384	288 bytes	120 bytes

6. **Forward secrecy** means that recording today’s traffic and stealing the server’s private key tomorrow does not let the attacker decrypt what was recorded.
7. It works because the actual session key comes from an **ephemeral** Diffie-Hellman exchange, with a key pair generated for that one connection and then thrown away. The long-term private key only signs; it never encrypts the session key.
8. Old RSA key transport, where the client encrypted the premaster secret to the server’s RSA public key, had no forward secrecy. That is precisely why TLS 1.3 removed it. Every TLS 1.3 handshake is forward secret.
9. You can see the ephemeral key in real output. This line from a live handshake to `github.com` shows an ephemeral X25519 key was used:

```
Server Temp Key: X25519, 253 bits
New, TLSv1.2, Cipher is ECDHE-RSA-AES256-GCM-SHA384
```

10. The E at the end of ECDHE is the important letter. It stands for ephemeral. ECDH without the E is static and not forward secret.

11. Chapter 44 goes much further into cryptography, including post-quantum key exchange, which is now being deployed in TLS to protect against future quantum computers recording traffic today.

■ 32.3.6 WORDS – remember these

1. Symmetric encryption — one shared key both ways — a block or stream cipher such as AES-256-GCM, keyed identically at both ends.
2. Hash — a short fingerprint of any data — a one-way function such as SHA-256 producing a fixed 256-bit digest.
3. Public key — the half you publish — the key used to verify signatures and, in RSA, to encrypt to the holder.
4. Private key — the half you never share — the key used to sign and to decrypt, and the only thing that proves identity.
5. Diffie-Hellman — how strangers agree a secret in public — a key agreement using modular exponentiation or elliptic curve point multiplication.
6. Ephemeral — thrown away after one use — a per-connection key pair, giving forward secrecy, marked by the E in ECDHE.

32.4 The TLS handshake, in full

■ 32.4.1 PLAIN – in simple words

1. Before any web page byte moves, the two machines have a short conversation to set up the lock. That conversation is the **handshake**.
2. It does four jobs: agree the version, agree the algorithms, prove the server's identity, and create the shared keys.
3. In the old design this took two full round trips before you could send your first HTTP request.
4. A round trip is one message out and one message back. Over a long path that costs real time, and the reader's own path went from India to Microsoft edge sites in Delhi, Mumbai and Pune.
5. The current design, TLS 1.3, does the same four jobs in one round trip.
6. It manages this by guessing. The client sends its key-exchange material in the very first message, assuming the server will accept it. Almost always the guess is right.
7. There is also a mode with zero round trips, for repeat visits. It is faster and it has a real weakness, described later in this section.

■ 32.4.2 PLAIN – a picture in your head

1. Think of two people meeting to do business, who have never met before.
2. TLS 1.2 is the polite, slow version. "Hello, I speak English, French and Hindi." "Hello, let us use English." "Here is my identity card." "Here is a number I have signed to prove the card is mine." "Now here is my half of the secret." "Here is my half." "Agreed, switching to code." "Agreed."
3. Two complete there-and-back exchanges before a single word of business.
4. TLS 1.3 is the version where you have done this a hundred times before and you know how it will go. "Hello, I speak English, and here is my half of the secret already, assuming you agree." "Agreed, here is mine, here is my card, here is my signature, and I am already speaking in code."
5. One there-and-back. The business starts on your second message. Where this comparison breaks: people can recognize a face. Machines cannot, so the identity card must be checked mathematically every single time. And the "half of the secret" is not half of anything; it is a full public value that is useless without the matching private one.

■ 32.4.3 PLAIN – a worked example

1. Here is a real TLS 1.2 handshake, captured live against `github.com`. The `>>>` lines were sent by us, the `<<<` lines came back.

```
>>> TLS 1.2, Handshake, ClientHello
<<< TLS 1.2, Handshake, ServerHello
<<< TLS 1.2, Handshake, Certificate
<<< TLS 1.2, Handshake, ServerKeyExchange
<<< TLS 1.2, Handshake, ServerHelloDone
>>> TLS 1.2, Handshake, ClientKeyExchange
>>> TLS 1.2, ChangeCipherSpec
>>> TLS 1.2, Handshake, Finished
<<< TLS 1.2, Handshake, NewSessionTicket
<<< TLS 1.2, ChangeCipherSpec
<<< TLS 1.2, Handshake, Finished
```

2. Count the direction changes: out, in, out, in. That is two round trips before application data.
3. Now the same server, forced to TLS 1.3:

```
>>> TLS 1.3, Handshake, ClientHello
<<< TLS 1.3, Handshake, ServerHello
<<< TLS 1.3, Handshake, EncryptedExtensions
<<< TLS 1.3, Handshake, Certificate
<<< TLS 1.3, Handshake, CertificateVerify
<<< TLS 1.3, Handshake, Finished
>>> TLS 1.3, ChangeCipherSpec
>>> TLS 1.3, Handshake, Finished
```

4. Out, in, out. One round trip, and the client can attach its HTTP request to that last outbound flight.
5. Notice what disappeared: `ServerKeyExchange`, `ServerHelloDone` and `ClientKeyExchange` are gone. Their work moved into the two hello messages.
6. Notice what appeared: `EncryptedExtensions` and `CertificateVerify`. And notice that from `EncryptedExtensions` onwards, everything is already encrypted, including the server's certificate.
7. The single `ChangeCipherSpec` still there in TLS 1.3 is a fake. It does nothing. It is sent only so that old network middleboxes, which expected to see one, do not break the connection.

■ 32.4.4 PLAIN – what is really happening inside

1. In `ClientHello` the client sends: the highest version it supports, 32 random bytes, a session identifier, a list of cipher suites it will accept, and a list of extensions.
2. Two extensions matter enormously. `server_name` carries the hostname, so one address can serve many sites. `supported_versions` is how TLS 1.3 actually announces itself, because the old version field had to be frozen at 1.2 for compatibility.
3. In TLS 1.3 the client also sends `key_share`: an actual ephemeral public key, usually X25519, guessed in advance.
4. In `ServerHello` the server picks one cipher suite, sends its own 32 random bytes, and in TLS 1.3 sends its own `key_share`.
5. At that instant both sides can compute the shared secret, so everything after the `ServerHello` can be encrypted. That is the core trick of TLS 1.3.
6. `Certificate` carries the server's certificate and the intermediates needed to build a chain. In TLS 1.3 it is encrypted, so a passive observer no longer learns which certificate was served.

7. `CertificateVerify` is a signature, made with the certificate's private key, over a hash of the entire handshake so far. This is the step that proves the server actually holds the private key and is not just replaying a copied certificate.
8. `Finished` is a MAC over the whole transcript, sent by each side. If any earlier message was altered, the two transcripts differ and the check fails. This is what protects the cipher negotiation from downgrade.
9. In TLS 1.2 the equivalent of step 7 was `ServerKeyExchange`, which carried the server's ephemeral DH parameters plus a signature over them. The client then sent `ClientKeyExchange` with its own half.

■ 32.4.5 TECHNICAL - the engineer's version

1. The version history, with real dates:

Version	Published	Reference
SSL 2.0	Feb 1995	Netscape, no RFC
SSL 3.0	1996	RFC 6101 (2011)
TLS 1.0	Jan 1999	RFC 2246
TLS 1.1	Apr 2006	RFC 4346
TLS 1.2	Aug 2008	RFC 5246
TLS 1.3	Aug 2018	RFC 8446

2. SSL 1.0 was never released publicly; it had known flaws. SSL 2.0 shipped in Netscape Navigator 1.1 in 1995. SSL 3.0 was a full redesign by Paul Kocher with Phil Karlton and Alan Freier at Netscape. Taher Elgamal, then Netscape's chief scientist, is widely credited as a driving force behind SSL.
3. TLS 1.3 removed, deliberately: RSA key transport, static Diffie-Hellman, custom DH groups, renegotiation, compression, CBC mode ciphers, all non-AEAD ciphers, RC4, DSA, MD5 and SHA-1 signatures in the handshake, and the change cipher spec protocol as a real message.
4. TLS 1.3 defines only five cipher suites, and OpenSSL 3.0.13 on this machine offers three of them:

TLS_AES_256_GCM_SHA384	Enc=AESGCM(256)	Mac=AEAD
TLS_CHACHA20_POLY1305_SHA256	Enc=CHACHA20	Mac=AEAD
TLS_AES_128_GCM_SHA256	Enc=AESGCM(128)	Mac=AEAD

5. Reading a TLS 1.2 cipher suite string. Take `ECDHE-RSA-AES256-GCM-SHA384`, which is what `github.com` actually negotiated when forced to TLS 1.2 in the test above.
6. ECDHE is the key exchange: elliptic curve Diffie-Hellman, ephemeral. RSA is the authentication: the certificate holds an RSA key used for the signature. AES256 is the bulk cipher and its key size. GCM is the mode, giving AEAD. SHA384 is the hash used by the key derivation function and, in older suites, by the MAC.
7. TLS 1.3 suite names dropped the key exchange and authentication parts entirely, because those are now negotiated by separate extensions. That is why `TLS_AES_128_GCM_SHA256` looks so short.
8. Here are the two handshakes side by side.

TLS 1.2 - two round trips before data

Client	Server
-- ClientHello ----->	
<----- ServerHello --	
<----- Certificate --	
<----- ServerKeyExchange -	
<----- ServerHelloDone -	

```
|-- ClientKeyExchange ----->|
|-- ChangeCipherSpec ----->|
|-- Finished ----->|
|<----- ChangeCipherSpec-|
|<----- Finished-|
|-- HTTP GET ----->|
```

TLS 1.3 - one round trip before data

Client	Server
-- ClientHello + key_share ----->	
<----- ServerHello + key_share -	
<----- EncryptedExtensions -	encrypted
<----- Certificate -	encrypted
<----- CertificateVerify -	encrypted
<----- Finished --	encrypted
-- Finished + HTTP GET ----->	encrypted

9. **0-RTT**, also called early data. On a repeat visit the client has a pre-shared key from a `NewSessionTicket` sent during the previous connection. It can encrypt application data with a key derived from that ticket and put it in the very first flight, alongside the `ClientHello`.
10. The caveat, stated plainly in RFC 8446 section 8 and appendix E.5: 0-RTT data has **no replay protection**. An attacker who records the early-data flight can send it again later, and the server may accept it a second time.
11. Also, 0-RTT data is not forward secret, because it is protected by keys derived from the earlier ticket rather than by a fresh exchange.
12. The practical rule: only allow 0-RTT for safe, idempotent requests. Never for a `POST` that moves money. Cloudflare and other providers restrict early data to `GET` requests without query parameters by default.

■ 32.4.6 WORDS - remember these

1. Handshake — the setup conversation — the sub-protocol that negotiates parameters, authenticates the peer and establishes keys.
2. `ClientHello` — the opening message — carries versions, random bytes, cipher suites, and extensions including `server_name` and `key_share`.
3. Cipher suite — the chosen set of algorithms — key exchange, authentication, bulk cipher and hash, named as one string.
4. `key_share` — the client's guess at the key exchange — an ephemeral public key sent in the first message so TLS 1.3 needs only one round trip.
5. `CertificateVerify` — proof the server owns the key — a signature over the handshake transcript using the certificate's private key.
6. Finished — the tamper check on the whole conversation — a MAC over the full transcript that defeats downgrade attacks.

32.5 Certificates: what is actually in one

■ 32.5.1 PLAIN - in simple words

1. A certificate is a small signed document that says: "this public key belongs to this name".
2. It is not secret. It is sent to everyone who connects. There is nothing private inside it.
3. It contains a name, a public key, two dates, the name of whoever vouched for it, and that voucher's signature.

4. The signature is the whole point. Without it the document is just a claim anybody could type.
5. Your browser does not trust the certificate because of what it says. It trusts it because of who signed it, and because it can check that signature mathematically.
6. The format is called X.509. It is old, it is complicated, and every browser and server on earth uses it.

■ 32.5.2 PLAIN – a picture in your head

1. Think of a passport.
2. The photograph is the public key: the part that identifies you and can be shown to anyone.
3. Your face is the private key: the thing that must match the photograph, that you carry with you, and that cannot be copied out of the book.
4. The name page is the subject: who this passport is for.
5. The issuing authority printed on the cover is the issuer: who vouched.
6. The expiry date is the validity period.
7. The holographic security printing is the signature: hard to forge, easy to check, and it covers the whole book so nothing can be swapped. Where this comparison breaks: a passport is checked by a human who can use judgement. A certificate is checked by software that follows fixed rules and has none. Also, a passport proves who you are. A certificate only proves you control a name. It says nothing about whether you are honest, and section 32.7 makes a great deal of that difference.

■ 32.5.3 PLAIN – a worked example

1. Here is a real certificate, created with `openssl` for this chapter. It is for a home router, and it is used again in section 32.8.
2. It was made with `openssl req -x509` for the authority, then `openssl req` for the request, then `openssl x509 -req -CA ca.crt` to sign. An extensions file supplied the modern half of the certificate:

```
subjectAltName=DNS:router.home,IP:192.168.0.1
basicConstraints=CA:FALSE
keyUsage=digitalSignature,keyEncipherment
extendedKeyUsage=serverAuth
```

3. Now read it back. This is genuine `openssl x509 -text` output, with the long key and signature byte columns removed to fit the page:

```
Certificate:
Data:
  Version: 3 (0x2)
  Serial Number: 6EBE14595FF0AED0A54EF7E6925D84514B5421DC
  Signature Algorithm: sha256WithRSAEncryption
  Issuer: C = IN, O = KedByte Home CA,
         CN = KedByte Home Root CA
  Validity
    Not Before: Aug 13 02:27:52 2026 GMT
    Not After : Sep 14 02:27:52 2027 GMT
  Subject: CN = router.home
  Subject Public Key Info:
    Public Key Algorithm: rsaEncryption
    Public-Key: (2048 bit)
    Exponent: 65537 (0x10001)
  X509v3 extensions:
    X509v3 Subject Alternative Name:
      DNS:router.home, IP Address:192.168.0.1
```

```

X509v3 Basic Constraints:
  CA:FALSE
X509v3 Key Usage:
  Digital Signature, Key Encipherment
X509v3 Extended Key Usage:
  TLS Web Server Authentication
X509v3 Subject Key Identifier:
  A4:68:C7:64:BF:C2:49:91:9E:73:CE:44:3C:86:19:70:
  52:25:4B:3B
Signature Algorithm: sha256WithRSAEncryption

```

4. Every one of those lines matters. Read them in order and the whole design becomes obvious.
5. Now the verification, which is the part people never actually try:

```

$ openssl verify router.crt
CN = router.home
error 20 at 0 depth lookup: unable to get local issuer
certificate
error router.crt: verification failed

$ openssl verify -CAfile ca.crt router.crt
router.crt: OK

```

6. The same certificate failed and then passed. Nothing about the certificate changed. Only the set of trusted issuers changed. That single fact is the entire subject of section 32.6.

■ 32.5.4 PLAIN – what is really happening inside

1. A certificate is a structure written in a binary encoding called DER, then often wrapped in base 64 text between BEGIN CERTIFICATE and END CERTIFICATE lines, which is called PEM format.
2. The structure has three top-level parts: the body, the algorithm identifier, and the signature bits.
3. Signing means: take the exact bytes of the body, hash them with SHA-256, and then apply the issuer's private key operation to that hash.
4. Verifying means: take the same body bytes, hash them the same way, and use the issuer's public key to check that the signature corresponds to that hash.
5. Because the signature covers the body bytes exactly, changing any single character anywhere in the certificate breaks it. You cannot edit the expiry date. You cannot add a hostname.
6. The **subject** field is the old way of naming the site, using a Common Name. It is dead. Since RFC 9525 of November 2023, the Common Name must not be used for identity at all.
7. The living way is the **Subject Alternative Name** extension, usually shortened to SAN. It is a list, so one certificate can cover example.com, www.example.com and *.api.example.com at once.
8. **basicConstraints: CA:FALSE** says this certificate may not sign other certificates. This is what stops a normal website certificate from being used to mint fake ones. It is checked, and it is critical.

■ 32.5.5 TECHNICAL – the engineer's version

1. X.509 began in 1988 as part of the CCITT X.500 directory series. The profile used on the internet is RFC 5280, May 2008, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile".
2. Field by field, the body of a certificate contains: version, serial number, signature algorithm, issuer distinguished name, validity notBefore and notAfter, subject distinguished name, subject public key info, and the extensions.
3. Extensions that matter in practice: subjectAltName, basicConstraints, keyUsage, extended-KeyUsage, authorityKeyIdentifier, subjectKeyIdentifier, crlDistributionPoints, author-

ityInfoAccess which carries the OCSP responder and issuer URLs, certificatePolicies, and the signed certificate timestamp list used by Certificate Transparency.

4. Maximum lifetime is set by the CA/Browser Forum Baseline Requirements, not by any RFC. It was 398 days for certificates issued on or after 1 September 2020.
5. Ballot SC-081v3, passed in April 2025, sets a reducing schedule:

From	Max lifetime
until 14 Mar 2026	398 days
15 Mar 2026	200 days
15 Mar 2027	100 days
15 Mar 2029	47 days

6. Domain validation reuse periods shrink on the same schedule, down to 10 days from March 2029. The direction of travel is short lifetimes and full automation.
7. Here is a real, unmodified root certificate from this machine's trust store, which you can check yourself on any Linux box:

```
$ openssl x509 -in /etc/ssl/certs/ISRG_Root_X1.pem \
  -noout -subject -issuer -dates
subject=C = US, O = Internet Security Research Group,
        CN = ISRG Root X1
issuer=C = US, O = Internet Security Research Group,
        CN = ISRG Root X1
notBefore=Jun  4 11:04:38 2015 GMT
notAfter=Jun  4 11:04:38 2035 GMT
```

8. Subject equals issuer. That is the definition of a self-signed certificate, and every root in every trust store on earth is one.
9. The honest version: the sandbox used to write this chapter sends all outbound traffic through a company proxy that re-issues certificates. So the live chain captured here shows that proxy as the issuer, not a public authority. Section 32.9 uses that as the worked example, because it is a perfect real specimen of interception rather than a flaw in the method.

■ 32.5.6 WORDS – remember these

1. Certificate — a signed statement binding a name to a key — an X.509 version 3 structure profiled by RFC 5280.
2. Subject — who the certificate is about — a distinguished name, now effectively decorative for TLS identity.
3. Issuer — who signed it — the distinguished name of the certificate authority whose private key produced the signature.
4. SAN — the real list of names this certificate covers — the Subject Alternative Name extension, the only source of identity since RFC 9525.
5. Validity — the two dates it works between — notBefore and notAfter, compared against the verifier's own clock.
6. basicConstraints — whether this may sign other certificates — a critical extension carrying CA:TRUE or CA:FALSE and an optional path length.

32.6 The chain of trust

■ 32.6.1 PLAIN – in simple words

1. Your browser does not know `github.com`. It has never heard of it.
2. What your browser does know is a list of about 150 organizations it was told to trust, which came with your operating system or your browser.
3. Those are the **root** certificate authorities.
4. A root does not sign website certificates directly. It signs a small number of **intermediate** certificates.
5. Intermediates sign the actual website certificates, called **leaf** certificates.
6. So the shape is: root signs intermediate, intermediate signs leaf, leaf belongs to the website.
7. When you connect, the server sends you the leaf and usually the intermediate. Your browser already has the root.
8. The browser then checks each signature in turn until it reaches something in its own list. If it gets there, the chain is trusted. If it does not, you get a warning.

■ 32.6.2 PLAIN – a picture in your head

1. Think of a university degree certificate.
2. You have never met the person holding it. You cannot verify their study.
3. But the degree is signed by a department head. The department head was appointed by the university. The university is recognized by a national body you already accept.
4. You do not check the student. You check the chain of signatures upward until you reach a body you already decided to accept, and then you stop.
5. The national body is the root. The university is the intermediate. The degree is the leaf.
6. Now the important part: the national body keeps its own official seal in a locked vault and almost never takes it out. It delegates day-to-day signing to the universities. Where this comparison breaks: you can choose not to accept a degree. Software cannot exercise judgement, so the list of accepted bodies must be exactly right. Also, a national body has one country. A root certificate authority is trusted for every name on the internet, which is a very strange amount of power for a private company to hold.

■ 32.6.3 PLAIN – a worked example

1. Here is a real chain, captured live in this sandbox. Read the `s:` line as “subject” and the `i:` line as “issuer”.

```
Certificate chain
0 s:CN = github.com
  i:0 = Anthropic,
    CN = Egress Gateway SDS Issuing CA (production)
  a:PKKEY: rsaEncryption, 2048 (bit); sigalg: RSA-SHA256
  v:NotBefore: Aug  6 23:49:45 2026 GMT;
    NotAfter: Sep  5 23:50:45 2026 GMT
1 s:0 = Anthropic,
  CN = Egress Gateway SDS Issuing CA (production)
  i:0 = Anthropic,
    CN = sandbox-egress-gateway-production Egress Gateway CA
2 s:0 = Anthropic,
  CN = sandbox-egress-gateway-production Egress Gateway CA
  i:0 = Anthropic,
    CN = sandbox-egress-gateway-production Egress Gateway CA
```

2. Read it as a ladder. Certificate 0 is the leaf for `github.com`. Its issuer is the subject of certificate 1. Certificate 1's issuer is the subject of certificate 2. Certificate 2 is its own issuer, so it is a root.
3. That is exactly the shape you want: leaf, intermediate, root.
4. What is wrong with it is who those names are. That is section 32.9.
5. On the reader's own Mac at home, running the same command against `github.com`, positions 1 and 2 would carry the name of a public certificate authority instead, and position 2 would match a certificate already in the system trust store.
6. Here is the trust store on this machine, which is a Debian-family Linux:

```
$ ls /etc/ssl/certs/*.pem | wc -l
152
$ grep -c "BEGIN CERTIFICATE" \
  /etc/ssl/certs/ca-certificates.crt
152
```

7. And here are four real roots from it, with their real dates:

Root	Not before	Not after
GlobalSign Root CA	Sep 1998	Jan 2028
Baltimore CyberTrust	May 2000	May 2025
DigiCert Global Root	Nov 2006	Nov 2031
ISRG Root X1	Jun 2015	Jun 2035

■ 32.6.4 PLAIN – what is really happening inside

1. Why intermediates exist at all, in four reasons.
2. **Protection.** The root private key can be kept offline, in a hardware module in a safe, powered up a few times a year under camera. The intermediate key is online and doing daily work. If the online one is stolen, you replace the intermediate, not the root.
3. **Recovery.** You cannot fix a compromised root. It is baked into a billion devices and updating them takes years. You can fix a compromised intermediate in days.
4. **Constraint.** An intermediate can be limited by policy or, in some cases, by a name constraint, so it can only issue for certain domains.
5. Now path building. The browser has a leaf and possibly some extra certificates. It must find a route to a trusted root.
6. It looks at the leaf's issuer name and authority key identifier, and searches: first the certificates the server sent, then its own store, then any cached intermediates it has collected before.
7. This is a search, not a straight line. There can be several valid paths. A single intermediate is often cross-signed by two different roots so it can be validated by both old and new devices.
8. Once a candidate path is found, it validates every link: signature correct, dates valid, CA:TRUE present on every non-leaf, keyUsage including certificate signing, path length not exceeded, and the final certificate present in the trust store.
9. A very common production bug: the server forgets to send the intermediate. It works in a browser, because browsers cache intermediates from earlier visits, and fails in `curl` or in a Java service, which do not. Always test with a fresh tool, not with your browser.

■ 32.6.5 TECHNICAL – the engineer's version

1. Where the root store actually lives:

System	Location
macOS	System Roots keychain
Windows	Trusted Root cert store
Debian/Ubuntu	/etc/ssl/certs
Firefox	its own NSS store

2. On macOS, open Keychain Access and look at System Roots, or use `security find-certificate` and `security dump-trust-settings` from the terminal. The file behind it is `SystemRootCertificates.keychain` under `/System/Library/Keychains`.
3. On Windows, run `certlm.msc` for the machine store or `certmgr.msc` for the user store, and look under Trusted Root Certification Authorities. Windows can also fetch roots on demand from Microsoft Update.
4. On Debian and Ubuntu the source files live in `/usr/share/ca-certificates`, local additions go in `/usr/local/share/ca-certificates`, and `update-ca-certificates` rebuilds `/etc/ssl/certs/ca-certificates.crt`. On Red Hat family systems the directory is `/etc/pki/ca-trust/source/anchors` and the command is `update-ca-trust`.
5. Firefox has always shipped its own store, so a root added to the operating system does not automatically work in Firefox. There is an enterprise policy to make Firefox also read the system store.
6. Chrome historically used the operating system store. It began shipping its own Chrome Root Store in 2022, from Chrome 105 onwards, becoming the default on Windows and macOS later that year. Treat the exact version numbers as approximate; the direction is not in doubt.
7. Who decides what goes in: the four root programmes are Mozilla, Microsoft, Apple and Google. Each publishes its own policy. The CA/Browser Forum, founded in 2005, publishes the Baseline Requirements that the programmes reference.
8. To get thrown out takes much less. Real examples: DigiNotar removed in September 2011 after a breach, and bankrupt within the month. WoSign and StartCom distrusted through 2016 and 2017 for backdating certificates and misissuance. Symantec's entire authority business distrusted by Chrome in stages through 2018, with the business sold to DigiCert in 2017.
9. What a certificate authority actually verifies before issuing, in three levels.
10. **Domain Validation (DV)**. Proves only that the applicant controls the domain right now. Methods, from the Baseline Requirements: put a specific token at a specific path over HTTP, publish a specific DNS TXT record, respond to email at `admin@` or the address in WHOIS, or answer a TLS-ALPN challenge. Time taken: seconds, fully automated.
11. **Organisation Validation (OV)**. Adds a check that a legal entity of that name exists, usually against a government register and a phone directory. Time taken: hours to days.
12. **Extended Validation (EV)**. Adds documents, a lawyer's or accountant's confirmation, and a callback. Guidelines published by the CA/Browser Forum in 2007. Time taken: days to weeks.
13. How weak is this really? Domain Validation, which is what almost every site uses, proves control of DNS or of a web path. Anyone who can hijack DNS or BGP for a few minutes can pass it. It says nothing about identity.
14. The blunt summary: the modern web's certificate system proves control of a name. It stopped trying to prove who anybody is, because that turned out not to work.
15. Certificate Transparency, RFC 6962 of June 2013 and RFC 9162 of December 2021, is the real check today. Every publicly trusted certificate must be logged to public append-only logs, and Chrome refuses certificates without proof of logging. Domain owners can therefore see certificates issued for their names by anyone.

■ 32.6.6 WORDS – remember these

1. Leaf — the website's own certificate — the end-entity certificate with CA:FALSE and the site's names in its SAN list.
2. Intermediate — the middle link — a subordinate CA certificate signed by a root and used for day-to-day issuance.
3. Root — the anchor of trust — a self-signed CA certificate present in the verifier's trust store, kept offline where possible.
4. Trust store — your machine's list of accepted anchors — an operating system or application collection of root certificates.
5. Path building — finding a route to a trusted anchor — the search through supplied, stored and cached certificates for a valid chain.
6. Certificate Transparency — public logging of every certificate — append- only logs, mandatory for Chrome, defined in RFC 6962 and RFC 9162.

32.7 What the browser verifies, and what it does NOT

■ 32.7.1 PLAIN – in simple words

1. When the padlock appears, the browser has checked a short and very specific list.
2. It checked the signature chain reaches a root it trusts.
3. It checked today's date sits between the two dates in the certificate.
4. It checked the hostname you typed appears in the certificate's list of names.
5. It checked that no certificate in the chain claims permissions it should not have.
6. It checked revocation, badly, and often not at all.
7. That is the entire list. Now the other list.
8. It did **not** check that the site is honest.
9. It did **not** check that the company behind it is real, registered, or solvent.
10. It did **not** check that the content is safe, that the download is not malware, or that the form is not a phishing page.
11. It did **not** check that the name is the one you meant. A certificate for `paypal.com` is perfectly valid; it just is not PayPal.
12. The padlock has never meant "this site is safe". It means "the pipe to whoever owns this name is private". Those are completely different claims.

■ 32.7.2 PLAIN – a picture in your head

1. Think of a sealed pneumatic tube between your desk and an office somewhere.
2. The padlock is a guarantee about the tube. Nobody can read what goes through it. Nobody can slip a different note in. The tube really does end at the office with that name on the door.
3. It is not a guarantee about the office. The people in that office may be thieves. The name on the door may have been chosen last night to look like a bank.
4. A criminal can rent an office, put any name on the door, and order a perfectly good sealed tube from the tube company. The tube company only checks that the criminal really does control that door.
5. And that is exactly what happens: phishing sites almost all use HTTPS now, because certificates are free and automated.

Where this comparison breaks: a physical office costs money and can be raided. A domain costs a few dollars, is registered in minutes, and can be abandoned the moment it is reported. The economics are entirely different, which is why the "green padlock means safe" idea was retired.

■ 32.7.3 PLAIN – a worked example

1. Take a real user story. Someone receives a message about a parcel and opens a link.
2. The address bar shows a padlock. There is no warning. The certificate is valid, issued minutes earlier by a free automated authority.
3. Every browser check passes: chain to a trusted root, dates fine, hostname matches the SAN entry exactly, CA:FALSE, correct key usage.
4. The page is a copy of a courier company's site, asking for a card number.
5. TLS did its job perfectly. It delivered the card number, encrypted and untampered, straight to the criminal.
6. Now contrast that with what the browser actually would have blocked: an attacker sitting on the cafe wireless trying to intercept the real courier site. That attacker cannot get a certificate for the real name, so the browser stops him.
7. Two different threats. TLS solves one of them completely and the other one not at all.
8. Modern browsers do have a defence against the phishing case, but it is a separate system: Google Safe Browsing, Microsoft SmartScreen and similar reputation lists. That is content safety, not transport security, and it works on a blocklist rather than on mathematics.

■ 32.7.4 PLAIN – what is really happening inside

1. The hostname check is more subtle than people think.
2. The browser takes the name from the URL, not from DNS. If you typed `github.com`, the answer must contain `github.com`, whatever address the connection went to.
3. It compares against the SAN list only. The Common Name is ignored by modern clients.
4. Wildcards match one label only, and only at the leftmost position. `*.example.com` matches `api.example.com` but not `example.com` itself and not `a.b.example.com`.
5. The date check uses the client's own clock. A device with a badly wrong clock reports every certificate as expired or not yet valid. This is a very common cause of "all HTTPS is broken" on cheap devices and on machines whose battery died.

■ 32.7.5 TECHNICAL – the engineer's version

1. The full check list a conforming client performs, per RFC 5280 and RFC 9525:

Check	Enforced
Chain to trusted root	strictly
Signature on each link	strictly
notBefore / notAfter	strictly
Hostname against SAN	strictly
basicConstraints CA	strictly
keyUsage, extKeyUsage	strictly
Name constraints	strictly
Certificate Transparency	Chrome: yes
Revocation	weakly or not

2. **CRL**, Certificate Revocation List. The authority publishes a signed list of revoked serial numbers. Problem: the list grows without bound, downloading it is slow, and it may be hours or days out of date.
3. **OCSP**, Online Certificate Status Protocol, RFC 6960 of June 2013. The client asks the authority about one specific serial number. Problems: it adds a network round trip to a third party during

page load; it tells the authority which sites you visit, which is a privacy leak; and if the responder is unreachable, clients “soft-fail” and continue anyway.

4. Soft-fail is the fatal flaw. An attacker who can intercept your connection can also block your OCSP query, and then the check silently passes. A check an attacker can turn off is not a check.
5. **OCSP stapling**, the `status_request` extension from RFC 6066. The server fetches its own signed OCSP response periodically and staples it into the handshake. This fixes the privacy leak and the extra round trip. It does not fix soft-fail, because a server can simply not staple.
6. In 2024 and 2025 Let’s Encrypt withdrew from OCSP entirely for privacy and cost reasons. It announced the plan in December 2024, dropped OCSP URLs from certificates on 7 May 2025, and switched off its OCSP responders on 6 August 2025. It relies on CRLs and on short lifetimes instead.
7. **Short-lived certificates are the real direction of travel.** If a certificate only lives days, revocation barely matters, because expiry does the job.
8. Let’s Encrypt issued its first six-day certificate on 19 February 2025. Combined with the CA/Browser Forum schedule that takes the maximum to 47 days in March 2029, the industry answer to broken revocation is simply to make certificates expire faster than problems can spread.
9. Where experts disagree: some argue hard-fail revocation should have been mandated from the start and that soft-fail was a failure of nerve. Others argue hard-fail would have made the web fragile and dependent on authority uptime, and that short lifetimes are the better engineering answer. Both positions are held by serious people.

■ 32.7.6 WORDS – remember these

1. Padlock — the browser’s “connection is private” mark — an indicator that chain, dates, hostname and constraints validated; nothing more.
2. Hostname verification — the name you typed is in the certificate — a match against `subjectAltName` entries per RFC 9525.
3. Revocation — cancelling a certificate before expiry — publishing its serial as invalid via CRL or OCSP.
4. OCSP — asking the authority about one certificate — an online status query, RFC 6960, usually soft-failing and now being retired.
5. Stapling — the server carries its own status proof — the `status_request` TLS extension, removing the client’s third-party lookup.
6. Soft-fail — carrying on when the check does not answer — the default behaviour that makes revocation checking defeatable by an attacker.

32.8 Self-signed certificates and the reader’s own router

■ 32.8.1 PLAIN – in simple words

1. A **self-signed** certificate is one where the subject and the issuer are the same. The key vouches for itself.
2. Nobody else has checked anything. There is no chain. There is nothing to follow upward.
3. So the browser cannot answer the question “does this reach a root I trust”, and it stops and warns you.
4. The warning is not saying the encryption is weak. The encryption is exactly as strong as any other TLS connection.
5. The warning is saying: I cannot tell you who is at the other end.
6. That is the honest message. Everything else follows from it.

7. The reader's home router lives at 192.168.0.1. Opening it in a browser over HTTPS produces exactly this warning, on every home router ever made.

■ 32.8.2 PLAIN – a picture in your head

1. Somebody hands you a business card that says “I am the manager”.
2. The card was printed by that same person. There is no company logo, no reference you can call.
3. If you are standing inside your own house and the person handing it over is the box on your own shelf that you plugged in yourself, the card tells you nothing you did not already know, and you do not need it.
4. If you are on a street in another city and a stranger hands you the same self-printed card claiming to be your bank manager, the card is worthless and the situation is alarming.
5. Same card. Completely different meaning, because of where you are standing and what else you know.

Where this comparison breaks: you can see a person and a shelf. You cannot see a network path. The reason the router case is safe is not that it feels close, but that the path is provably short, which the next block explains.

■ 32.8.3 PLAIN – a worked example

1. Take the reader's real setup. The default gateway is 192.168.0.1. The Mac is on the same wireless link as that box.
2. The certificate the router presents is self-signed. Here is the shape of one, generated for this chapter:

```
Subject: CN = router.home
Issuer : CN = KedByte Home Root CA
SAN    : DNS:router.home, IP Address:192.168.0.1
```

3. Now ask the only question that matters: could anything be sitting between the Mac and the router, pretending to be the router?
4. To do that, an attacker must already be on the reader's own wireless network or inside the flat with a cable. They would need the wireless password, or physical access.
5. But if an attacker is already on the local network with the wireless password, they can attack the router directly. Accepting the certificate warning does not make that worse.
6. So the risk added by clicking through, in this one situation, is close to zero. The endpoint is known, the path is one hop, and the attacker who could exploit it already has better options.
7. Now change one thing. The reader is on cafe wireless, types a bank address, and the same warning appears.
8. Here the path is long, the endpoint is unknown, and the warning is consistent with exactly one thing: somebody on the path is terminating the connection and cannot produce a real certificate.
9. That is the definition of an active interception attempt in progress. Clicking through hands them everything.
10. The asymmetry in one line: on the router you already know the endpoint, so the certificate adds nothing. On a public site the certificate is the only thing that identifies the endpoint, so ignoring it removes everything.

■ 32.8.4 PLAIN – what is really happening inside

1. Clicking “proceed anyway” does not weaken the encryption. The session is still AES-GCM with an ephemeral key exchange.

2. What it removes is the authenticity property from section 32.2. You keep two of the three guarantees and throw away the one that made them useful against an active attacker.
3. Doing it properly instead has two options, and both remove the warning without removing the guarantee.
4. **Option one: trust that one certificate.** Export the router's certificate once, check its fingerprint over a channel you trust, and add it to your machine's trust store as an anchor.
5. **Option two: run a tiny private authority.** Create your own root once, add that root to your machines, and issue certificates for every device on your network from it.
6. Option two scales. Option one does not, but for one router it is fine.
7. Either way you have made a deliberate, narrow decision, instead of teaching yourself to click through warnings, which is the real long-term harm.

■ 32.8.5 TECHNICAL – the engineer's version

1. A self-signed certificate for an IP address must carry that address as an IP Address entry in the SAN extension. A DNS entry with the same digits does not match, and modern clients will reject it.
2. Adding a private root, per platform:

System	Command or place
macOS	<code>security add-trusted-cert</code>
Debian	<code>/usr/local/share/ca-certificates</code>
RHEL	<code>/etc/pki/ca-trust/source/anchors</code>
Firefox	Settings, View Certificates

3. Understand what you have done: a root you add is trusted for **every name on the internet**, not just your router. A private root whose key is on a laptop that gets stolen is a total compromise of that laptop's browsing. Use `nameConstraints` if your tooling supports it.
4. **Certificate pinning** narrows trust in the other direction. The application refuses any certificate except a specific key it was built to expect, ignoring the trust store entirely.
5. On the web, pinning was standardized as HTTP Public Key Pinning, RFC 7469 of April 2015. It was a failure: a wrong pin bricked your own site for months with no recovery. Chrome removed support in version 72, early 2019. Do not use it.
6. **HSTS**, HTTP Strict Transport Security, RFC 6797 of November 2012, is the mechanism that survived. A response header tells the browser to use HTTPS for this host for a period, and to refuse to let the user click through certificate warnings at all.
7. Here is a real one, from the `github.com` response captured earlier in this chapter:


```
strict-transport-security: max-age=31536000;
includeSubDomains; preload
```
8. `max-age=31536000` is one year in seconds. `includeSubDomains` extends it to every subdomain. `preload` is a request to be added to a list that ships inside the browser itself.
9. The preload list matters because HSTS normally needs one successful HTTPS visit before it can protect you. Preloading closes that first-visit gap by baking the rule into the browser binary.
10. HSTS is also why you cannot click through a warning on many major sites, and that is deliberate.

■ 32.8.6 WORDS – remember these

1. Self-signed — the key vouches for itself — subject equals issuer, no chain to any external anchor.
2. Trust anchor — a certificate you accept without proof — a root installed in a trust store, trusted for all names unless constrained.
3. Private CA — your own small authority — a locally generated root used to issue certificates for internal hosts.
4. Pinning — accept only one specific key — an application-level restriction that bypasses the trust store entirely.
5. HSTS — always use HTTPS for this host — the Strict-Transport-Security header from RFC 6797, which also blocks warning bypass.
6. Preload — the rule ships inside the browser — inclusion in the browser's built-in HSTS list, protecting the very first visit.

32.9 When TLS is deliberately intercepted

■ 32.9.1 PLAIN – in simple words

1. Everything in this chapter says a man in the middle cannot read your traffic, because he cannot produce a trusted certificate.
2. There is one way around that: put a certificate authority of your own into the machine's trust store.
3. Once that root is trusted, the middlebox can mint a certificate for any site on demand, and the browser will accept it happily.
4. This is not a bug. It is the system working exactly as designed. Trust stores are lists, and whoever controls the list controls the trust.
5. Employers do this on company laptops. Antivirus products do it on home machines. Developers do it on purpose with debugging tools.

■ 32.9.2 PLAIN – a picture in your head

1. Imagine your office post room is instructed to open every sealed envelope, read it, photocopy it, put the contents in a fresh envelope, and reseal it with a company seal.
2. You were also given, on your first day, a stamp that says company seals are just as good as official ones.
3. So every letter you receive looks correctly sealed. You have no way to tell, from the seal alone, that it was opened, unless you look closely at whose crest is on it.
4. That closer look is exactly what the next block teaches.

Where this comparison breaks: a post room reads letters slowly. A TLS middlebox reads everything, all day, for thousands of people, and stores it. And unlike a post room, it can also modify content on the way through.

■ 32.9.3 PLAIN – a worked example

1. This is real. The sandbox in which this chapter was written routes all outbound traffic through a company proxy.
2. Here is the actual command and the actual first lines of output when connecting to github.com:

```
openssl s_client -connect github.com:443 \  
-servername github.com </dev/null  
  
depth=2 0 = Anthropic,  
CN = sandbox-egress-gateway-production Egress Gateway CA
```

```

verify return:1
depth=1 0 = Anthropic,
  CN = Egress Gateway SDS Issuing CA (production)
verify return:1
depth=0 CN = github.com
verify return:1
...
Verification: OK
Verify return code: 0 (ok)

```

3. Read it slowly. The subject at depth 0 is `github.com`. Correct.
4. The issuer, at depth 1, is not a public certificate authority. It is the proxy. The root at depth 2 is the proxy's own root.
5. And yet the last two lines say OK. Verification passed, because that root was installed into the machine's trust store on purpose.
6. Notice the validity dates on that leaf, from the same output: `NotBefore: Aug 6 23:49:45 2026` and `NotAfter: Sep 5 23:50:45 2026`. Exactly 30 days, minted by the box, not by GitHub.
7. If this were the reader's Mac at home, the issuer line would name a public authority instead. One line of output is the whole diagnosis.

■ 32.9.4 PLAIN – what is really happening inside

1. The middlebox sits in the connection path. It sees the `ClientHello` and reads the server name extension.
2. It opens its own TLS connection outward to the real server, and validates that one properly, using the real public trust store.
3. It then generates a certificate for the requested name, signs it with its own private authority key, and completes a second, separate TLS handshake back towards you.
4. Two connections, joined in the middle, with the plaintext exposed at the join. That join is where scanning, logging or blocking happens.
5. Anything you type goes through that plaintext point: passwords, session cookies, private messages, source code.
6. Certificate pinning is the one thing that breaks this. An application that only accepts its own key refuses the forged certificate and fails loudly. That is why some mobile applications stop working on corporate networks.

■ 32.9.5 TECHNICAL – the engineer's version

1. Categories of deliberate interception you will actually meet:

Kind	Who runs it	Typical issuer text
Corporate proxy	employer	company name, "SSL"
Antivirus scan	local software	product name
Debug proxy	you	mitmproxy, Charles
Captive portal	venue	portal or gateway name

2. Antivirus TLS scanning did the same thing on consumer machines. It has a poor security record; the 2017 paper "The Security Impact of HTTPS Interception" by Durumeric and others found many interceptors downgraded the outward connection to weaker cryptography than the browser would have used.

3. Debugging proxies are the honest version: mitmproxy, Charles Proxy, Fiddler and Burp Suite all install a local root and are indispensable for development. You install the root yourself and you should remove it when done.
4. Captive portals are not usually interception in this sense. They intercept plain HTTP and DNS to redirect you to a login page, and they cannot intercept HTTPS without a warning. That is exactly why an HTTPS-only device often fails to show the portal page at all.
5. How to detect it, in order of effort:

```
openssl s_client -connect github.com:443 \  
-servername github.com </dev/null 2>/dev/null \  
| openssl x509 -noout -issuer -dates
```

6. If the issuer is not a public authority you recognize, your traffic is being terminated somewhere. Compare with the same command from a phone on mobile data.
7. Certificate Transparency helps here too: an intercepting certificate is not logged publicly, whereas a genuine one must be. Chrome enforces this only for publicly trusted roots, and deliberately exempts locally installed roots, precisely so corporate interception keeps working.
8. Say this honestly: in many workplaces this is normal, disclosed in the acceptable use policy, and legal. It is also a genuine risk, because it concentrates every employee's plaintext at one appliance, and appliances get breached.

■ 32.9.6 WORDS – remember these

1. TLS interception — somebody opens and re-seals your traffic — an on-path proxy terminating and re-originating TLS with a locally trusted root.
2. Secure web gateway — the corporate box doing it — an appliance performing inspection, filtering and data loss prevention.
3. Locally installed root — a trust anchor added by an administrator — a CA certificate placed in the machine store, exempt from Certificate Transparency enforcement.
4. mitmproxy — a developer's interception tool — an open-source intercepting proxy used for debugging and testing.
5. Captive portal — the login page on public wireless — a redirect of plain HTTP and DNS, which cannot transparently intercept HTTPS.

32.10 HTTP versions: 1.1, 2 and 3

■ 32.10.1 PLAIN – in simple words

1. HTTP/1.0 opened a fresh TCP connection for every single file. A page with 40 images meant 40 connections.
2. HTTP/1.1 added **keep-alive**: reuse one connection for many requests. That was a huge saving.
3. But on one HTTP/1.1 connection the answers must come back in the order the questions were asked.
4. So one slow file blocks everything queued behind it. That is called **head-of-line blocking**.
5. HTTP/2 fixed it properly by chopping everything into small labelled pieces, so many exchanges share one connection and interleave.
6. But HTTP/2 still runs on TCP, and TCP itself insists on delivering bytes in order. So one lost packet still stalls every stream.
7. HTTP/3 fixed that by abandoning TCP and building a new transport on UDP, called QUIC, which keeps streams genuinely independent.

■ 32.10.2 PLAIN – a picture in your head

1. HTTP/1.1 is a single-lane road. Cars arrive in order. One breakdown stops the lane.
2. HTTP/2 paints six lanes on that road. Cars can overtake. But the road is still one bridge, and if the bridge is blocked, all six lanes stop.
3. HTTP/3 replaces the bridge with six separate bridges. One collapsing affects only its own lane.

Where this comparison breaks: the HTTP/2 lanes are not physical; they are labels inside one byte stream. And the HTTP/3 bridges are still one UDP flow, sharing congestion control; only loss recovery is separated.

■ 32.10.3 PLAIN – a worked example

1. A page needs one HTML file, one stylesheet and eight images: ten items.
2. Round trip time to the server is 100 milliseconds.
3. Under HTTP/1.1 with six connections and keep-alive: six handshakes, then ten requests spread over six lanes, roughly 400 milliseconds.
4. Under HTTP/2 on one connection: one handshake, all ten requests sent at once, answers interleaved, roughly 250 milliseconds.
5. Under HTTP/3: same as HTTP/2, plus setup folded into one round trip, plus no stall if a packet is lost, so the worst case improves most.

■ 32.10.4 PLAIN – what is really happening inside

1. HTTP/2 is binary. Every message becomes **frames** with a type, a length and a stream identifier.
2. Because each frame carries its stream number, frames from different requests can be interleaved on one connection and reassembled at the far end. That is **multiplexing**.
3. Headers repeat enormously between requests on one page. HTTP/2 compresses them with HPACK, which keeps a shared table of previously seen headers so the second request sends an index instead of the text.
4. **Server push** let the server send a file you had not asked for, guessing you would need it. It sounded good and failed in practice: servers pushed things browsers already had cached, wasting bandwidth.
5. QUIC moves the reliability, ordering and encryption out of the kernel and into the application. It runs on UDP because UDP is the only transport that middleboxes will pass without inspecting.
6. QUIC merges the transport handshake and the TLS 1.3 handshake into one, so a new connection is one round trip, and a resumed one can be zero.
7. Each QUIC stream has its own delivery order. Losing a packet belonging to stream 3 does not hold up stream 7. That is the head-of-line fix.
8. **Connection migration**: a QUIC connection is identified by a connection ID, not by the four-part address pair. Move from wireless to mobile data and the connection survives with a different IP address.
9. It had to live in user space because changing TCP means changing every operating system kernel and every middlebox on earth, which takes decades. Shipping a library inside a browser takes weeks.

■ 32.10.5 TECHNICAL – the engineer's version

Version	Year	Transport	Framing
HTTP/1.0	1996	TCP	text
HTTP/1.1	1997/1999	TCP	text

Version	Year	Transport	Framing
HTTP/2	2015	TCP + TLS	binary
HTTP/3	2022	QUIC on UDP	binary

1. Specifications: HTTP/1.0 is RFC 1945, May 1996. HTTP/1.1 was RFC 2068 January 1997 and RFC 2616 June 1999, now replaced by RFC 9112, June 2022.
2. HTTP/2 is RFC 7540, May 2015, now RFC 9113, June 2022. HPACK header compression is RFC 7541, May 2015.
3. QUIC is RFC 9000, May 2021, with TLS integration in RFC 9001 and loss recovery in RFC 9002. HTTP/3 is RFC 9114, June 2022, with QPACK header compression in RFC 9204.
4. HTTP/2 requires TLS in every browser implementation, though the specification permits cleartext. The ALPN token is h2; for HTTP/3 it is h3.
5. Server push was defined in RFC 7540 and disabled by default in Chrome 106 in 2022. Google reported that only about 1.25 percent of HTTP/2 sites used it, with no clear net performance gain. The replacement is the 103 Early Hints status code, RFC 8297, December 2017.
6. HTTP/3 is discovered, not requested. The server advertises it with an Alt-Svc response header or an HTTPS DNS record, and the client upgrades on a later connection.
7. Costs of QUIC: encryption and reassembly happen in user space, so CPU cost per byte is higher than kernel TCP, and some networks rate-limit or block UDP port 443 entirely. Clients therefore always keep a TCP fallback.

■ 32.10.6 WORDS - remember these

1. Keep-alive — reuse one connection for many requests — persistent connections, default in HTTP/1.1.
2. Head-of-line blocking — one slow item stalls the rest — an in-order delivery constraint at the HTTP or TCP layer.
3. Multiplexing — many exchanges on one connection — independent streams identified by stream ID in HTTP/2 frames.
4. HPACK — squeeze repeated headers — the HTTP/2 header compression format, RFC 7541, using static and dynamic tables.
5. QUIC — a new transport on UDP — RFC 9000, providing streams, reliability and TLS 1.3 in user space.
6. Connection migration — keep the session when your address changes — QUIC's use of connection IDs instead of address pairs.

32.11 Cookies, sessions, and CORS

■ 32.11.1 PLAIN - in simple words

1. HTTP has no memory. Each request stands alone. The server does not know the last request came from you.
2. That was deliberate, and it is why the web scaled. But logging in needs memory.
3. A **cookie** is the fix. The server sends a small piece of text. The browser stores it and sends it back on every later request to that site.
4. That is all a cookie is: a string the browser repeats back automatically.
5. A **session** is what the server does with it. Usually the cookie holds a random identifier and the server keeps the real data.

6. **Authentication** is proving who you are, once. **Session management** is staying recognized afterwards. They are different jobs and are attacked differently.

■ 32.11.2 PLAIN – a picture in your head

1. You check into a hotel. At the desk you show your passport once. That is authentication.
2. They hand you a plastic key card with no name on it. That is the session cookie.
3. From then on, no one asks who you are. The card opens the door. That is session management.
4. Anyone who finds your card in the corridor can open your room, without ever seeing your passport.
5. The card stops working at checkout time, or if the desk cancels it. That is session expiry and revocation.

Where this comparison breaks: a card must be physically carried. A cookie is copied invisibly by any script that can read it, which is why `HttpOnly` exists to hide it from scripts entirely.

■ 32.11.3 PLAIN – a worked example

1. Here are three real `Set-Cookie` headers, captured live from `github.com` in this chapter. The long session value has been shortened:

```
set-cookie: _gh_sess=Xkj3u...JW0DdiSIORH1F; path=/;
  HttpOnly; secure; SameSite=Lax
set-cookie: _octo=GH1.1.1717559084.1786588114;
  expires=Fri, 13 Aug 2027 02:28:34 GMT;
  domain=.github.com; path=/; secure; SameSite=Lax
set-cookie: logged_in=no;
  expires=Fri, 13 Aug 2027 02:28:34 GMT;
  domain=.github.com; path=/; HttpOnly; secure;
  SameSite=Lax
```

2. `_gh_sess` has no expires, so it is a session cookie: it dies when the browser closes.
3. `_octo` has an expiry one year ahead, so it is persistent and survives restarts.
4. `domain=.github.com` widens it to every subdomain. Leaving `domain` out, as `_gh_sess` does, restricts it to exactly the host that set it, which is the safer default.
5. `path=/` means send it for every path on the site.
6. `secure` means never send this over plain HTTP.
7. `HttpOnly` means JavaScript cannot read it. That is what limits the damage of a cross-site scripting bug.
8. `SameSite=Lax` means do not send this cookie on cross-site requests except top-level navigations. That is the main defence against cross-site request forgery.
9. Note that `logged_in=no` is `HttpOnly` too. Even trivial state gets the protection, because it is free.

■ 32.11.4 PLAIN – what is really happening inside

1. Mechanically: the server sends `Set-Cookie` in a response header. The browser stores name, value and attributes in a jar keyed by domain and path.
2. On every later request the browser looks up matching cookies and sends them in a single `Cookie` request header, without asking anyone.
3. **Session cookie versus token.** A session cookie is an opaque identifier; the server holds the state and can revoke instantly. A token, such as a signed JWT, carries the claims inside itself; the server needs no lookup but also cannot easily revoke it before expiry.

4. **Local storage** is a separate browser store that JavaScript reads and writes explicitly. It is never sent automatically, so it is immune to cross-site request forgery, and fully exposed to cross-site scripting because scripts can read it. Neither store is simply safer than the other.
5. Now the same-origin policy. An **origin** is the triple of scheme, host and port. `https://a.com` and `https://a.com:8443` are different origins. So are `http://a.com` and `https://a.com`.
6. By default a page may send a request to another origin, but its JavaScript may not read the response. That rule, from Netscape Navigator 2.0 in 1995, is what stops a random page reading your webmail.
7. **CORS** is the controlled exception. The receiving server opts in, using response headers, to let a named origin read the answer.
8. For anything beyond a simple request, the browser first sends a **preflight**: an `OPTIONS` request asking whether the real one is allowed.
9. This is the key point that confuses everyone: the server almost always processed the request perfectly. The browser then refused to hand the response to the page. So the error appears only in the browser console, and the server log shows a normal 200.
10. It follows that `curl` never sees a CORS error, and that adding headers on the client cannot fix one. Only the server can grant permission.

■ 32.11.5 TECHNICAL – the engineer’s version

1. Cookies were invented by Lou Montulli at Netscape in 1994. The current specification is RFC 6265, April 2011, with SameSite and other modern rules in the long-running RFC 6265bis work.
2. Attributes and what they do:

Attribute	Effect
Domain	widens to subdomains
Path	limits by URL prefix
Expires / Max-Age	persistent, not session
Secure	HTTPS only
HttpOnly	hidden from JavaScript
SameSite	cross-site sending rule

3. SameSite values: Strict never sends cross-site, Lax sends only on top-level GET navigations, None always sends and requires Secure. Chrome made Lax the default in Chrome 80, rolled out during 2020.
4. Here is a real preflight and its answer, captured from a small local server written for this chapter:

```

OPTIONS /api/profile HTTP/1.1
Host: api.kedbyte.test:8099
Origin: https://app.kedbyte.test
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: content-type

HTTP/1.1 204 No Content
Access-Control-Allow-Origin: https://app.kedbyte.test
Access-Control-Allow-Methods: GET, POST, PUT
Access-Control-Allow-Headers: Content-Type, Authorization
Access-Control-Max-Age: 600
Content-Length: 0

```

5. Access-Control-Max-Age: 600 lets the browser cache that permission for ten minutes, so it does not preflight every call.

6. A request avoids preflight only if it is GET, HEAD or POST, uses no custom headers, and has a Content-Type of application/x-www-form-urlencoded, multipart/form-data or text/plain. Sending JSON triggers a preflight, which surprises everyone building an API.
7. To send cookies cross-origin you need credentials: "include" on the client and, on the server, Access-Control-Allow-Credentials: true plus an exact origin. The wildcard * is rejected with credentials, by design.

■ 32.11.6 WORDS – remember these

1. Cookie — a string the browser repeats back — a name-value pair with attributes, stored per domain and path, RFC 6265.
2. Session — being recognized after login — server-side state referenced by an opaque identifier in a cookie.
3. HttpOnly — hidden from page scripts — a cookie attribute blocking document.cookie access, limiting XSS damage.
4. SameSite — the cross-site sending rule — Strict, Lax or None, the main defence against cross-site request forgery.
5. Origin — scheme plus host plus port — the security boundary of the same-origin policy.
6. Preflight — asking permission before the real request — an OPTIONS request with Access-Control-Request-Method.
7. CORS — a server opting in to cross-origin reads — response headers telling the browser to release the response to a named origin.

32.12 The whole journey, assembled

■ 32.12.1 PLAIN – in simple words

1. This section answers the classic question in full: what happens when you type a web address and press Enter.
2. It uses the reader's own machine, the reader's own router at 192.168.0.1, the reader's own resolver at 1.1.1.1, and the address github.com resolved to 20.207.73.82.
3. Everything in Part F of this book meets here in one list.
4. The reader's real session did not finish this journey. It stopped at a precise step, and the list below says exactly which one.

■ 32.12.2 PLAIN – a picture in your head

1. Think of posting a letter to a company in another country.
2. You look up their address in a directory. That is DNS.
3. You walk to the post box, which is the only exit from your building. That is the default gateway.
4. The letter changes hands between postal services, none of which reads it. That is routing.
5. At the far end, a receptionist confirms the company's identity to you before you hand over anything valuable. That is the TLS handshake.

Where this comparison breaks: a letter goes once. A web page needs dozens of these exchanges, most of them to other companies entirely, and they overlap.

■ 32.12.3 PLAIN – a worked example

1. The complete walkthrough. Times are realistic for the reader's connection.

| key press -> DNS -> TCP -> TLS -> HTTP -> paint

2. Step 1. A key is pressed. The keyboard sends a scan code; the operating system turns it into a character and gives it to the browser.
3. Step 2. The browser's address bar decides: is this a search or a URL. A dot and no spaces means URL.
4. Step 3. The host name is normalized: lowercased, and if it contains non-ASCII it is converted to Punycode.
5. Step 4. The browser checks its HSTS list. `github.com` is preloaded, so even `http://` would be rewritten to `https://` before anything is sent.
6. Step 5. The browser checks its own HTTP cache. A fresh copy would end the story here with no network at all.
7. Step 6. The browser checks whether it already has an open connection to this host. Reusing one skips steps 8 to 26 entirely.
8. Step 7. Name resolution starts. The browser's own DNS cache is checked.
9. Step 8. Miss. The Mac sends a DNS query to the configured resolver, `1.1.1.1`, not to the router. This is the reader's real configuration.
10. Step 9. To send that packet the Mac needs a next hop. `1.1.1.1` is not on the local subnet, so the route table selects the default gateway, `192.168.0.1`.
11. Step 10. The Mac needs the gateway's hardware address, so it sends an ARP request for `192.168.0.1` and caches the reply.
12. Step 11. The resolver at `1.1.1.1` answers, from cache or by walking the root, then `.com`, then GitHub's own name servers.
13. Step 12. The answer arrives: `github.com` is `20.207.73.82`, with a time to live saying how long it may be cached.
14. Step 13. That address is in a Microsoft-owned range. GitHub has been owned by Microsoft since 2018 and fronts traffic through Microsoft's network edge.
15. Step 14. IPv6 was reported as none on this connection, so there is no happy-eyeballs race between families. IPv4 only.
16. Step 15. The browser opens a socket and starts a TCP connection to `20.207.73.82` port 443.
17. Step 16. The SYN goes to the gateway, `192.168.0.1`, which is hop 1 of the reader's traceroute.
18. Step 17. Hop 3, `137.97.29.249`, is a public address on the same ISP.
19. Step 18. Several hops showed two or three addresses. That is per-flow load balancing across parallel links, not an error.
20. Step 19. **This is where the reader's session stopped.** The SYN went out and nothing came back. No SYN-ACK, no reset, no ICMP unreachable. Silence, then a 15 second timeout in `curl`.
21. Step 20. What that proves: the connection never opened, so not one byte of TLS or HTTP was ever attempted. Steps 27 onwards did not happen on that day.
22. Step 21. In the normal case the server replies SYN-ACK and the client sends ACK. One round trip. The connection is open.
23. Step 22. TLS begins. The client sends `ClientHello` with its versions, random bytes, cipher suites, the `server_name` extension carrying `github.com`, ALPN offering `h2` and `http/1.1`, and a `key_share`.
24. Step 23. The server replies `ServerHello` with its chosen suite and its own `key_share`. Both sides now compute the shared secret.
25. Step 24. Everything after this point is encrypted, including the certificate.
26. Step 25. The server sends `EncryptedExtensions`, `Certificate`, `CertificateVerify` and `Finished`.
27. Step 26. The browser builds a chain from the leaf through the intermediate to a root in its trust

store.

28. Step 27. It checks dates, basicConstraints, key usage, and that github.com appears in the certificate's SAN list.
29. Step 28. It checks Certificate Transparency proofs, and revocation, the latter weakly.
30. Step 29. It verifies CertificateVerify, which proves the server holds the private key right now.
31. Step 30. The client sends its own Finished. The handshake is complete in one round trip. Measured here, about 0.22 seconds.
32. Step 31. ALPN selected h2, so the browser speaks HTTP/2 and sends its request as compressed binary frames.
33. Step 32. The request may not reach GitHub's own servers at all. It may be answered by a load balancer or CDN edge that terminated TLS. Section 32.13.
34. Step 33. The response headers come back: 200 OK, content-type, strict-transport-security, set-cookie, content-security-policy.
35. Step 34. The body streams in, usually compressed with gzip or brotli, and is decompressed on arrival.
36. Step 35. The HTML parser starts building the DOM tree while bytes are still arriving.
37. Step 36. It meets links to stylesheets, scripts, fonts and images, and issues those requests immediately over the same connection.
38. Step 37. DOM and CSSOM combine into a render tree of things that will actually be drawn.
39. Step 38. Layout computes the exact box and position of every element.
40. Step 39. Paint fills in pixels per layer, and compositing assembles the layers, often on the GPU.
41. Step 40. Remaining scripts run, fonts swap in, images decode, and the page fires its load event.
42. Step 41. The connection stays open for later requests, and a NewSessionTicket is stored so the next visit can resume faster.

■ 32.12.4 PLAIN - what is really happening inside

1. Count the round trips in the good case: one for DNS, one for TCP, one for TLS 1.3. Three before the first HTTP byte.
2. The reader's failure sat in the middle of that list, at the TCP layer, and the shape of the failure was the clue.
3. A refused connection sends a reset. A dead server on a live network often produces an ICMP unreachable. A silent drop produces nothing.
4. Silence is what a firewall or filter configured to drop rather than reject produces, and it is also what a broken path produces.
5. What is proven: no response of any kind on that path, while the same request succeeded over mobile data from the same phone.
6. What is only suggested: where on the path the drop happened, and who caused it. The evidence does not name a party, and neither will we.

■ 32.12.5 TECHNICAL - the engineer's version

1. The layered view, with the reader's real values.

Layer	Thing	Reader's value
-----	-----	-----
Application	HTTP/2 request	GET / on github.com
Security	TLS 1.3 record	AES-GCM, X25519
Transport	TCP segment	dst port 443
Network	IPv4 packet	dst 20.207.73.82
Link	Ethernet / 802.11	next hop 192.168.0.1

2. Commands that observe each step, in order: `dig github.com`, `dig @1.1.1.1 github.com`, `netstat -rn` or `route -n`, `get default`, `arp -a`, `tracert github.com`, `nc -vz github.com 443`, `curl -v https://github.com`, `openssl s_client -connect github.com:443`, and the browser's network panel.
3. The single most useful diagnostic split: if `nc -vz host 443` succeeds but `curl` fails, the problem is above TCP. If `nc` itself hangs with no response, the problem is at or below TCP, which is exactly what happened here.

■ 32.12.6 WORDS – remember these

1. Happy eyeballs — trying IPv6 and IPv4 together — RFC 8305, racing address families so a broken one costs milliseconds, not seconds.
2. Punycode — non-ASCII names in ASCII form — the encoding that turns international domain names into xn- - labels.
3. Critical rendering path — the shortest route to first pixels — the DOM, CSSOM, render tree, layout, paint and composite sequence.
4. Round trip time — one message out and back — the quantity that dominates page load time on long paths.
5. Silent drop — a packet discarded with no reply — the failure mode that produces a timeout rather than an error, as in the reader's session.

32.13 CDNs, load balancers and caching in front

■ 32.13.1 PLAIN – in simple words

1. The server you connect to is usually not the server that owns the site.
2. In front of it sit machines whose job is to be closer to you, to spread load, and to answer from memory when they can.
3. A **load balancer** takes one incoming connection and hands the work to one of many identical servers behind it.
4. A **CDN**, a content delivery network, is a large set of such machines placed in many cities, so most users reach one within a few milliseconds.
5. The real server behind all of it is called the **origin**.
6. This is why `github.com` answered from an edge site rather than from one room somewhere.

■ 32.13.2 PLAIN – a picture in your head

1. A national newspaper does not post every copy from its head office.
2. It prints in regional plants near readers. Delivery is fast because the paper travelled a short distance.
3. The head office is the origin. The regional plants are the edge.
4. If a story changes after printing, every plant must be told to reprint. That is cache invalidation, and it is genuinely hard.

Where this comparison breaks: a newspaper is identical for everyone. Web responses vary by user, by language and by device, so the edge must know exactly which parts are shareable and which are personal.

■ 32.13.3 PLAIN – a worked example

1. Real evidence from this chapter's live capture of `github.com`:

```
x-github-edge-region: iad
server: github.com
etag: W/"6a7d294b-8e2"
last-modified: Thu, 13 Aug 2026 02:17:47 GMT
cache-control: no-cache
```

2. `x-github-edge-region: iad` names the edge site, using the airport code for Washington Dulles. The answer came from an edge machine.
3. `etag` is a version tag for the content. On the next request the browser may send `If-None-Match` with that tag.
4. If the content has not changed, the server replies `304 Not Modified` with no body. That is a full round trip saved on bytes, though not on latency.
5. `cache-control: no-cache` does not mean “do not cache”. It means “you may store it, but revalidate before every use”. The header that forbids storage is `no-store`.

■ 32.13.4 PLAIN – what is really happening inside

1. TLS is normally terminated at the edge. The edge holds a certificate and a private key for the site's name, decrypts your request, and then opens its own connection to the origin.
2. So the CDN sees your plaintext. That is unavoidable if it is going to cache or inspect anything.
3. That adds a party you must trust: your data is private from the network, and fully visible to the CDN operator.
4. Cache freshness has two mechanisms. A **time to live** says how long a copy may be used without asking. **Validation** asks the origin whether a stored copy is still good, using `ETag` or `Last-Modified`.
5. Invalidation is the hard part, because copies live in thousands of places. The industry answer is to avoid it: give every version of a file a unique name, cache it forever, and change the name when the content changes.
6. That is why built assets have hashed file names, and why the HTML that points at them is served with a very short cache time.

■ 32.13.5 TECHNICAL – the engineer's version

1. Caching is specified in RFC 9111, June 2022. The directives that matter:

Directive	Meaning
<code>max-age=N</code>	fresh for N seconds
<code>s-maxage=N</code>	N seconds, shared caches
<code>no-cache</code>	store, but revalidate
<code>no-store</code>	do not store at all
<code>private</code>	browser only, not CDN
<code>immutable</code>	never revalidate
<code>stale-while-revalidate</code>	serve old, refresh

2. `Vary` tells caches which request headers change the answer. The real capture above showed `vary: Accept-Encoding, Accept, X-Requested-With`. A `Vary` on a high-cardinality header such as `User-Agent` destroys hit rates.

3. Layers you may meet in front of an origin, in order: DNS-based traffic steering or anycast, a network load balancer at layer 4, a reverse proxy or application load balancer at layer 7, a web application firewall, and the cache itself.
4. Anycast means the same IP address is announced from many locations, and routing delivers you to the nearest. It is why one address such as 1.1.1.1 answers from a different machine in every country.
5. The trade-off, stated plainly: a CDN reduces latency, absorbs attacks, and cuts origin cost. It also inserts a company that can read, modify, log and block every request, and whose outage becomes your outage.

■ 32.13.6 WORDS – remember these

1. Origin — the real server behind everything — the authoritative source a cache or proxy fetches from on a miss.
2. Edge — a machine near the user — a point of presence terminating TLS and serving cached content.
3. TLS termination — where encryption ends — the point at which the request becomes plaintext, usually the edge, not the origin.
4. ETag — a version tag for content — an opaque validator compared with If-None-Match to produce a 304 Not Modified.
5. Anycast — one address, many locations — the same prefix announced from many sites, with routing choosing the nearest.
6. Cache invalidation — telling every copy to forget — the hard problem usually avoided by giving each version a unique name.

32.98 Common wrong ideas

1. Wrong: the padlock means the site is safe. Right: it means the connection to whoever controls that name is private and untampered. Criminals get certificates too, in seconds, for free.
2. Wrong: HTTPS hides which site I visit. Right: the destination IP address is always visible, and until Encrypted Client Hello, RFC 9849 of March 2026, the server name was in the clear as well. DNS may leak it too.
3. Wrong: self-signed means insecure. Right: it means unverified. The encryption is identical. What is missing is any proof of who is at the other end, which matters enormously on a public site and hardly at all on your own router at 192.168.0.1.
4. Wrong: encryption alone is enough. Right: without authentication you may have a perfectly encrypted tunnel to the attacker. Authenticity comes first; confidentiality is only useful afterwards.
5. Wrong: a certificate authority checks that the company is real. Right: almost all certificates are Domain Validated, which proves control of a name for a few seconds and nothing else.
6. Wrong: a CORS error means the server rejected my request. Right: the server usually processed it fine. The browser refused to give the response to your script. Only the server can grant permission.
7. Wrong: HTTP/2 solved head-of-line blocking. Right: it solved it at the HTTP layer, but TCP still delivers in order, so one lost packet still stalls every stream. HTTP/3 over QUIC fixed the transport layer.
8. Wrong: my company cannot read my HTTPS traffic. Right: if they installed a root certificate on the machine, they can read all of it, and one `openssl s_client` command shows you whether they do.

32.99 Chapter summary in 20 lines

1. HTTP is plain text: a request line, headers, a blank line, and a body. HTTPS is exactly that, carried inside TLS.
2. Methods differ by whether they are safe and whether they are idempotent, and those two properties drive caching, retries and 0-RTT rules.
3. Status codes come in five families; 4xx blames the request and 5xx blames the server, while 502 and 504 usually blame something in front of it.
4. TLS solves three problems: confidentiality, integrity and authenticity.
5. Encryption without authentication is nearly useless, because the tunnel may end at the attacker.
6. The building blocks are symmetric ciphers, hashes, message authentication codes, public key pairs and key exchange.
7. Diffie-Hellman lets two strangers agree a secret in public, using modular exponentiation that nobody can reverse at real key sizes.
8. Elliptic curves replaced RSA for new work because P-256 matches RSA 3072 in about a fifth of the bytes.
9. Ephemeral keys give forward secrecy, so a stolen server key cannot decrypt yesterday's recorded traffic.
10. TLS 1.2, RFC 5246 of 2008, needs two round trips. TLS 1.3, RFC 8446 of August 2018, needs one, by sending a key share in the first message.
11. 0-RTT saves another round trip and has no replay protection, so use it only for safe, idempotent requests.
12. A certificate binds a name to a public key and is trusted only because of who signed it, never because of what it says.
13. The chain runs leaf, intermediate, root; intermediates exist so the root key can stay offline and a compromise can be repaired.
14. Browsers check chain, dates, hostname against the SAN list, basic constraints and, in Chrome, Certificate Transparency.
15. Browsers do not check honesty, legitimacy, content safety, or whether the brand matches the name. The padlock never meant safe.
16. Revocation is largely broken; the industry answer is short-lived certificates, falling to a 47 day maximum from March 2029.
17. On your own router the endpoint is known and the link is one hop, so accepting a self-signed certificate once is a small risk. On a public site the same click may be surrendering to an active interception.
18. A locally installed root lets a proxy read everything. One `openssl s_client` command reveals it, by showing the issuer.
19. HTTP/2 multiplexed over one TCP connection; HTTP/3 over QUIC, RFC 9114 of June 2022, moved to UDP to remove head-of-line blocking in the transport.
20. Cookies give HTTP the memory it lacks; the same-origin policy and CORS decide who may read what, and a CORS error is always the browser's decision, never the server's.

The OSI and TCP/IP Models, Retrofitted to Your Own Session

33.0 What this chapter gives you

1. You will be able to say why networks are built in layers at all, what each layer is allowed to know, and what it is forbidden to know.
2. You will be able to name all seven OSI layers, define each one, and say what unit of data each one handles.
3. You will be able to explain that the OSI model came from a standards effort begun in 1977, published as ISO 7498 in 1984, and that the protocol suite it was built for lost to TCP/IP while the model survived as a teaching tool.
4. You will be able to name the four layers of the TCP/IP model, cite RFC 1122 from October 1989 as the document that states them, and map the two models against each other with real protocols in every row.
5. You will be able to place every single thing from the reader's own broken session at exactly one layer, and say which chapter covered it.
6. You will be able to walk one real HTTP request down the stack with real header sizes, and state what fraction of the bytes on the air were payload.
7. You will be able to show, with evidence, that layers 1, 2 and 3 worked in that session, that layer 4 did not complete, and that layers 5 to 7 were never reached at all.
8. You will be able to use the general fault rule: find the highest layer that works and the lowest that fails, and the fault is between them.
9. You will be able to list the places the model is simply wrong about reality, including TLS, VPNs, MPLS, NAT and QUIC, without losing faith in the model.
10. You will be able to spot the same layering pattern in graphics, storage and compilers, and see it as general engineering rather than a networking curiosity.

33.1 Why layers exist at all

■ 33.1.1 PLAIN – in simple words

1. A network has to do many different jobs at once.
2. It must turn bits into a real signal, get data across one hop, get data across the world, notice losses and resend them, hide the contents from watchers, and agree what the contents mean.
3. So we split the work into **layers**. A layer is one level of the job.
4. Each layer does one kind of work. Each layer uses the layer below it and does not care how that layer works. Each layer offers a service upward and does not care what the layer above does with it.
5. The benefit is large. You can rip out one layer, put in a completely different one, and nothing above it needs to change.
6. The reader can carry the laptop out of the flat, join a friend's wired network instead of the home WiFi, and the same `git push` works.
7. And when a failure crosses layers, nobody owns it, because each layer can honestly say it did its own job correctly.

■ 33.1.2 PLAIN – a picture in your head

1. Think about posting a birthday card to a cousin in another city.
2. You put it in an envelope and write the address. Now the card can be handled by people who will never read it.
3. A worker empties the post box into a sack with a label saying which sorting office it goes to. A van carries the sack down a road.
4. That is layering. Each level has its own wrapping, its own address format, and its own workers who ignore everything else.

Where this comparison breaks:

5. In the post, the envelope survives the whole journey. In a network, the outer wrapping is thrown away and rebuilt at every single router. Only the middle wrapping, the IP header, travels end to end.
6. The post does not chop your card into forty numbered pieces, send them by different roads, and glue them back together. TCP does exactly that.
7. And in the post the address on the envelope is the real address of the real house. On the internet, the address on the reader's packets was rewritten twice on the way out, by two different machines. That is NAT, Chapter 24.

■ 33.1.3 PLAIN – a worked example

1. Here is the benefit, made concrete. The reader is on home WiFi in a flat in India, with a router at 192.168.0.1. Now imagine plugging in a cable.

Layer of the job	On WiFi	On a cable
Radio or wire	5 GHz radio	copper pair
Link framing	802.11 frame	Ethernet frame
Link addresses	MAC address	MAC address
Internet address	192.168.0.x	192.168.0.x
Connection	TCP port 443	TCP port 443
Encryption	TLS 1.3	TLS 1.3

Layer of the job	On WiFi	On a cable
The request	git over HTTPS	git over HTTPS

2. The git program has no code in it for radio and none for copper. It does not know which is in use and cannot easily find out.
3. WiFi sometimes loses a frame because somebody switched on a microwave oven.
4. TCP was designed for wires, where “something is missing” almost always means the network is too busy. So TCP does the thing that helps a busy network. It slows down.
5. That is the price of the rule that a layer may not look below itself. It is also why WiFi does its own resending, so TCP never finds out.

■ 33.1.4 PLAIN – what is really happening inside

1. A layer is not a place. It is a piece of code with a defined edge, and that edge is an **interface**: the short list of things the layer above is allowed to ask for.
2. The internet layer offers upward, roughly: “give me a lump of bytes and a destination address, and I will try to deliver it”.
3. The transport layer offers a much stronger promise: “give me a stream of bytes and I will deliver them all, in order, exactly once”.
4. Going down, each layer adds a small block of information at the front, called a **header**, holding only what that layer needs: addresses, lengths, numbers, flags.
5. So a packet is boxes inside boxes, with the message innermost. That wrapping is called **encapsulation**, and section 33.5 measures it in real bytes.
6. One strict rule: a layer must be replaceable without the layer above noticing. If replacing it breaks something above, the boundary was drawn in the wrong place.

■ 33.1.5 TECHNICAL – the engineer’s version

1. The OSI vocabulary is worth knowing because it still appears in specs. A layer N entity talks to its peer using a **protocol**, and to layer N-1 on the same machine using a **service**.
2. A PDU at one layer is an SDU at the layer below. That is the cleanest one-line definition of encapsulation there is.
3. The costs of layering are documented, not folklore. RFC 3439, “Some Internet Architectural Guidelines and Philosophy”, by Randy Bush and David Meyer, December 2002, contains a section titled “Layering Considered Harmful”.

Cost of layering	Concrete case
Duplicated work	802.11 and TCP both resend
Hidden state	TCP cannot see radio loss
Header overhead	128 bytes on a 190-byte GET
Fault ownership gaps	Every layer says “not me”

4. The counterweight is the end-to-end argument, from Saltzer, Reed and Clark, “End-to-End Arguments in System Design”, presented in 1981 and published in ACM Transactions on Computer Systems in 1984.
5. Its claim: a function such as reliability can only be completely and correctly implemented at the endpoints, so putting it in the network is at best a performance optimization, never a substitute.

6. Experts genuinely disagree about how strictly boundaries should hold. One camp treats any cross-layer visibility as technical debt. The other says wireless, mobility and congestion control cannot be done well without it.
7. Tools that isolate one layer on macOS: `ifconfig` for layers 1 and 2, `arp -a` for the layer 2 to 3 mapping, `netstat -rn` and `traceroute` for layer 3, `nc -vz` for layer 4, `openssl s_client` for TLS, and `curl -v` and `dig` for layer 7.

■ 33.1.6 WORDS – remember these

1. Layer — one level of the job — a set of functions with a service interface upward and a peer protocol to the same layer remotely.
2. Header — the small label a layer adds — a prefix carrying that layer's control information only.
3. Encapsulation — a box inside a box — a layer N PDU becoming the payload, or SDU, of a layer N-1 PDU.
4. PDU — the parcel a layer sends — protocol data unit, the unit exchanged between peer entities at one layer.
5. End-to-end argument — the ends must check, the middle cannot be trusted to — the principle from Saltzer, Reed and Clark, 1981 and 1984.

33.2 The OSI seven-layer model

■ 33.2.1 PLAIN – in simple words

1. In the late 1970s a large international committee set out to write down, once and for all, how computer networks should be built.
2. They split the job into seven layers, numbered 1 at the bottom to 7 at the top. Here they are, bottom first.
3. Layer 1, **physical**. Turning bits into something real: a voltage, a light pulse, a radio wave. It handles single bits.
4. Layer 2, **data link**. Getting a chunk of data across one hop, to the next box on the same local network. It handles frames.
5. Layer 3, **network**. Getting a chunk of data across many hops to a machine anywhere in the world. It handles packets.
6. Layer 4, **transport**. Making delivery reliable and ordered, and choosing which program on the machine gets it. It handles segments.
7. Layer 5, **session**. Starting, pausing and stopping a long conversation.
8. Layer 6, **presentation**. Agreeing how things are written down, and doing encrypting and squashing.
9. Layer 7, **application**. The thing you wanted: fetching a page, sending mail, pushing code.
10. Now the honest part. Layers 1, 2, 3 and 4 describe real things you can point at and measure.
11. Layers 5 and 6 barely describe anything that exists. In real systems that work is done inside the application or inside a library.

■ 33.2.2 PLAIN – a picture in your head

1. Think of a large company that wrote a rulebook for how any two of its offices must exchange goods.
2. The rulebook has seven chapters. Chapter 1 covers the loading bay, 2 the local van, 3 the national route, 4 checking nothing was lost. Chapters 5, 6 and 7 cover the meeting: opening it, agreeing a language, and the business.

3. Then a smaller, scruffier company down the road ships goods using four scribbled pages, and wins every customer, because the four pages already worked while the seven chapters were still being argued over.
4. Twenty years later every logistics course still teaches the seven chapters, because they are the clearest description of the job. Everybody actually ships using the four scribbled pages.

Where this comparison breaks:

5. The OSI effort was not slow because the people were bad. It was slow because it was a genuine international consensus process across many governments and many computer makers, and consensus is slow.

■ 33.2.3 PLAIN – a worked example

1. Place one ordinary thing at all seven layers: the reader's laptop asking `github.com` for something.
2. Layer 7: the intent, "give me the list of branches", expressed in HTTP.
3. Layer 6: the text is written as UTF-8 bytes and the whole thing is encrypted by TLS. Both are presentation-flavoured jobs.
4. Layer 5: the connection is opened, reused for several requests, then closed. In practice HTTP and TLS do this themselves. No separate session program exists.
5. Layer 4: TCP opens a connection to port 443, numbers every byte, and resends anything lost.
6. Layer 3: IP puts `20.207.73.82` on the front and asks the network to carry it.
7. Layer 2: `802.11` puts the router's hardware address on the front and sends it over WiFi to `192.168.0.1`.
8. Layer 1: the radio turns those bits into a 5 GHz signal in the air.
9. Notice how thin layers 5 and 6 were. That is not a flaw in the example. That is what the model looks like when you apply it honestly.

■ 33.2.4 PLAIN – what is really happening inside

1. The unit of data has a different name at each layer, and engineers use the names precisely.
2. Bits at layer 1. Frames at layer 2. Packets at layer 3. Segments at layer 4 for TCP, datagrams for UDP. Just "data" at layers 5, 6 and 7.
3. Using the wrong word signals that you have not understood the model. A router does not forward frames end to end. It forwards packets and builds a new frame for each hop.
4. Layer 2 uses a MAC address, burned into the hardware, meaningful only on the local link.
5. Layer 3 uses an IP address, meaningful worldwide, changeable by software in a second.
6. Layer 4 uses a port number, which picks a program on one machine.
7. Four naming systems, and three lookups between them: DNS turns names into IP addresses, ARP turns IP addresses into MAC addresses, and the operating system turns port numbers into programs.

■ 33.2.5 TECHNICAL – the engineer's version

1. The layers with their PDU names and real examples.

Layer	PDU	Real example
7 Application	data	HTTP, SMTP, DNS, SSH
6 Presentation	data	TLS, ASN.1 DER, JPEG
5 Session	data	RPC, NetBIOS, SIP
4 Transport	segment	TCP, UDP, SCTP, QUIC

Layer	PDU	Real example
3 Network	packet	IPv4, IPv6, ICMP
2 Data link	frame	Ethernet, 802.11, PPP
1 Physical	bit	1000BASE-T, 802.11ac

2. Layer 1, physical: the mechanical, electrical, functional and procedural means to activate and maintain a connection for bit transmission. It defines connectors, voltages, modulation and timing, and has no notion of a message.
3. Layer 2, data link: transfer across a single physical link, including framing, physical addressing, error detection and, on shared media, access control. It is bounded by the link. It does not route.
4. Layer 3, network: delivery across an arbitrary number of intermediate links, including logical addressing, routing and fragmentation.
5. Layer 4, transport: end-to-end delivery between processes, including multiplexing by port, reliability, ordering and flow control.
6. Layer 5, session: establishment, management and termination of dialogues, including checkpointing and synchronization points.
7. Layer 6, presentation: syntax and semantics of transferred data, including representation conversion, compression and encryption.
8. Layer 7, application: the interface to the application process, not the application itself.
9. The history, with dates. Work began in ISO Technical Committee 97, Subcommittee 16, in 1977, created specifically to produce an open systems architecture.
10. Charles Bachman, at Honeywell, is generally credited with proposing the seven-layer structure to that subcommittee in 1977 and 1978. He had won the ACM Turing Award in 1973 for his database work.
11. Hubert Zimmermann published the readable statement of it, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection", in IEEE Transactions on Communications, April 1980.
12. The model was published as the international standard ISO 7498 in 1984, and in parallel by CCITT, now ITU-T, as Recommendation X.200.
13. OSI was never only a model. It was a full protocol suite meant to replace everything: X.25 and CLNP at the network layer, TP0 to TP4 at transport, X.400 for mail, X.500 for directory, FTAM for files.
14. Governments mandated it. In the United States, FIPS 146, the Government OSI Profile or GOSIP, made OSI protocols a requirement for federal procurement from August 1990. The 1995 revision permitted TCP/IP and the mandate ended.
15. Why it lost is argued about, but the usual reasons are: TCP/IP was already deployed and free, the OSI specifications were large and expensive to obtain, implementations lagged the standards by years, and the ARPANET had completed its own switch to TCP/IP on 1 January 1983.
16. Andrew L. Russell's article "OSI: The Internet That Wasn't", IEEE Spectrum, July 2013, is the standard readable account.
17. Now the part most courses omit: OSI pieces that survived and that you use.

OSI-derived thing	Where you use it today
X.509 certificates	Every HTTPS connection
ASN.1 and DER	Certificate encoding, SNMP
IS-IS routing	Large ISP backbones

OSI-derived thing	Where you use it today
LDAP from X.500	Corporate directories

18. IS-IS deserves a pause. It is ISO/IEC 10589, an OSI routing protocol, adapted to carry IP by RFC 1195 in December 1990, and it runs inside some of the largest carrier backbones in the world.
19. X.509 came out of the X.500 directory work in 1988. Every TLS certificate the reader's browser checks is an X.509 certificate encoded in ASN.1 DER. Both are OSI presentation-layer artefacts.
20. On layers 5 and 6, be blunt. No mainstream internet protocol implements a separate session or presentation entity. RFC 1122 does not have them. Anyone who says TLS "is layer 6" is using a convenient label, not a standard.

■ 33.2.6 WORDS – remember these

1. OSI — the seven-layer teaching model — Open Systems Interconnection Basic Reference Model, ISO 7498:1984 and ITU-T X.200.
2. Frame — the parcel on one hop — the layer 2 PDU, with link addresses and a frame check sequence.
3. Packet — the parcel for the whole journey — the layer 3 PDU, with source and destination IP addresses.
4. X.509 — the format of a website's certificate — the ITU-T certificate standard from the 1988 X.500 series, encoded in ASN.1 DER.
5. IS-IS — a routing protocol from the OSI family — ISO/IEC 10589, extended for IP by RFC 1195, widely used in carrier cores.
6. GOSIP — the government rule that OSI must be bought — US FIPS 146, mandatory from August 1990, relaxed in 1995.

33.3 The TCP/IP four-layer model

■ 33.3.1 PLAIN – in simple words

1. The internet you are actually using was not built from the seven-layer model. It was built from a smaller, older, plainer set of ideas.
2. That set has four layers.
3. The **link** layer: getting bits across one hop. This is the OSI physical and data link layers rolled into one.
4. The **internet** layer: getting a packet across the world. This is IP, and only IP.
5. The **transport** layer: getting a stream to the right program, reliably. This is TCP and UDP.
6. The **application** layer: everything else. HTTP, DNS, SSH, mail, git. This is OSI layers 5, 6 and 7 rolled into one.
7. Notice the shape: a thin waist in the middle. There is exactly one internet layer protocol, IP, and everything above and below is plural.
8. That shape is called the hourglass. Many link technologies below, many applications above, one IP in the middle.

■ 33.3.2 PLAIN – a picture in your head

1. Picture an hourglass made of glass, standing on a table.
2. The wide top is full of sand grains of every colour: web, mail, video calls, games, git, software updates, streaming.
3. The wide bottom is also full of different grains: WiFi, Ethernet, fibre, 4G, 5G, satellite, Bluetooth.

4. The narrow neck is one single grain type. That is IP. Every grain from the top must pass through it.
5. Because the neck is one thing everybody agreed on, the top and the bottom can each change completely without asking permission.

Where this comparison breaks:

6. The neck is not as narrow as the picture suggests. There are two IPs now, version 4 and version 6, and making them coexist has taken thirty years and is unfinished. The reader's machine reported IPv6: (none), which is that unfinished job showing up in a flat in India.
7. And the neck is being widened from above. QUIC, in section 33.8, pushed transport logic up into applications, which the clean picture does not allow.

■ 33.3.3 PLAIN - a worked example

1. Take the reader's `curl -v https://github.com` and label it in four layers instead of seven.

Application	HTTPS request for github.com, plus the TLS handshake, plus the earlier DNS query
Transport	TCP connection to 20.207.73.82 port 443
Internet	IPv4 packets, source 192.168.0.x, destination 20.207.73.82, TTL counting down
Link	802.11 frames over 5 GHz WiFi to the MAC address of 192.168.0.1

2. Try the same with seven layers and you will find yourself arguing about whether TLS is 5 or 6, and where the DNS query belongs.

■ 33.3.4 PLAIN - what is really happening inside

1. The four-layer model is written into the document that tells you how to build a host: RFC 1122, October 1989, "Requirements for Internet Hosts - Communication Layers", edited by Robert Braden.
2. Its companion RFC 1123, published the same month, covers the application layer in detail. Together they are the closest thing the internet has to a constitution for end machines, and they are still cited in new work.
3. The reason the internet model came first is chronology. IP was specified in RFC 791 in September 1981 and TCP in RFC 793 the same month. ISO 7498 appeared in 1984. Working code beat the committee by three years.

■ 33.3.5 TECHNICAL - the engineer's version

1. The mapping table, with real protocols in every row. This is the one to memorize.

OSI layer	TCP/IP layer	Real protocols
7 Application	Application	HTTP, DNS, SSH, SMTP
6 Presentation	Application	TLS, MIME, JPEG
5 Session	Application	HTTP keep-alive, RPC
4 Transport	Transport	TCP, UDP, QUIC, SCTP
3 Network	Internet	IPv4, IPv6, ICMP, ARP
2 Data link	Link	Ethernet, 802.11, PPP
1 Physical	Link	1000BASE-T, 802.11ax

2. ARP is placed at “Internet” above, and that is a compromise. ARP has its own EtherType, 0x0806, and rides directly on Ethernet, so it is not carried by IP, yet its entire purpose is to serve IP. RFC 1122 discusses it under the link layer. Textbooks place it at 2, at 3 and at “2.5”. All are defensible.
3. ICMP is carried inside IP packets, protocol number 1, so structurally it sits above IP. Yet it is universally called a layer 3 protocol because its job is network control. RFC 1122 requires every host to implement it.
4. Numbers worth carrying for the protocols in that table:

Protocol	Document	Year
UDP	RFC 768	1980
IPv4	RFC 791	1981
TCP	RFC 793, now 9293	1981, 2022
ICMP	RFC 792	1981
ARP	RFC 826	1982
Host requirements	RFC 1122	1989
IPv6	RFC 8200	2017
TLS 1.3	RFC 8446	2018
QUIC	RFC 9000	2021
HTTP/1.1	RFC 9112	2022

5. RFC 9293, August 2022, is the current consolidated TCP specification. RFC 793 is obsolete as a citation, though every engineer still says “793” out of habit. RFC 8200, July 2017, is IPv6 as a full internet standard, replacing RFC 2460 from 1998.

■ 33.3.6 WORDS – remember these

1. TCP/IP model — the four-layer picture of the real internet — the internet architecture as stated normatively in RFC 1122, October 1989.
2. Link layer — the “one hop” level — OSI layers 1 and 2 combined: framing, link addressing and the physical medium.
3. Internet layer — the “whole world” level — IP, plus ICMP and the routing protocols that populate the forwarding table.
4. Hourglass — many things above, many below, one in the middle — the property that IP is the single universal convergence layer.

33.4 The reader’s own session, layer by layer

■ 33.4.1 PLAIN – in simple words

1. Everything the reader saw in one broken evening is going to be put in its place in the model, one layer at a time.
2. That is worth doing for one reason: once a thing has a layer, a fault in it has a layer, and a fault with a layer is a fault you can find.
3. The setting: a flat in India, a home broadband line, a Mac on WiFi, and a `git push` to GitHub that would not go.
4. The evidence gathered was a traceroute that reached twelve hops, a `curl` that sat silent for fifteen seconds, a DNS answer that was correct, and the same site loading instantly on mobile data.

■ 33.4.2 PLAIN – a picture in your head

1. Picture a tall office block with seven floors and a parcel chute down the middle.
2. On floor 7 somebody writes a note: “send me the list of branches”. They fold it into a bag, label the bag, and drop it to floor 6.
3. Floor 5 labels which conversation it belongs to. Floor 4 adds a piece number so nothing is lost. Floor 3 adds the street address of the far building. Floor 2 names the next van driver. Floor 1 loads the van.
4. At the far building the same seven floors unwrap in the opposite order, each removing only its own label, until floor 7 reads the note.

Where this comparison breaks:

5. Nothing physically moves down the floors. The message is written once into one block of memory, and each layer writes its header into space reserved in front of it. There is no copying and no chute.
6. And the labels are not equal. The floor 3 label, the IP address, is the only one that survives the whole journey. The floor 2 label is torn off and rewritten at every building on the route.

■ 33.4.3 PLAIN – a worked example

1. Here is the whole session sorted by layer. This table is the answer to “where does everything I have learned actually go”.

Layer	From the reader's session	Where covered
1 Physical	5 GHz WiFi radio in air	Ch 23
1 Physical	Signal strength in dBm	Ch 23
2 Data link	802.11 frame and header	Ch 23
2 Data link	MAC address of router	Ch 23
2 Data link	ARP for 192.168.0.1	Ch 23
2 Data link	The switch in the box	Ch 23
3 Network	192.168.0.1 gateway	Ch 24
3 Network	20.207.73.82 target	Ch 24, Ch 25
3 Network	172.31.0.17, 172.26.x	Ch 24
3 Network	137.97.29.249 ISP edge	Ch 24
3 Network	TTL and time exceeded	Routing chapter
3 Network	BGP into ntwk.msn.net	Routing chapter
4 Transport	TCP SYN to port 443	TCP chapter
4 Transport	The 15-second silence	TCP chapter
5 and 6	TLS 1.3, never started	TLS chapter
7 Application	HTTPS and the HTTP GET	HTTP chapter
7 Application	DNS query to 1.1.1.1	Ch 25
7 Application	git pack and sideband	Git chapters

2. Eighteen rows, every one a real observed thing from one evening.

■ 33.4.4 PLAIN – what is really happening inside

Layer 1, physical: the radio in the air.

1. What it is: the level that turns ones and zeros into something physical that travels. Voltage on copper, light in glass, or a radio wave in air.

2. What sits here: the WiFi radio inside the Mac and the matching radio inside the box at 192.168.0.1, working at either 2.4 GHz or 5 GHz.
3. This layer is why walls matter and why a microwave oven can slow a download. Covered in Chapter 23, “Networking From Zero”.

Layer 2, data link: getting across the room.

4. What it is: the level that moves a chunk of data across exactly one hop, to the next box sharing the same wire or the same air.
5. What sits here: the 802.11 frame, the addresses burned into the hardware, and the ARP question that found the router.
6. A MAC address is a six-byte number fixed to a network card. It means something only on the local network and never travels past the router.
7. The Mac knew the router’s IP address, 192.168.0.1, but a radio card cannot send to an IP address. It needs a MAC address.
8. So the Mac shouted to everybody on the local network: “who has 192.168.0.1, tell me your hardware address”. That question is ARP.
9. The router answered, and the Mac then put that hardware address on the front of every frame it sent.
10. The box at 192.168.0.1 is three devices in one shell: a radio access point, a small switch, and a router. The switch part lives entirely at this layer. Covered in Chapter 23.

Layer 3, network: getting across the world.

11. What it is: the level that carries a packet across many hops, through machines that have never heard of you, to any address on earth.
12. What sits here: every single IP address the reader saw.
13. 192.168.0.1 is the router, a private address valid only inside the flat.
14. 20.207.73.82 is what github.com resolved to: a public address in a Microsoft-owned range.
15. 172.31.0.17 at hop 2, and the 172.26.x addresses at hops 4, 5 and 6, are also private, but they belong to the reader’s internet provider, not to the flat. A provider using private addresses in its own core is normal and is covered in Chapter 24.
16. 137.97.29.249 at hop 3 is a public address belonging to the provider.
17. Every IP packet carries a counter called TTL, time to live. Every router it passes reduces the counter by one. At zero the router discards the packet and sends back a complaint.
18. Traceroute abuses that on purpose. It sends a packet with the counter set to 1, then 2, then 3, and collects the complaints. Each one names a router. That is how the reader’s twelve-hop list was made.
19. Routing is the question of which way to send a packet next. Between companies that is answered by BGP, the protocol by which every network tells its neighbours which addresses it can reach.
20. BGP is why hop 7 was the first machine named ntwk.msn.net. At that point the packets left the provider and entered Microsoft’s own network.
21. Covered in Chapter 24 for addresses, and in the routing chapter for TTL, traceroute and BGP.

Layer 4, transport: reaching the right program, reliably.

22. What it is: the level that turns “a machine” into “a program on a machine”, and turns unreliable delivery into a reliable stream.
23. What sits here: TCP, the number 443, and the fifteen seconds of silence.

24. An IP address gets you to a machine. A port number picks which program on it. 443 is the agreed number for secure web traffic.
25. TCP does not just start sending. It performs a three-step greeting.
26. Step one: the Mac sends a small packet marked SYN, meaning “I would like to open a connection”.
27. Step two: the server should reply with SYN and ACK, meaning “yes, and I would like one too”.
28. Step three: the Mac replies ACK, and only then may real data flow.
29. In the reader’s session step one happened, over and over. Step two never happened, not once.
30. curl printed Trying 20.207.73.82:443..., waited fifteen seconds and gave up. That was the Mac patiently repeating step one. Covered in the TCP chapter.

Layers 5 and 6, the awkward ones: TLS.

31. What they are supposed to be: managing a conversation, and agreeing how things are written and encrypted.
32. What sits here: TLS, the thing that puts the S in HTTPS. It sits above TCP and below HTTP, turning a plain reliable stream into a secret, tamper-proof reliable stream.
33. The honest version: TLS does not fit. Its own specification never mentions an OSI layer number. It is not a session layer, because HTTP manages its own sessions. It is not a presentation layer, because it converts no data formats. It is wedged into a gap the model did not plan for.
34. In this session TLS never ran at all. TLS needs a working TCP connection before it can say its first word. Covered in the TLS chapter.

Layer 7, application: the thing you actually wanted.

35. What it is: the protocol that says what the request means.
36. What sits here: three separate application protocols, all involved in one git push.
37. HTTP is the request-and-response language of the web. HTTPS is that same language carried inside TLS.
38. DNS is the naming service. Before anything else, the Mac asked 1.1.1.1 what github.com meant and got back 20.207.73.82.
39. Git has an application protocol of its own, spoken inside HTTP, which sends a pack of objects and receives progress messages back.
40. The error the reader saw twice, send-pack: unexpected disconnect while reading sideband packet, comes from that git protocol. The sideband is the side channel git uses for progress text while a pack transfers.
41. That is a layer 7 error message describing a layer 4 event. Section 33.6 returns to it, because it is the most useful thing in this chapter.
42. Covered in Chapter 25 for DNS, in the HTTP chapter for HTTP and HTTPS, and in the git chapters for the pack protocol.

■ 33.4.5 TECHNICAL - the engineer’s version

Layer 1, physical.

1. Home WiFi uses two delicensed bands: roughly 2.400 to 2.4835 GHz, and roughly 5.15 to 5.85 GHz. Exact permitted sub-bands differ by country and change over time, so check your national regulator rather than a textbook.
2. In the 2.4 GHz band only three 20 MHz channels do not overlap: 1, 6 and 11. In 5 GHz there are far more, bondable into 40, 80 or 160 MHz.

3. 802.11ac uses OFDM with up to 256-QAM, 8 bits per symbol; 802.11ax adds 1024-QAM at 10 bits per symbol. Subcarrier spacing is 312.5 kHz for 802.11a, n and ac.
4. Received signal strength in dBm is the number to look at.

RSSI	Practical meaning
-30 to -50 dBm	Excellent, next to the AP
-50 to -67 dBm	Good, video calls fine
-67 to -70 dBm	Usable, some retries
-70 to -80 dBm	Poor, rate drops hard
below -85 dBm	Effectively unusable

Layer 2, data link.

5. An 802.11 data frame header is 24 bytes: frame control 2, duration 2, three 6-byte address fields, and sequence control 2. A QoS data frame adds a 2-byte QoS control field, giving 26.
6. To carry IP, an 8-byte LLC/SNAP header is inserted, ending in EtherType 0x0800. On Ethernet that EtherType sits in the 14-byte header instead.
7. WPA2 with CCMP adds an 8-byte header and an 8-byte message integrity check. A 4-byte CRC-32 frame check sequence closes every frame.
8. WiFi acknowledges every unicast frame at layer 2 and retransmits if the ACK is missing. Ethernet does not. That is the duplicated-reliability cost from section 33.1, made real.
9. ARP is RFC 826, November 1982, by David Plummer. A request is a broadcast to ff:ff:ff:ff:ff:ff with EtherType 0x0806.
10. Inspect the cache with `arp -a`, which prints lines like ? (192.168.0.1) at 1c:3b:f3:0a:9e:71 on en0 ifscope [ethernet]. An entry reading (incomplete) means the request went unanswered, which is a pure layer 2 fault. Watch it live with `sudo tcpdump -i en0 -n arp`.

Layer 3, network.

11. The IPv4 header is 20 bytes with no options: version and header length 1, DSCP and ECN 1, total length 2, identification 2, flags and fragment offset 2, TTL 1, protocol 1, header checksum 2, source 4, destination 4.
12. Address classification for the session, against RFC 1918, February 1996:

Address	Kind	Whose
192.168.0.1	Private, RFC 1918	The flat
172.31.0.17	Private, RFC 1918	The ISP core
172.26.22.235	Private, RFC 1918	The ISP core
172.16.18.33	Private, RFC 1918	The ISP core
137.97.29.249	Public	The ISP
104.44.196.187	Public	Microsoft
20.207.73.82	Public	Microsoft

13. The 172.16.0.0/12 block covers 172.16.0.0 through 172.31.255.255, so every 172.16, 172.26 and 172.31 address above is private. This is the range people most often misread.
14. TTL is an 8-bit field. macOS sets an initial TTL of 64. Each forwarding router decrements it and recomputes the header checksum, because the header changed.

15. At zero the router discards the packet and returns ICMP type 11 code 0, time exceeded in transit, per RFC 792, September 1981. Traceroute, written by Van Jacobson in 1987, exists entirely to harvest those.
16. macOS traceroute sends UDP probes to high ports by default, three per hop. `traceroute -I` uses ICMP echo, and `traceroute -P TCP -p 443` uses TCP, which is the more informative test when one port is suspected.
17. The reader's hops 5, 6, 8, 9 and 11 each printed more than one address. That is equal-cost multi-path forwarding: three probes took three parallel links. It is a healthy sign, not a fault.
18. From hop 7 every name ends in `ntwk.msn.net`, which is Microsoft network infrastructure. The city codes are airport-style: `del01` Delhi, `bom01` Mumbai, `pnq20` and `pnq21` Pune.
19. The role and interface codes are Microsoft naming conventions, not standards: `ibr` internal backbone router, `rwa` regional wide-area router, `owr` outer wide-area router, and `ae`, `be` and `po` all meaning a bundle of physical links acting as one logical link.
20. Inter-domain routing is BGP-4, RFC 4271, January 2006, descended from RFC 1105 in 1989. Microsoft's best known autonomous system number is 8075. I have not verified which AS announces the exact prefix containing `20.207.73.82`, so treat that attribution as likely rather than proven.

Layer 4, transport.

21. The TCP header is 20 bytes minimum. A macOS SYN typically carries 20 bytes of options, giving a 40-byte header and a 60-byte IP packet with no payload at all. Typical options are maximum segment size, window scale, selective acknowledgement permitted, and timestamps, with NOP bytes for alignment.
22. Port 443 is the IANA-registered port for HTTP over TLS. Port 22 is SSH. Both appear in this session and only one of them worked.
23. The initial retransmission timeout is 1 second per RFC 6298, June 2011, and doubles on each retry. So in fifteen seconds the Mac sends a SYN at roughly $t=0, 1, 3, 7$ and 15 seconds: about five packets, about 300 bytes, and zero bytes received.
24. macOS bounds the whole connect attempt with the `sysctl net.inet.tcp.keepinit`, default 75000 milliseconds, so 75 seconds. `curl` gave up at 15 seconds, well before the kernel would have.
25. What did not happen is the evidence. No TCP RST came back, which would have meant a closed port. No ICMP type 3 came back, which would have meant a device admitting it refused the packet. Nothing came back.
26. Confirm the shape with `sudo tcpdump -i en0 -n 'host 20.207.73.82 and tcp port 443'`. Repeated outbound `Flags [S]` lines with nothing inbound is the exact signature. Test one port quickly with `nc -vz -G 5 20.207.73.82 443`.

Layers 5 and 6, TLS.

27. The TLS record header is 5 bytes: content type 1, legacy version 2, length
 2. Under TLS 1.3 with an AEAD cipher such as AES-128-GCM, each record also carries a 16-byte authentication tag and one byte of real content type inside the encrypted body.
28. A full TLS 1.3 handshake costs one round trip before application data: ClientHello with a key share, then ServerHello, EncryptedExtensions, Certificate, CertificateVerify and Finished, then the client's Finished.
29. The ClientHello carries the server name extension, SNI, containing the text `github.com`. Whether that name is visible on the wire depends on whether Encrypted Client Hello is in use at both ends; check its current standards status before relying on it either way.

30. None of this ran. TLS is carried over TCP and TCP never opened, so the reader's machine never emitted a single TLS byte to 20.207.73.82.
31. Test this layer alone with `openssl s_client -connect github.com:443 -servername github.com`, which separates a TLS fault from a TCP fault by showing how far it got.

Layer 7, application.

32. DNS is RFC 1034 and RFC 1035, November 1987, over UDP and TCP port 53. The reader's resolver was 1.1.1.1, Cloudflare's public resolver, not the router. It answered 20.207.73.82, correctly and fast.
33. HTTP/1.1 is now RFC 9112, June 2022, replacing RFC 7230 from 2014 and RFC 2616 from 1999. HTTP/2 is RFC 9113 and HTTP/3 is RFC 9114.
34. Git over HTTPS uses the smart HTTP protocol: a GET of `/info/refs?service=git-receive-pack` for discovery, then a POST to `/git-receive-pack` carrying the pack.
35. Git frames its own messages in pkt-line format: a four-character hexadecimal length prefix followed by that many bytes, with 0000 as a flush marker. That is layer 7 framing inside HTTP inside TLS inside TCP.
36. The sideband multiplexes three channels in one stream: channel 1 pack data, channel 2 progress text, channel 3 a fatal error. `send-pack: unexpected disconnect while reading sideband packet` means the stream ended while git was waiting for the next pkt-line on that side channel.
37. Observe layer 7 with `dig github.com`, `curl -v`, and `GIT_TRACE_PACKET=1 git push`, which prints every pkt-line git exchanges.

■ 33.4.6 WORDS - remember these

1. MAC address — the hardware name of a card — a 48-bit link-layer address whose top 24 bits are an IEEE-assigned OUI.
2. ARP — the shout that finds a router's hardware address — Address Resolution Protocol, RFC 826, EtherType 0x0806, broadcast request, unicast reply.
3. TTL — the counter that stops packets circling forever — an 8-bit IPv4 field decremented per hop, triggering ICMP type 11 at zero.
4. BGP — how networks tell each other what they can reach — Border Gateway Protocol version 4, RFC 4271, the inter-domain routing protocol.
5. SYN — the first word of a TCP conversation — the synchronize flag opening the three-way handshake, retransmitted with exponential backoff.
6. SNI — the site name sent before encryption starts — Server Name Indication, the TLS extension letting one address serve many certificates.
7. pkt-line — how git chops its own stream into messages — a four-hex-digit length prefix plus payload, with 0000 as a flush packet.
8. Sideband — the channel carrying progress text — git's multiplexing of pack data, progress and errors over one connection.

33.5 Encapsulation walked through with real numbers

■ 33.5.1 PLAIN – in simple words

1. Encapsulation means putting a thing inside a bigger thing that carries it. Every layer does it, adding a small label at the front of what it was given.
2. So the message you cared about ends up buried in the middle of a much bigger bundle of bytes.
3. This costs you. The labels are real bytes, sent over the real air, taking real time, and you are paying for them.
4. The answer, for a small request over WiFi, is that roughly four bytes in every ten are labels rather than message.
5. For large transfers the picture is much better, because the labels stay the same size while the message gets bigger.

■ 33.5.2 PLAIN – a picture in your head

1. Think of wooden nesting dolls, the kind where each one opens with a smaller one inside.
2. The smallest doll is a folded note: the thing you actually wanted to send.
3. Around it goes a doll marked “secret”, which locks. That is TLS.
4. Around that, a doll marked “piece 4 of 9”. That is TCP.
5. Around it goes a doll marked “secret”, which locks. That is TLS.
6. Around that, a doll with the far building’s street address. That is IP.
7. Around that, a doll naming the courier taking it to the end of the road. That is WiFi. You post the outermost doll.

Where this comparison breaks:

8. Real dolls get much thicker at each level. Network headers do not. The IP header is 20 bytes whether it wraps 1 byte or 1,460.
9. And a real doll survives the journey. The outermost network shell is stripped and rebuilt at every router, with different labels every time.

■ 33.5.3 PLAIN – a worked example

1. Here is one real, complete HTTP request, of the kind git sends when it starts talking to GitHub.

```
GET /kedbyte/notes.git/info/refs?service=git-upload-pack HTTP/1.1
Host: github.com
User-Agent: git/2.39.5 (Apple Git-154)
Accept: */*
Accept-Encoding: deflate, gzip
Pragma: no-cache
```

2. Counting every character, including the carriage return and line feed ending each line and the blank line that closes the headers, that is exactly 190 bytes.
3. Now watch it grow on the way down, one layer at a time.

Step	Adds	Running total
HTTP request text	-	190 bytes
TLS 1.3 record	22 bytes	212 bytes
TCP segment	32 bytes	244 bytes
IPv4 packet	20 bytes	264 bytes
802.11 frame	54 bytes	318 bytes

4. Final answer: 318 bytes go over the air to carry 190 bytes of request.

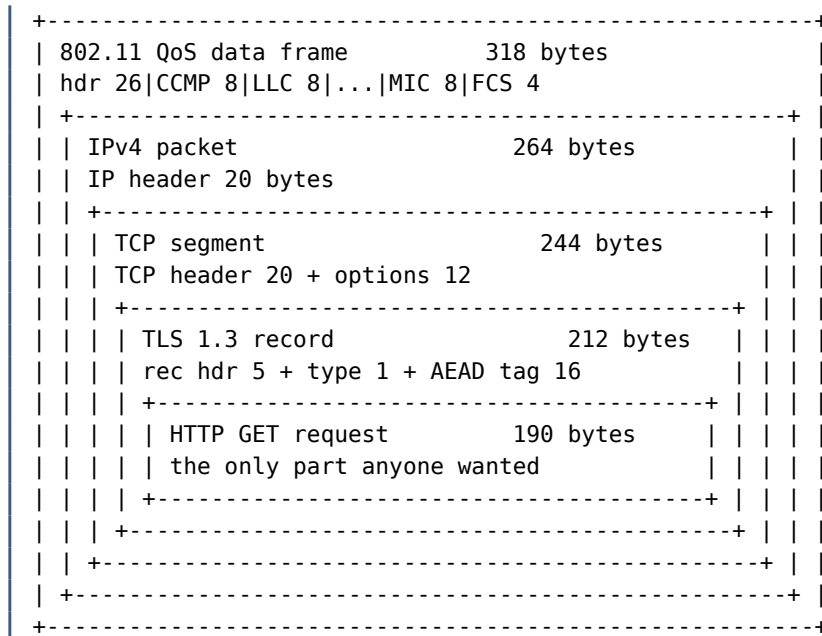
5. That is 128 bytes of overhead, 40.3 percent of the frame. The payload is 59.7 percent.

■ 33.5.4 PLAIN – what is really happening inside

1. Nothing is copied five times. The operating system allocates one buffer and deliberately leaves empty space at the front, called headroom.
2. Each layer writes its header backwards into that space and moves a pointer. The message bytes never move.
3. On macOS and the BSD family that buffer is an mbuf. On Linux it is an sk_buff. Both reserve headroom for exactly this reason.
4. This is why “the data travels down through the layers” is a teaching lie. Section 33.98 says so plainly.
5. The one place bytes really are copied is the boundary between the kernel and the network card, and modern cards use DMA to avoid even that.

■ 33.5.5 TECHNICAL – the engineer’s version

1. The full nesting, drawn to scale of description rather than of size.



2. The 128 bytes of overhead, itemized so you can check every number.

Layer	Fields	Bytes
TLS 1.3	hdr 5, type 1, tag 16	22
TCP	header 20, options 12	32
IPv4	header, no options	20
802.11	QoS hdr 26, FCS 4	30
WPA2 CCMP	header 8, MIC 8	16
LLC/SNAP	for EtherType 0x0800	8

3. On wired Ethernet the layer 2 cost would be 18 bytes rather than 54: a 14-byte header and a 4-byte FCS. The same request would be 282 bytes on the wire and 67.4 percent payload.
4. Layer 1 on WiFi is not measured in bytes at all. It is measured in microseconds, and that is where the true overhead hides.

5. Approximate airtime for that single 318-byte frame on 5 GHz, 802.11ac, one spatial stream, 80 MHz, at a physical rate of 433 Mbit/s:

Component	Approximate time
DIFS wait	34 microseconds
Average backoff	67 microseconds
VHT preamble	40 microseconds
The 318 bytes	5.9 microseconds
SIFS plus ACK	60 microseconds
Total	about 207 microseconds

6. Read that again. The data itself occupies under 3 percent of the airtime. Everything else is protocol.
7. Efficiency improves enormously with size.

Payload	Frame on WiFi	Payload share
190 bytes	318 bytes	59.7 percent
536 bytes	664 bytes	80.7 percent
1,398 bytes	1,526 bytes	91.6 percent

8. 1,398 is not arbitrary. From a 1,500-byte MTU, subtract 20 for IP and 32 for TCP with timestamps to get an MSS of 1,448, then subtract 22 for the TLS record, leaving 1,426 bytes of application data. Round down further for a VPN or PPPoE header, which is why 1,398 and 1,400 are common settings.
9. Read real header sizes off the wire with `tcpdump -i en0 -v -X`, or open a capture in Wireshark, where each layer appears as an expandable tree in exactly the nesting shown above.

■ 33.5.6 WORDS - remember these

1. Encapsulation — putting a message inside a bigger envelope — wrapping a layer N PDU as the payload of a layer N-1 PDU.
2. Overhead — the bytes that are not your message — header and trailer bytes as a fraction of the transmitted unit.
3. Headroom — the empty space in front of the message — reserved bytes at the start of a kernel packet buffer so headers can be prepended without copying.
4. MTU — the biggest frame a link will carry — maximum transmission unit, 1,500 bytes for standard Ethernet.
5. MSS — the biggest chunk of data one TCP segment carries — maximum segment size, the MTU minus the IP and TCP headers.
6. FCS — the check at the end of a frame — frame check sequence, a 32-bit CRC letting the receiver discard corrupted frames.

33.6 Where the reader's failure sits in the model

■ 33.6.1 PLAIN - in simple words

1. The question is not “what is wrong”. It is “which layer is wrong”, because that is a question evidence can answer.

2. Layer 1 worked. The laptop was on the WiFi, frames were going out and coming back. If the radio were dead nothing at all would have happened.
3. Layer 3 worked, and this is the strong evidence. The traceroute reached twelve routers, the last inside Microsoft's network in Pune.
4. Think about what that proves. Packets from the flat crossed the provider's network, entered Microsoft's, travelled Delhi to Mumbai to Pune, and complaints from those routers came all the way back.
5. Layer 4 did not complete. The laptop said hello to port 443 about five times over fifteen seconds and was never answered.
6. Layers 5, 6 and 7 were never attempted. TLS cannot start without TCP, HTTP cannot start without TLS, git cannot start without HTTP.
7. So the fault is at layer 4, or in something that only affects layer 4 traffic to that address.
8. That is a far smaller question than "the internet is broken", and it was reached in about two minutes of evidence gathering.

■ 33.6.2 PLAIN – a picture in your head

1. Imagine a lift in a tall building that is not arriving. You could stand there guessing, or you could check floors.
2. On floor 4 you press and nothing happens. No light, no sound, no error.
3. You now know something precise: the fault is at or just below floor 4, and floors 5, 6 and 7 have not been tested and cannot be, because you cannot get to them.

Where this comparison breaks:

4. In a building, a fault on floor 4 is on floor 4. On a network, the thing that broke layer 4 may sit at layer 3, in a router that drops only packets with certain port numbers.
5. So "the fault is at layer 4" means "the symptom is at layer 4". The cause may be a device operating below it that inspects above itself. Section 33.8 is about exactly that dishonesty.

■ 33.6.3 PLAIN – a worked example

1. The evidence table. Left, the layer. Middle, what was observed. Right, what that proves.

Layer	Observation	Verdict
1 Physical	Associated, traffic flows	Works
2 Data link	Router reachable, ARP fine	Works
3 Network	Traceroute reached hop 12	Works
4 Transport	5 SYNs, 0 replies, 15 s	Fails
5 and 6 TLS	No ClientHello ever sent	Untested
7 Application	DNS answered correctly	Partly works
7 Application	HTTP never sent	Untested

2. That is not a contradiction. DNS to 1.1.1.1 is a different destination on UDP port 53, not TCP port 443. It took a different path with a different transport, and it succeeded.
3. The most useful thing was the mobile data test: the same request over a different provider's network succeeded instantly.
4. Change one variable, hold the rest. The variable that changed was the path from layer 1 to layer 3. The failure followed the path, not the machine and not the server.

■ 33.6.4 PLAIN – what is really happening inside

1. Find the highest layer that demonstrably works, and the lowest layer that demonstrably fails. The fault is between them.
2. Work from the bottom up, because a low layer failing makes every test above it meaningless.
3. There is no point debugging a certificate error if ARP is failing. The certificate error is a lie told by a program that could not reach anything.
4. The traceroute proves that packets reached hop 12 and replies returned. It proves nothing about hops 13 onward, because routers commonly refuse to send the replies traceroute needs.
5. Silence at the end of a traceroute is normal and is not evidence of a fault. Saying otherwise is the most common mistake in network diagnosis.
6. What settled the reader's case was a TCP connection attempt that got no response of any kind, combined with the same attempt succeeding over a different network.
7. That combination supports one conclusion: something on that particular path silently discarded the packets. It does not identify who, and honest diagnosis stops there.

■ 33.6.5 TECHNICAL – the engineer's version

1. The outcomes of a TCP connection attempt, and what each proves, are the sharpest diagnostic in networking.

Response	Meaning	Layer implicated
SYN-ACK	Port open, path fine	None, it works
RST	Reached host, port shut	7, service not running
ICMP type 3	A device refused it	3, policy in the path
Nothing at all	Silently discarded	3 or 4, on the path

2. The reader got the fourth row. A RST would have meant packets arrived and something answered. An ICMP unreachable would have meant a device admitted responsibility. Silence means neither.
3. The sequence of commands that produces this evidence, in order:

```
ifconfig en0 | grep -E 'status|inet '
arp -a | grep 192.168.0.1
ping -c 3 192.168.0.1
traceroute -n 20.207.73.82
traceroute -P TCP -p 443 -n 20.207.73.82
nc -vz -G 5 20.207.73.82 443
nc -vz -G 5 20.207.73.82 22
dig +short github.com @1.1.1.1
curl -v --max-time 15 https://github.com
```

4. Line 5 is the one most people omit and the most informative: a TCP traceroute to port 443 shows how far port 443 traffic specifically gets, which a UDP or ICMP traceroute cannot tell you.
5. Compare lines 6 and 7. In the reader's session one transport worked and the other did not, localizing the problem to something treating port 443 differently from port 22.
6. Note what that does not prove. It does not prove who is doing the treating, nor that it is deliberate. Consistent with the evidence: a filter, a misrouted prefix, an overloaded state table, or an asymmetric return path. Not consistent: the server being down, since the same address served the same content over mobile data.
7. The git error from the same session, `send-pack: unexpected disconnect while reading sideband packet`, is the same lesson in a different costume: a layer 7 message whose real cause was a layer 4 connection dying mid-stream.

- That is why the error tells you nothing about whether the push landed. The disconnect happened while reading the response. The write may already have been applied on the server. The only correct action is to re-query the remote state, never to assume from the error text.

■ 33.6.6 WORDS – remember these

- Bottom-up diagnosis — check the ground floor first — testing layers in ascending order so no test is invalidated by a lower failure.
- Silent drop — the packet vanishes with no complaint — a discard with no RST and no ICMP error, giving the sender no information at all.
- RST — a slammed door — the TCP reset flag, meaning the host was reached and refused the connection.
- ICMP unreachable — a written refusal — ICMP type 3, with codes for host, network, port and administrative prohibition.
- Localizing — narrowing where the fault is — changing exactly one variable at a time until the failure follows the variable.

33.7 A fault-to-layer table

■ 33.7.1 PLAIN – in simple words

- If you learn the shape of each failure, the error message tells you where to look before you have run a single command.
- The table in this section is the one to pin above your desk. Read it in either direction: from a symptom find the layer, or from a layer find the command that tests it.

■ 33.7.2 PLAIN – a picture in your head

- Think of a doctor with a list: fever means one family of causes, a rash another, a limp a third.

Where this comparison breaks:

- A body has one owner. A network path crosses many, and the layer showing the symptom is often not the layer that owns the cause.

■ 33.7.3 PLAIN – a worked example

- Symptoms and the tool that settles each one.

Layer	What you see	Tool
1 Physical	No link, weak signal	ifconfig, RSSI
2 Data link	ARP incomplete	arp -a, tcpdump arp
3 Network local	Gateway unreachable	ping 192.168.0.1
3 Network remote	Trace stops, loops	tracert -n
4 Transport	Timeout or refused	nc -vz, tcpdump
5/6 TLS	Cert or version error	openssl s_client
7 DNS	Name does not resolve	dig, dig @1.1.1.1
7 HTTP	403, 404, 500	curl -v -i
7 App auth	Permission refused	the service API

- The same nine rows, each with a real example.

Layer	Real example of a failure
1 Physical	Wi-Fi at -88 dBm, rate collapses
2 Data link	arp -a prints (incomplete)
3 Network local	ping to 192.168.0.1 times out
3 Network remote	trace dies before hop 7
4 Transport	15 s of silence on port 443
5/6 TLS	Certificate expired yesterday
7 DNS	NXDOMAIN for a mistyped name
7 HTTP	403 on a push, token scope
7 App auth	Token lacks workflow scope

3. The last row is from the reader's own session. A push was refused because the personal access token did not carry the workflow scope and the push changed a CI workflow file.
4. That failure is as high in the stack as a failure can be. Every layer below it worked perfectly. The bytes arrived, were decrypted, were understood, and were then declined on policy grounds.

■ 33.7.4 PLAIN – what is really happening inside

1. Two rules make the table usable. First, an error message names the layer that noticed, not the layer that failed. Git's sideband error was written at layer 7 about a layer 4 event.
2. Second, the higher the layer, the more specific the error text, and the more likely it is to be honest about itself.
3. A 403 from an API tells you precisely what is wrong. A timeout tells you almost nothing, because a timeout is what you get when nobody spoke.
4. So an ugly, detailed error message is good news. Silence is bad news, because the failure is low and nobody on the path will explain themselves.

■ 33.7.5 TECHNICAL – the engineer's version

1. Exact commands, roughly one per layer, on macOS.

```
# 1 physical / 2 link
ifconfig en0 | grep -E 'status|ether|inet '
system_profiler SPAirPortDataType | grep -A4 'Current Net'
# 2 link
arp -a -n | grep 192.168.0.1
sudo tcpdump -i en0 -n arp
# 3 network
netstat -rn -f inet | head
ping -c 3 -W 1000 192.168.0.1
traceroute -n -w 1 -q 3 20.207.73.82
# 4 transport
nc -vz -G 5 20.207.73.82 443
sudo tcpdump -i en0 -n 'host 20.207.73.82 and tcp'
# 5/6 TLS
openssl s_client -connect github.com:443 -servername github.com
# 7 application
dig +short github.com @1.1.1.1
curl -v -i --max-time 15 https://github.com
GIT_TRACE_PACKET=1 GIT_CURL_VERBOSE=1 git push
```

2. Interpretation notes that are easy to get wrong.
3. ping failing does not prove layer 3 is broken. Many networks and hosts drop ICMP echo by policy. Use it as positive evidence only.

4. `nc -vz` reporting “Connection refused” is a success for layers 1 to 4. It means a RST came back, so packets reach the host in both directions.
5. `openssl s_client` printing `verify error:num=20` means it could not build a chain to a trusted root, usually a missing intermediate certificate on the server, not an expired one. Read the number.
6. `curl` exit codes are layered too, which makes them a shortcut: 6 is DNS failure, 7 is connect failure, 28 is timeout, 35 is a TLS handshake error, 60 is certificate verification failure.

■ 33.7.6 WORDS – remember these

1. Symptom — what you see — the observable behaviour, belonging to the layer that reported it, not necessarily the layer at fault.
2. NXDOMAIN — the name does not exist — the DNS response code for a queried name with no records at all.
3. Verify error 20 — the chain could not be built — OpenSSL’s code for unable to get local issuer certificate, usually a missing intermediate.
4. Connection refused — a host said no — an RST in response to a SYN, which proves bidirectional reachability.
5. Scope — what a token is permitted to do — the permission set attached to a credential, checked at the application layer after everything else worked.

33.8 Where the model lies

■ 33.8.1 PLAIN – in simple words

1. The layered model is a map. The internet is the territory. The map is wrong in several specific, well-known places.
2. Five places matter, and every working engineer meets all five.
3. First, TLS does not fit anywhere. It sits between transport and application and belongs to neither.
4. Second, a VPN carries whole layer 3 packets inside a layer 7 connection, so the layers stop being an order and become a loop.
5. Third, MPLS is so awkwardly placed that the industry gave up and invented the number 2.5 for it.
6. Fourth, NAT reaches into two layers at once and rewrites both, which the model says is forbidden.
7. Fifth, QUIC moved the transport layer out of the operating system and into ordinary programs.

■ 33.8.2 PLAIN – a picture in your head

1. Think of a road map of a city printed in 1984.
2. But a tunnel has been dug that the map does not show, a one-way street has been reversed, and two roads have been merged into a flyover.

Where this comparison breaks:

3. A map can be reprinted. The OSI model cannot, because its value is that everybody already agreed on the numbers. Fixing it would destroy the only thing it is good for.

■ 33.8.3 PLAIN – a worked example

1. The reader’s own Mac showed several `utun` interfaces. Those are tunnel interfaces, and they are the clearest example of the model folding in on itself.
2. The VPN software takes that whole packet, encrypts it, and sends it as the payload of another connection, which has its own layer 3 and 4 headers.

3. So the order, reading outward, is: application, TLS, TCP, IP, and inside that, IP again, TCP again, TLS again, application again.
4. Layer 3 is now above layer 7. The model has no vocabulary for that.

■ 33.8.4 PLAIN – what is really happening inside

1. The honest version: layers are not a physical fact about packets. They are a naming scheme for functions.
2. A packet does not know it is layer 3. It is a sequence of bytes that some piece of code chooses to read as an IP header.
3. Once you see that, tunnelling stops being strange. It is bytes carried by bytes.
4. And middleboxes stop being strange too. Nothing physically prevents a router from reading a TCP header, or a firewall from reading an HTTP path. The model says they should not. It cannot stop them.

■ 33.8.5 TECHNICAL – the engineer’s version

1. The five violations, precisely.
2. **TLS.** RFC 8446, August 2018, never assigns TLS an OSI layer number. TLS requires a reliable ordered byte stream below it and presents one above it, which makes it a transport-layer shim, but it performs no multiplexing and no addressing, which are the defining transport functions. Industry usage is inconsistent: load-balancer vendors say “layer 7”, security vendors say “layer 6”, researchers write “layer 4.5”. None is normative.
3. **VPNs.** WireGuard, released by Jason Donenfeld in 2016 and merged into Linux 5.6 in March 2020, carries IP inside UDP inside IP: layer 3 over 4 over 3. OpenVPN, from James Yonan in 2001, can carry IP inside TLS inside TCP. IPsec, RFC 4301 from December 2005, carries IP inside IP. VXLAN, RFC 7348 from August 2014, carries Ethernet frames inside UDP: layer 2 over 4.
4. **MPLS.** RFC 3031, January 2001. The shim header is 4 bytes: a 20-bit label, 3 traffic class bits, 1 bottom-of-stack bit and an 8-bit TTL. It sits between the layer 2 and layer 3 headers, and forwarding uses the label instead of the IP address. It is neither layer 2 nor layer 3, so the industry calls it layer 2.5. That number appears in no standard.
5. **NAT.** RFC 3022, January 2001, updating RFC 1631 from May 1994. A NAT rewrites the layer 3 source address and the layer 4 source port together, then recomputes both the IP header checksum and the TCP or UDP checksum, because the transport checksum covers a pseudo-header containing the IP addresses. One device edits two layers in a single operation, and must understand the layer above it to do so at all.
6. The reader’s own traffic was translated at least twice: once by the box at 192.168.0.1, and again inside the provider’s private core at the 172.31 and 172.26 addresses, which is carrier-grade NAT.
7. **QUIC.** RFC 9000, May 2021, with HTTP/3 in RFC 9114, June 2022. QUIC implements connection setup, reliability, ordering, flow control and congestion control, all transport functions, in user space over UDP.
8. Three consequences follow. Transport logic now ships and updates on the browser’s schedule instead of the operating system’s. Most of the transport header is encrypted, so middleboxes cannot read or modify it. And streams are independent, so one lost packet no longer stalls the others, which was TCP’s head-of-line blocking problem.
9. Two more untidy cases. ARP has no clean layer: it uses its own EtherType rather than riding on IP, yet exists solely to serve IP. ICMP is carried inside IP yet is universally treated as part of layer 3.
10. Summary of the lies, with the honest placement.

Thing	Model says	Reality
TLS	Layer 5 or 6	Between 4 and 7
VPN tunnel	Strict ordering	Layer 3 inside layer 7
MPLS	No such layer	Called layer 2.5
NAT	One layer each	Edits 3 and 4 together
QUIC	Transport in kernel	Transport in the app

- Keep the model anyway. Nobody has proposed a replacement that is both more accurate and as easy to talk about, and the numbers are how engineers communicate during incidents.

■ 33.8.6 WORDS - remember these

- Tunnel — carrying a whole packet inside another packet — encapsulating a layer N PDU inside a protocol at or above layer N.
- Middlebox — a device in the path that reads above its own layer — NATs, firewalls, proxies and load balancers that inspect layer 4 or 7.
- MPLS — forwarding by label instead of address — Multiprotocol Label Switching, RFC 3031, a 4-byte shim between layers 2 and 3.
- TCP meltdown — two reliability timers fighting — throughput collapse from running TCP inside a TCP tunnel.
- Head-of-line blocking — one lost piece stalling everything behind it — the TCP property that QUIC's independent streams remove.
- Pseudo-header — why NAT must fix two checksums — the IP address fields included in the TCP and UDP checksum computation.

33.9 Why layering still earns its keep

■ 33.9.1 PLAIN - in simple words

- After a section listing what is wrong with the model, the fair question is whether it is worth keeping. It is, for three reasons. First, modularity: each piece can be built, tested and replaced on its own.
- Second, independent innovation. Anyone can improve one layer without asking permission from the layers above or below.
- Third, reasoning about faults. The whole of section 33.6 was only possible because the parts had names and an order.
- The second reason is the one that built the internet. Nobody had to coordinate. A radio company could invent faster WiFi while a browser company invented a faster web, and neither had to speak to the other.

■ 33.9.2 PLAIN - a picture in your head

- Think of the standard shipping container, introduced in the 1950s.
- The container is the agreed shape in the middle. Everything above and below it changed completely over seventy years. IP is that container.

Where this comparison breaks:

- A container is a physical object with a fixed size. An IP packet is a convention that anyone can, and does, quietly violate with tunnels.

■ 33.9.3 PLAIN – a worked example

1. Three real cases where one layer changed enormously and nothing above it needed to know.
2. Case one, layers 1 and 2. Ethernet ran at 10 Mbit/s in 1980 and now runs at 400 Gbit/s. That is 40,000 times faster.
3. The frame format did not change: six bytes of destination address, six of source, two of type. An IP stack written for 10 Mbit/s Ethernet describes the same frame that crosses a modern data centre.
4. Case two, layers 1 and 2 again, at home. The reader's household connection would have been a dial-up modem in the 1990s, then ADSL over a phone line, then fibre, and on a phone it is 4G or 5G radio.
5. Case three, the awkward TLS layer. TLS 1.3 was published as RFC 8446 in August 2018 and replaced almost everything about the handshake: two round trips became one, old ciphers were deleted, the negotiation was redesigned.
6. Web applications did not change. Servers and browsers were upgraded, and the HTTP requests flowing over the top were byte-for-byte identical.

■ 33.9.4 PLAIN – what is really happening inside

1. The mechanism behind all three cases is a stable interface. As long as the service offered upward keeps its shape, the implementation beneath can be replaced entirely.
2. This is the same reason a program written in 1990 still runs on a modern processor: the instruction set is a stable interface, and everything under it was rebuilt many times.
3. It is also why badly drawn boundaries cause so much trouble. NAT broke the assumption that an address is stable end to end, and the internet is still paying for it thirty years later.

■ 33.9.5 TECHNICAL – the engineer's version

1. Concrete figures for the three cases.

Change	Layer	Nothing above changed
10 Mbit/s to 400 Gbit/s	1 and 2	Same Ethernet frame
Dial-up to fibre to 5G	1 and 2	Same IP, same HTTP
TLS 1.2 to TLS 1.3	The shim	Same HTTP bytes

2. Ethernet's frame format is essentially unchanged since the DIX Ethernet version 2 specification of 1982 and IEEE 802.3, approved in 1983. Later additions such as the 4-byte 802.1Q VLAN tag from 1998 are insertions, not redesigns.
3. Wi-Fi shows the same pattern at layer 1: 802.11 in 1997 at 2 Mbit/s, 802.11b in 1999 at 11 Mbit/s, 802.11g in 2003 at 54 Mbit/s, 802.11n in 2009, 802.11ac in 2013, 802.11ax as Wi-Fi 6 in 2021, and Wi-Fi 7 certification from January 2024. The MAC frame the reader's laptop sends is recognizably the 1997 frame.
4. The counterexample proves the rule. IPv6, specified in 1998 and now RFC 8200 from July 2017, changed the layer everything else depends on, and adoption has taken three decades. The reader's machine reported IPv6: (none).

■ 33.9.6 WORDS – remember these

1. Modularity — building in replaceable pieces — decomposition into units with stable interfaces and hidden implementations.
2. Stable interface — the promise that does not change — the contract a layer offers upward, permitting its implementation to be replaced.

3. Narrow waist — the one thing everybody must agree on — the single universal convergence protocol, IP, in the internet architecture.
4. Ossification — the network becoming hard to change — the effect of middleboxes depending on details protocols never promised to keep.

33.10 The same idea outside networking

■ 33.10.1 PLAIN – in simple words

1. Layering is not a networking idea. It is an engineering idea that networking happens to describe most carefully.
2. In graphics, a game does not talk to a graphics chip. It talks to a library, which talks to a driver, which talks to the chip.
3. In storage, a program does not talk to flash memory. It writes a file, which a filesystem turns into numbered blocks, which a driver turns into commands, which the drive turns into operations on cells.
4. In compilers, your source becomes tokens, then a tree, then a neutral intermediate form, then machine instructions, then something smaller still inside the processor.

■ 33.10.2 PLAIN – a picture in your head

1. Think of a restaurant. You read a menu, not a recipe. The cook reads a recipe, not a farm. The farm grows food without knowing any menu.

Where this comparison breaks:

2. In a restaurant a customer can walk into the kitchen. In a well-built software stack, upper levels usually cannot reach past the level below, and when they can, that shortcut becomes a bug that lasts for years.

■ 33.10.3 PLAIN – a worked example

1. The graphics stack, top to bottom, with the reader's Mac in mind.

Your program	draw this triangle
Graphics API	Metal, Vulkan, OpenGL calls
Driver	translates to GPU commands
Command buffer	a queue in shared memory
GPU hardware	thousands of small cores

2. The storage stack has the same shape.

Your program	write("notes.txt", data)
Filesystem	APFS or ext4: file -> blocks
Block layer	queue, scheduling, caching
Driver	NVMe or SATA commands
Flash translation	pretends to be numbered sectors
NAND cells	charge trapped in floating gates

3. The flash translation layer is the interesting one. It presents a lie upward: it pretends the drive is numbered sectors that can be overwritten in place. NAND flash cannot do that at all.
4. That is exactly what NAT does at layer 3, and it causes the same class of surprises: performance that does not match the model, and behaviour that only makes sense once you know about the hidden layer.

■ 33.10.4 PLAIN – what is really happening inside

1. The compiler stack has the same shape. Source text becomes tokens, tokens become a tree, and the tree becomes an intermediate representation that belongs to no particular processor.
2. The value of the neutral middle form is the value of IP: many languages above it, many processors below it, one thing in the middle.

■ 33.10.5 TECHNICAL – the engineer’s version

1. Stable interfaces acting as narrow waists, outside networking.

Domain	The narrow waist	Since
Graphics	OpenGL, then Vulkan	1992, 2016
Compilers	LLVM IR	2003
Processors	The x86-64 instruction set	2000
Storage	The block device interface	1970s
Operating systems	The POSIX system calls	1988

2. LLVM began as Chris Lattner’s work at the University of Illinois around 2000 to 2003, and its intermediate representation is now the middle layer for Swift, Rust, Clang and many others.
3. Direct3D appeared in 1995, Metal in 2014, Vulkan 1.0 in February 2016. All are the same architectural move: a stable command interface above a driver that changes with every hardware generation.
4. NVMe 1.0, published in March 2011, replaced the disk-shaped assumptions of older interfaces with a queue-based one while keeping the block abstraction above it unchanged. Filesystems did not need rewriting.
5. The instruction set architecture is the layer boundary that matters most in a processor: the same binary runs on chips from different companies with entirely different internal designs, which is precisely the layering promise.

■ 33.10.6 WORDS – remember these

1. Abstraction layer — a level that hides the mess below — a module presenting a simplified interface over a more complex implementation.
2. Intermediate representation — the neutral middle form in a compiler — a machine-independent program form such as LLVM IR.
3. Flash translation layer — the part that makes flash look like a disk — the controller firmware mapping logical blocks to physical NAND pages.
4. Instruction set architecture — the contract between software and a chip — the documented instructions and behaviour a processor must implement.
5. Leaky abstraction — when the mess below shows through — an interface whose performance or failure behaviour reveals its implementation.

33.98 Common wrong ideas

1. Wrong: data physically travels down through the layers. Right: nothing moves. The message sits in one buffer with reserved space at the front, and each layer writes its header into that space and moves a pointer. On a Mac that buffer is an `mbuf`, on Linux an `sk_buff`.
2. Wrong: TCP/IP has seven layers. Right: the internet architecture stated in RFC 1122, October 1989, has four: link, internet, transport and application. Seven is OSI, a different model from a different organization.

3. Wrong: the OSI model is what the internet uses. Right: the internet uses protocols designed before OSI existed. IP was RFC 791 in September 1981; ISO 7498 appeared in 1984. OSI survives as vocabulary and as a teaching frame, not as the running system.
4. Wrong: layer 8 is a real layer, for the user. Right: there is no layer 8 in ISO 7498 or any standard. It is a joke that became a habit, and using it in a serious document marks you as guessing.
5. Wrong: TLS is layer 6. Right: no standard says so. RFC 8446 assigns no layer number. TLS sits between transport and application and is called 5, 6 or 4.5 by different vendors, all informally.
6. Wrong: the traceroute stopped, so that is where the fault is. Right: routers commonly refuse to send the ICMP time-exceeded replies traceroute needs. Stars at the end of a trace are normal. The reader's trace ended in stars after hop 12 and that fact alone proved nothing.
7. Wrong: a router forwards frames. Right: a router forwards packets. It discards the incoming layer 2 frame entirely and builds a new one for the next hop with different addresses. Only the layer 3 header survives end to end, minus the TTL.
8. Wrong: switches work at layer 3 because they have IP addresses. Right: the management address on a switch is for talking to the switch. Its forwarding decisions use MAC addresses, which is layer 2. A device that forwards on IP addresses is a router, whatever the box is labelled.
9. Wrong: an error message tells you which layer failed. Right: it tells you which layer noticed. Git's `send-pack: unexpected disconnect while reading sideband packet` is a layer 7 message describing a layer 4 event.
10. Wrong: if the model and the network disagree, the network is doing it wrong. Right: the model is a map. NAT, VPNs, MPLS and QUIC all break it, they are all deployed at enormous scale, and they are not going away.

33.99 Chapter summary in 20 lines

1. Layering splits network work into levels, where each level uses the level below without knowing how it works and serves the level above.
2. The benefit is that a whole level can be replaced and nothing above notices: radio to copper, dial-up to fibre, TLS 1.2 to TLS 1.3.
3. The cost is duplicated work, hidden information, header overhead, and faults that cross layers and belong to nobody. RFC 3439 from December 2002 says so in a section titled "Layering Considered Harmful".
4. OSI has seven layers: physical, data link, network, transport, session, presentation, application, handling bits, frames, packets, segments and data.
5. It came from ISO subcommittee 16 starting in 1977, was described by Hubert Zimmermann in 1980, and was published as ISO 7498 in 1984, also ITU-T X.200.
6. The OSI protocol suite lost to TCP/IP. GOSIP made it a US federal purchasing requirement in August 1990 and the mandate was relaxed in 1995.
7. Pieces of OSI did survive and you use them daily: X.509 certificates, ASN.1 encoding, IS-IS routing in carrier backbones, and LDAP descended from X.500.
8. Layers 5 and 6 barely map onto anything real. No mainstream internet protocol implements a separate session or presentation entity.
9. The TCP/IP model has four layers: link, internet, transport, application, stated normatively in RFC 1122, October 1989.
10. Its shape is an hourglass: many link technologies below, many applications above, and exactly one IP in the middle.
11. In the reader's session, layer 1 was the 5 GHz WiFi radio, layer 2 was the 802.11 frame and the ARP lookup for 192.168.0.1, and layer 3 was every IP address from 192.168.0.1 to 20.207.73.82,

including the private ISP hops 172.31.0.17 and 172.26.x.

12. Layer 4 was TCP to port 443, layers 5 and 6 were TLS, and layer 7 was DNS to 1.1.1.1, HTTP, and git's own pack and sideband protocol.
13. Encapsulating one real 190-byte HTTP GET produced a 318-byte WiFi frame: 22 bytes of TLS, 32 of TCP, 20 of IP and 54 of 802.11, so 40.3 percent overhead and 59.7 percent payload.
14. At layer 1 the picture is worse. That frame occupies about 6 microseconds of a roughly 207-microsecond airtime slot, under 3 percent.
15. The reader's failure sits precisely at layer 4. Layers 1, 2 and 3 reached hop 12 inside Microsoft's network in Pune, layer 4 never completed its handshake, and layers 5 to 7 were never attempted.
16. The general rule: find the highest layer that demonstrably works and the lowest that demonstrably fails, and the fault is between them.
17. Each layer has a signature failure and a tool that tests it: `ifconfig`, `arp -a`, `ping`, `tracert`, `nc -vz`, `openssl s_client`, `dig`, `curl`.
18. The model lies in five well-known places: TLS fits nowhere, VPNs put layer 3 inside layer 7, MPLS is called layer 2.5, NAT rewrites layers 3 and 4 together, and QUIC moved transport into user space over UDP.
19. Keep the model anyway. It is how engineers talk to each other during incidents, and no clearer replacement has been proposed.
20. The same pattern runs through graphics, storage and compilers, so learning it once pays for itself far outside networking.

Firewalls, VPNs, CDNs, IPv6 and the Gaps

34.0 What this chapter gives you

1. You will be able to say exactly what `localhost`, `127.0.0.1` and `0.0.0.0` each mean, and why `0.0.0.0` means two different things depending on where you write it.
2. You will be able to explain the three port ranges, why ports below 1024 need root on a UNIX machine, and why a busy machine can run out of ports even though it has 65,535 of them.
3. You will be able to describe what a firewall is, what it is not, and the difference between a stateless filter and a stateful one, with real rules written in `pf`, `iptables`, `nftables` and a cloud security group.
4. You will be able to explain the difference between `DROP` and `REJECT` well enough to diagnose a fault from the shape of the failure alone, and you will recognize the reader's own 15-second silence as the `DROP` signature.
5. You will be able to name the middleboxes sitting between you and any server, and say what each one can and cannot see.
6. You will be able to explain what a VPN really does at the packet level, choose between IPsec, OpenVPN and WireGuard for a reason, and state plainly what a commercial VPN does not protect you from.
7. You will be able to tell a forward proxy from a reverse proxy for the rest of your life, and write the SSH commands for local, remote and dynamic port forwarding without looking them up.
8. You will be able to describe how a CDN handles one request end to end, and explain anycast and DNS steering as two separate mechanisms that are constantly confused.
9. You will be able to compute the theoretical minimum round trip from India to the United States from the speed of light in glass, and say why buying more bandwidth will not fix a slow page.
10. You will be able to read and write IPv6 addresses, explain SLAAC and neighbour discovery, and say what IPv6: (none) in the reader's own output did and did not cost them.

34.1 localhost, 127.0.0.1 and 0.0.0.0

■ 34.1.1 PLAIN - in simple words

1. Three things get confused constantly, and all three appear in error messages every day. We will separate them once and for all.
2. `127.0.0.1` is an address that means **this machine**. Data sent there never leaves the computer.
3. `localhost` is a **name** for that address. It is a word, not an address, and something has to turn it into an address first.
4. `0.0.0.0` is not really an address at all. It is a placeholder, and it means two different things in two different places.

5. When a program **listens** on 0.0.0.0, it means “accept connections on every network the machine has”.
6. When something **sends** to 0.0.0.0, it means “this host, or an address I do not know yet”.
7. That double meaning is the whole confusion. Same four numbers, opposite jobs, decided by whether you are listening or sending.
8. The practical result trips up almost every beginner. A server started on 127.0.0.1 works perfectly from the same machine and is completely invisible from any other machine.
9. That is not a bug and not a firewall. It is the address doing exactly what it says.

■ 34.1.2 PLAIN – a picture in your head

1. Think of a large office building with a postal address on the street, and an internal mail system for moving envelopes between desks.
2. 127.0.0.1 is the internal mail system. You write a note, drop it in the internal tray, and it reaches another desk in the same building. It never goes near the street.
3. localhost is the phrase “internal mail”. It is a word people use. The mail room has to look it up to know which tray that is.
4. Binding to 127.0.0.1 is like telling reception “I only accept internal mail”. A courier from outside is turned away at the door.
5. Binding to 0.0.0.0 is like saying “I accept mail from any entrance the building has, including the loading bay and any door they add next year”.

Where this comparison breaks:

6. In a real building, a courier could still walk in and hand you the envelope in person. On a computer there is no such route. A packet addressed to 127.0.0.1 that arrives from a cable is thrown away by the operating system before any program sees it.
7. And the office has one internal mail system. Your machine has sixteen million loopback addresses, all valid, all pointing at itself.

■ 34.1.3 PLAIN – a worked example

1. Here is the loopback interface on a Mac, and the same idea on Linux.

```
$ ifconfig lo0
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
    inet 127.0.0.1 netmask 0xff000000
    inet6 ::1 prefixlen 128
    inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
```

2. Notice the MTU is 16384 bytes, not the usual 1500. There is no cable to respect, so the operating system allows much bigger chunks.
3. Notice ::1. That is the IPv6 version of 127.0.0.1, and it matters more than people expect, as we will see in a moment.
4. Now start a web server two ways and look at the difference.

```
$ python3 -m http.server 8000 --bind 127.0.0.1
$ lsof -nP -iTCP:8000 -sTCP:LISTEN
COMMAND PID USER FD TYPE NODE NAME
Python 4411 you 4u IPv4 TCP 127.0.0.1:8000 (LISTEN)

$ python3 -m http.server 8000 --bind 0.0.0.0
$ lsof -nP -iTCP:8000 -sTCP:LISTEN
```

```
COMMAND  PID  USER  FD  TYPE  NODE NAME
Python   4498  you   4u  IPv4  TCP   *:8000 (LISTEN)
```

5. The * in the second output is the whole story. The star means “any local address”. The first case names one address and only one.
6. From a phone on the same home WiFi, the first server is unreachable and the second answers on the laptop’s address, for example 192.168.0.24:8000.
7. Here is the modern trap. Ask for localhost and you may be handed ::1 before 127.0.0.1.

```
$ curl -v http://localhost:8000/
* Trying [::1]:8000...
* connect to ::1 port 8000 failed: Connection refused
* Trying 127.0.0.1:8000...
* Connected to localhost (127.0.0.1) port 8000
```

8. If the server had been listening only on the IPv4 loopback and the client had given up after the first attempt, you would see “connection refused” from a server that is plainly running. This exact problem hit many Node.js users after version 17, released in October 2021, changed the default order in which resolved addresses are tried.

■ 34.1.4 PLAIN – what is really happening inside

1. The loopback interface is a fake network card built into the operating system. It has no hardware, no driver for a chip, and no cable.
2. When a program sends to 127.0.0.1, the kernel builds a normal IP packet, hands it to the loopback device, and the loopback device immediately hands it back up the receive path.
3. So the packet goes down the sending half of the stack and straight up the receiving half, inside the same machine, in microseconds.
4. Everything else still happens. There is a real TCP handshake, real sequence numbers, real ports. You can capture it with `tcpdump -i lo0`.
5. The whole block 127.0.0.0 to 127.255.255.255 is loopback, not just the one address. That is 16,777,216 addresses, and every one of them means “this machine”.
6. On Linux you can use any of them straight away. Running one test service on 127.0.0.2:80 and another on 127.0.0.3:80 is a genuinely useful trick.
7. On macOS only 127.0.0.1 is configured by default, and you must add the others yourself with `sudo ifconfig lo0 alias 127.0.0.2 up`.
8. A packet from the outside world claiming to be from or to 127.x.x.x is called a **martian packet**, and the kernel discards it. That rule is why binding to loopback is a real isolation boundary and not just a hint.
9. 0.0.0.0 as a listening address is a wildcard. The socket is not tied to any one address, so it accepts anything arriving on any interface, including interfaces created later, such as a VPN tunnel that comes up afterwards.
10. /etc/hosts is a plain text file that maps names to addresses, and it is consulted before DNS on almost every system. Two columns: the address first, then one or more names.

■ 34.1.5 TECHNICAL – the engineer’s version

1. 127.0.0.0/8 is reserved for loopback by RFC 1122, October 1989, section 3.2.1.3, which states that a host must never send a datagram with such an address outside itself, and must discard one that arrives from a link. RFC 6890, April 2013, records it in the special-purpose address registry.
2. ::1/128 is the IPv6 loopback, defined in RFC 4291. IPv6 spends exactly one address on it rather than a sixteen-million block.

3. `localhost` is a special-use domain name under RFC 6761, February 2013. Resolvers are supposed to answer it locally with a loopback address and never send it to a DNS server. Not every stack obeys this.
4. `0.0.0.0` is `INADDR_ANY` in the sockets API. Passing it to `bind()` gives a wildcard binding. In IPv6 the equivalent is `in6addr_any`, written `::`.
5. A dual-stack listener bound to `::` will, on Linux with `net.ipv6.bindv6only` set to 0, also accept IPv4 connections through IPv4-mapped addresses of the form `::ffff:192.168.0.24`. On some BSDs the default is the opposite. This is a per-system default, not a standard.
6. `0.0.0.0` as a destination or source means “this host on this network” per RFC 1122. A DHCP client sends its first DISCOVER from source `0.0.0.0` because it has no address yet. Most stacks quietly rewrite a `connect()` to `0.0.0.0` as a connect to loopback, which is an implementation detail and not something to rely on.
7. `0.0.0.0/0` in a routing table is the default route: the least specific match possible, used when nothing else matches.

Where written	Meaning
<code>bind(0.0.0.0)</code>	all local addresses
source <code>0.0.0.0</code>	I have no address yet
route <code>0.0.0.0/0</code>	default route
<code>connect(0.0.0.0)</code>	usually loopback

8. The `/etc/hosts` file is the direct descendant of `HOSTS.TXT`, the single file distributed by the Stanford Research Institute Network Information Center before DNS existed. On Linux the order of name sources is set in `/etc/nsswitch.conf`, usually `hosts: files dns`. On Windows the file is at `C:\Windows\System32\drivers\etc\hosts`. On macOS `mDNSResponder` reads `/etc/hosts` and it takes precedence over DNS.
9. Docker is where this bites hardest. Inside a container, `127.0.0.1` is the container’s own loopback namespace, not the host’s. A server bound to `127.0.0.1` inside a container is unreachable even with `-p 8080:8080`, because the published port forwards to the container’s external address, not its loopback. Bind to `0.0.0.0` inside containers.
10. The reverse form `-p 127.0.0.1:8080:8080` publishes the port only on the host’s loopback, which is the correct way to expose a database container to the host and to nothing else.
11. Observation commands: `lsof -nP -iTCP -sTCP:LISTEN` on macOS, `ss -tlnp` on Linux, `netstat -ano | findstr LISTENING` on Windows, `dscacheutil -q host -a name localhost` to see what macOS resolves.

■ 34.1.6 WORDS – remember these

1. Loopback — a fake network card inside the machine — a virtual interface that returns transmitted packets to the local receive path.
2. `localhost` — the word for “this machine” — a special-use domain name reserved by RFC 6761, resolving to `127.0.0.1` or `::1`.
3. Wildcard bind — listen on everything — binding a socket to `INADDR_ANY`, `0.0.0.0`, or `in6addr_any`, `::`.
4. Martian packet — an impossible address arriving from outside — a datagram whose source or destination is reserved for local use, discarded on ingress.
5. Hosts file — a small local address book — a text file of address-to-name mappings consulted ahead of DNS.

34.2 Ports, properly this time

■ 34.2.1 PLAIN – in simple words

1. An address gets a packet to a machine. A **port** gets it to the right program on that machine.
2. A port is just a number between 0 and 65,535, carried in the TCP or UDP header. It is not a physical thing.
3. The numbers are split into three groups by agreement, and the groups have different rules.
4. The first group, 0 to 1023, holds the famous ones: 22 for SSH, 80 for plain web, 443 for secure web, 53 for DNS.
5. On Mac and Linux, only an administrator may open a program on those low numbers. That rule is old and it exists for a reason we will explain.
6. The second group, 1024 to 49151, is for registered services. Anyone may use these. Databases and development servers live here.
7. The third group, 49152 upward, is for temporary use. Every time your machine makes an outgoing connection, it borrows one number from this pool, uses it, and gives it back.
8. That borrowed number is why two browser tabs can both talk to the same website without the replies getting mixed up. Each tab's connection has a different borrowed number.

■ 34.2.2 PLAIN – a picture in your head

1. Think of a company with one street address and a switchboard with numbered extensions.
2. The street address is the IP address. The extension is the port. The switchboard is the operating system.
3. Extensions 1 to 1023 are reserved for official departments: reception, accounts, security. Only management may assign them.
4. Extensions in the middle range belong to named teams who have registered their number in the company directory.
5. The top range is a pool of temporary extensions. When someone in the building phones out, the switchboard lends them a spare extension so the reply comes back to the right desk.
6. When the call ends, the extension goes back in the pool, but only after a short cooling-off period, so a late reply to the old call does not ring a new person's phone.

Where this comparison breaks:

7. A real switchboard could reuse the same temporary extension for calls to two different companies at once. A computer can too, and that is the part people get wrong. The uniqueness rule is not about the port alone. It is about the whole set of four values: your address, your port, their address, their port.
8. Also, a phone extension has one line. A listening port can hold thousands of simultaneous connections, all with the same local port number.

■ 34.2.3 PLAIN – a worked example

1. Take the reader's own failed request. macOS lends an ephemeral port from its range, and the connection is described by four values.

Source address:	192.168.0.24	(the laptop on the home LAN)
Source port:	54318	(borrowed from 49152-65535)
Dest address:	20.207.73.82	(github.com, as resolved)
Dest port:	443	(HTTPS)

2. That set of four is the **four-tuple**. It is what makes the connection unique on the machine. The kernel matches arriving packets against it.
3. Now the exhaustion arithmetic, which is the thing worth remembering. Suppose a Linux server opens many short-lived connections to one database at one address and one port.
4. Linux offers 32768 to 60999 by default, which is 28,232 ports. All four other values in the tuple are fixed, so the port number is the only thing that can vary.
5. After each connection closes, the port sits in `TIME_WAIT` for 60 seconds on Linux before it can be reused.
6. So the sustainable rate is 28,232 divided by 60, which is about 470 new connections per second to that single destination. Above that, `connect()` starts failing with `EADDRNOTAVAIL`.
7. The fix is not a bigger port range. The fix is connection reuse: a pool that keeps connections open instead of opening a new one per query.
8. Here is what listening sockets look like on a real machine.

```
$ ss -tlnp
State Local Address:Port Peer Process
LISTEN 127.0.0.1:5432 0.0.0.0:* postgres
LISTEN 0.0.0.0:8080 0.0.0.0:* java
LISTEN [::]:443 [::]:* nginx

$ netstat -an -p tcp | head -4
Proto Local Address Foreign Address (state)
tcp4 192.168.0.24.54318 20.207.73.82.443 SYN_SENT
tcp4 127.0.0.1.5432 *.* LISTEN
```

9. That `SYN_SENT` line is the reader's fault frozen in place. The connection was attempted, the port was allocated, and no reply ever changed the state.

■ 34.2.4 PLAIN – what is really happening inside

1. When you call `connect()` without choosing a port, the kernel picks one for you. This is called an **ephemeral** or dynamic port.
2. Old systems picked the next number in sequence. That was predictable, so modern systems randomize the choice to make blind attacks harder.
3. The kernel then checks that the resulting four-tuple is not already in use. If it is, it picks again.
4. When a connection closes, the side that closed first keeps the tuple in a waiting state called `TIME_WAIT`.
5. The reason is simple and important. Packets from the old connection may still be wandering the network. If the same tuple were reused immediately, a straggler could be accepted as part of the new connection.
6. The second reason is that the last acknowledgement might be lost, and the other side may resend its final packet. Something must be there to answer it.
7. The privileged port rule is a convention from early UNIX. On a machine where only an administrator could bind port 513, receiving a connection from a low port was weak evidence that a trusted program on a trusted machine sent it.
8. That trust model is long dead, because anyone can run their own machine and bind whatever they like. The restriction survives anyway, and now it mainly stops an ordinary user from hijacking the web server's port.
9. `SO_REUSEADDR` is the socket option that says "let me bind this port even though an old connection is still cooling off". Without it, restarting a server often fails with "address already in use" for up to a minute.

■ 34.2.5 TECHNICAL – the engineer’s version

1. Ports are 16-bit unsigned fields in the TCP and UDP headers, so 0 to 65,535. Port 0 is reserved; binding to it asks the kernel to choose.
2. RFC 6335, August 2011, also known as BCP 165, defines the registry and the three ranges, jointly maintained by IANA and the IETF.

Range	Name	Rule
0–1023	System, well-known	IETF review to assign
1024–49151	User, registered	IANA assignment
49152–65535	Dynamic, private	never assigned

3. The privileged range on UNIX is enforced in the kernel. On Linux the check is against `CAP_NET_BIND_SERVICE`, and since kernel 4.11 in 2017 the threshold itself is tunable with `net.ipv4.ip_unprivileged_port_start`, default 1024. Windows has never had this restriction at all.
4. The actual ephemeral ranges in use do not match the IANA suggestion, and this surprises people. Real defaults:

System	Ephemeral range	Count
Linux	32768–60999	28,232
macOS, recent	49152–65535	16,384
Windows Vista+	49152–65535	16,384
FreeBSD	10000–65535	55,536

5. Check them with `sysctl net.ipv4.ip_local_port_range` on Linux, `sysctl net.inet.ip.portrange.first net.inet.ip.portrange.last` on macOS, and `netsh int ipv4 show dynamicport tcp` on Windows.
6. RFC 6056, January 2011, specifies port randomization algorithms, motivated by off-path attacks against TCP and against DNS after Dan Kaminsky’s 2008 disclosure.
7. `TIME_WAIT` is defined in RFC 793, September 1981, as twice the maximum segment lifetime, with an MSL of 2 minutes, giving 4 minutes. Linux ignores that and hardcodes 60 seconds in `TCP_TIMEWAIT_LEN`, a compile-time constant that no `sysctl` exposes.
8. `SO_REUSEADDR` permits binding a local address that is in `TIME_WAIT`. `SO_REUSEPORT`, added in Linux 3.9 in April 2013, is different: it lets several sockets bind the identical address and port so the kernel can spread incoming connections across worker processes. On BSD the two options have older and slightly different semantics.
9. `net.ipv4.tcp_tw_reuse` allows reuse of a `TIME_WAIT` entry for a new outgoing connection when timestamps confirm safety. Its dangerous cousin `tcp_tw_recycle` broke clients behind NAT and was removed from Linux in version 4.12, July 2017. If you find advice recommending it, the advice is out of date.
10. Diagnostic commands worth memorizing:

```
lsof -nP -iTCP -sTCP:LISTEN      # macOS: who is listening
lsof -nP -i :8080                 # who owns port 8080
ss -tan state time-wait | wc -l   # Linux: TIME_WAIT count
netstat -an -p tcp | grep 443    # macOS: all port 443 sockets
sysctl net.inet.ip.portrange.first # macOS ephemeral start
```

■ 34.2.6 WORDS – remember these

1. Port — a number picking the program — a 16-bit field in the TCP or UDP header identifying an endpoint within a host.
2. Ephemeral port — a temporary number borrowed for one outgoing call — a dynamically allocated source port from the local dynamic range.
3. Four-tuple — the four values that make a connection unique — source address, source port, destination address, destination port.
4. TIME_WAIT — the cooling-off period after a connection ends — a TCP state held by the closing side for twice the maximum segment lifetime.
5. Privileged port — a low number only an administrator may open — ports below 1024, requiring root or CAP_NET_BIND_SERVICE on UNIX systems.
6. Port exhaustion — running out of temporary numbers — failure of connect () with EADDRNOTAVAIL when no free four-tuple exists.

34.3 Firewalls, properly

■ 34.3.1 PLAIN – in simple words

1. A firewall is a set of rules that decides which packets may pass and which are stopped. That is the entire idea.
2. It sits somewhere on the path: in your laptop, in your home router, at the edge of a company, or in front of a cloud server.
3. What a firewall is not: it is not antivirus, it does not read your files, and it does not know whether a program is good or evil.
4. It also cannot protect you from something you invited in. If you connect out to a hostile server, the firewall was not asked.
5. The simplest kind looks at each packet alone and asks: where is it from, where is it going, which port, which protocol.
6. A smarter kind remembers connections. When you start a conversation, it notes that fact, and it lets the replies back in without a separate rule.
7. That memory is called **state**, and it is the single biggest idea in firewalls.
8. Two directions matter. **Ingress** filtering controls what comes in. **Egress** filtering controls what goes out. Most people only think about the first.
9. Egress filtering is what stops a compromised machine from phoning home. It is also, occasionally, what stops you reaching a site for no reason you can see.

■ 34.3.2 PLAIN – a picture in your head

1. Picture a security desk at the entrance to a building.
2. The stateless guard has a printed list: “let in anyone wearing a blue badge, turn away everyone else”. Each person is judged alone, with no memory.
3. If you send a courier out with a parcel and the courier returns with a reply, the stateless guard has no idea. He checks the printed list again.
4. So with a stateless guard you need a second rule saying “let returning couriers in”, and that rule is a hole, because you cannot tell a real returning courier from someone pretending.
5. The stateful guard keeps a notebook. When your courier leaves, he writes down who left, where they went and when.
6. When a reply arrives that matches a line in the notebook, it is let in automatically. Nothing else is.
7. He crosses out lines after a while, so the notebook does not grow forever.

Where this comparison breaks:

8. The guard's notebook holds a hundred lines. A firewall's table holds hundreds of thousands, and when it fills, new connections are silently dropped. That failure looks exactly like a network outage and is one of the nastiest faults in operations.
9. And the guard can see faces. A firewall in front of an HTTPS site cannot see the contents at all, only the outside of the envelope.

■ 34.3.3 PLAIN – a worked example

1. The same policy, “allow SSH and HTTPS in, allow everything out, drop the rest”, written four ways on real systems.

```
# Linux, iptables
iptables -P INPUT DROP
iptables -A INPUT -i lo -j ACCEPT
iptables -A INPUT -m conntrack \
    --ctstate ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -p tcp --dport 22 -j ACCEPT
iptables -A INPUT -p tcp --dport 443 -j ACCEPT

# Linux, nftables
nft add table inet filter
nft add chain inet filter input \
    '{ type filter hook input priority 0; policy drop; }'
nft add rule inet filter input ct state established,related accept
nft add rule inet filter input iif lo accept
nft add rule inet filter input tcp dport { 22, 443 } accept

# macOS and BSD, /etc/pf.conf
set skip on lo0
block drop in all
pass out all keep state
pass in proto tcp to any port { 22, 443 } keep state

# AWS security group, one ingress rule
aws ec2 authorize-security-group-ingress \
    --group-id sg-0abc123 --protocol tcp \
    --port 443 --cidr 0.0.0.0/0
```

2. Read the common shape. Every one of them: default deny inbound, allow loopback, allow replies to things we started, then a short list of open ports.
3. In the cloud version there is no “allow established” line, because AWS security groups are stateful by design and you cannot make them otherwise.
4. Look at the connection tracking table itself on a Linux box.

```
$ sudo conntrack -L | head -3
tcp 6 431999 ESTABLISHED src=10.0.1.5 dst=20.207.73.82
    sport=54318 dport=443 [ASSURED] mark=0 use=1
tcp 6 118 TIME_WAIT src=10.0.1.5 dst=10.0.2.9
    sport=41022 dport=5432 [ASSURED] mark=0 use=1
```

5. The number after the protocol is the seconds left on that entry. 431,999 seconds is just under five days, which is the default timeout for an established TCP flow on Linux, and it is far too long for a busy box.

■ 34.3.4 PLAIN – what is really happening inside

1. A stateless filter is a loop. For each packet, walk the rule list from the top, compare fields, and act on the first rule that matches.
2. The fields compared are the same handful every time: protocol number, source address, destination address, source port, destination port, and TCP flags.
3. A stateful filter does that only for the first packet of a flow. If the packet is allowed, it writes an entry in a table.
4. The entry holds the four-tuple, the protocol, the current TCP state, byte counters, and a timer.
5. Every later packet is looked up in that table first, usually with a hash, in constant time. Only unmatched packets fall through to the rules.
6. That is why stateful firewalls are fast for long connections and expensive for floods of new ones. A SYN flood attacks the table, not the rules.
7. For UDP, which has no connections, the firewall invents one. It creates an entry when it sees an outgoing datagram and expires it after a short timeout, usually 30 seconds. This is why your video call needs regular traffic to keep working.
8. An application-layer firewall goes further and parses the payload. It knows what an HTTP request looks like, so it can act on a method, a path or a header.
9. It cannot do that if the traffic is encrypted, unless it also terminates the encryption, which means installing its own certificate authority on every client. That is a large decision, not a checkbox.

■ 34.3.5 TECHNICAL – the engineer’s version

1. The vocabulary, ordered by how much the device understands:

Kind	Sees	Typical example
Stateless filter	one packet	router ACL, AWS NACL
Stateful filter	flows	pf, iptables, sec group
Proxy firewall	full protocol	TIS FWTK, Squid
Next generation	app and user	Palo Alto, Fortinet
Web app firewall	HTTP semantics	ModSecurity, AWS WAF

2. History, with names and dates. Jeff Mogul’s screend at Digital Equipment Corporation in 1989 was an early packet filter. DEC SEAL in 1991, built by Marcus Ranum, is widely called the first commercial firewall. Ranum’s TIS Firewall Toolkit followed in 1993.
3. Stateful inspection was commercialized by Check Point in FireWall-1, 1994, and patented by Gil Shwed, United States patent 5,606,668, filed December 1993 and granted February 1997.
4. William Cheswick and Steven Bellovin published “Firewalls and Internet Security: Repelling the Wily Hacker” in 1994, which is where much of the modern vocabulary comes from.
5. Palo Alto Networks was founded in 2005 by Nir Zuk, and the term next-generation firewall was popularized by Gartner around 2009. It means stateful inspection plus application identification, user identity and intrusion prevention in one device. It is a marketing category, not a standard.
6. ModSecurity, the first widely used open web application firewall, was released by Ivan Ristic in 2002. The OWASP Core Rule Set is the common rule pack.
7. Linux lineage: ipfwadm in kernel 2.0, 1996; ipchains in 2.2, January 1999; iptables with the net-filter framework by Rusty Russell in 2.4, January 2001; nftables merged in kernel 3.13, January 2014. Debian 10 in 2019 made nftables the default backend.
8. BSD lineage: pf was written by Daniel Hartmeier for OpenBSD 3.0 in December 2001, after a licence dispute over IPFilter. Apple adopted pf in Mac OS X 10.7 Lion, July 2011. macOS also has

a separate per-application firewall, controlled by `socketfilterfw`, present since Mac OS X 10.5 Leopard in 2007.

9. Windows: Internet Connection Firewall shipped in Windows XP in 2001, was enabled by default and renamed Windows Firewall in XP Service Pack 2, August 2004, gained outbound rules in Vista in 2007, and was renamed Microsoft Defender Firewall around 2019.
10. Connection tracking limits are real operational numbers. `nf_conntrack_max` is sized from RAM at boot and is often between 65,536 and 262,144. `nf_conntrack_tcp_timeout_established` defaults to 432,000 seconds. When the table fills, the kernel logs “`nf_conntrack: table full, dropping packet`” and new connections vanish without explanation.
11. Cloud: an AWS security group is stateful, allows only `allow` rules, and is evaluated as the union of all groups attached to an interface. A network ACL is stateless, has numbered `allow` and `deny` rules evaluated in order, and requires you to open the ephemeral range 1024-65535 for return traffic. Both fail silently, which matters enormously in section 34.4.
12. Observation: `sudo pfctl -sr` and `pfctl -si` on macOS, `iptables -L -n -v` and `nft list ruleset` on Linux, `netsh advfirewall firewall show rule name=all` on Windows, `conntrack -L` and `cat /proc/sys/net/netfilter/nf_conntrack_count`.

■ 34.3.6 WORDS – remember these

1. Firewall — a rule-based gate for packets — a policy enforcement point filtering traffic on header fields, flow state or application content.
2. Stateless filter — judges each packet alone — a filter with no flow table, requiring explicit rules for return traffic.
3. Stateful inspection — remembers conversations — filtering using a connection tracking table keyed on the flow tuple and protocol state.
4. Connection tracking table — the firewall’s notebook — the in-memory table of active flows, with states, timers and counters.
5. Ingress and egress — inbound and outbound filtering — policy applied to traffic entering or leaving an interface or network boundary.
6. Web application firewall — a filter that reads HTTP — a layer 7 filter matching request patterns against attack signatures.

34.4 DROP versus REJECT

■ 34.4.1 PLAIN – in simple words

1. When a firewall decides not to let a packet through, it has two ways to behave, and the difference decides how your day goes.
2. **REJECT** means: refuse, and say so. The firewall sends a short message back saying “no”.
3. **DROP** means: refuse, and say nothing. The packet is deleted. No reply of any kind is generated.
4. To the client, REJECT is a closed door with a sign on it. DROP is a wall where you expected a door.
5. With REJECT, your program fails almost instantly, usually in well under a second, with a clear error like “connection refused”.
6. With DROP, your program has no idea anything happened. It assumes the request got lost, waits, sends it again, waits longer, sends it again.
7. So it hangs. Fifteen seconds, thirty seconds, seventy-five seconds, depending on the system and the program, and then it gives up with “timed out”.
8. Both outcomes are a refusal. Only one of them tells you so.

9. This is the exact shape of the reader's own fault, and we will come back to it at the end of this section.

■ 34.4.2 PLAIN – a picture in your head

1. You knock on a door in a large building looking for an office.
2. In the REJECT case, someone opens the door and says “wrong floor”. You know immediately. Total time: two seconds. You go and look elsewhere.
3. In the DROP case, nobody answers. You wait. You knock again, louder. You wait longer. You knock a third time.
4. Eventually you leave. And here is the key point: you still do not know whether the office is empty, whether the person is ignoring you, or whether you were at the wrong building all along.
5. That is the whole cost of silence. The refusal carries no information.
6. Now think about why a building might do this on purpose. Someone walking the corridor knocking on every door learns nothing from silence. If every wrong door said “wrong floor”, they would map the building in a minute.

Where this comparison breaks:

7. A person knocking gives up in a minute. A computer keeps a strict schedule of retries with doubling gaps, and the total is predictable to the second.
8. And in a building the silence comes from one identifiable door. On a network, the silence could come from any of a dozen devices on the path, and you cannot tell which, because none of them left a trace.

■ 34.4.3 PLAIN – a worked example

1. The same request against three different policies, with real timings.

```
Case 1: REJECT with a TCP reset
$ time curl -v https://example.test
* Trying 203.0.113.9:443...
* connect to 203.0.113.9 port 443 failed: Connection refused
real 0m0.031s

Case 2: REJECT with ICMP administratively prohibited
$ time curl -v https://example.test
* Trying 203.0.113.9:443...
* connect to 203.0.113.9 port 443 failed: No route to host
real 0m0.042s

Case 3: DROP, nothing sent back
$ time curl -v https://example.test
* Trying 203.0.113.9:443...
* Connection timed out after 15001 milliseconds
real 0m15.004s
```

2. Case 3 is the reader's session, character for character. `curl -v` printed Trying 20.207.73.82:443... and then produced nothing for 15 seconds.
3. Now the packets on the wire in case 3. The client alone is talking.

```
0.000 192.168.0.24.54318 > 20.207.73.82.443: Flags [S]
1.001 192.168.0.24.54318 > 20.207.73.82.443: Flags [S]
3.005 192.168.0.24.54318 > 20.207.73.82.443: Flags [S]
7.010 192.168.0.24.54318 > 20.207.73.82.443: Flags [S]
```

```
15.020 192.168.0.24.54318 > 20.207.73.82.443: Flags [S]
(no packets in the other direction, ever)
```

- The gaps double: one second, two, four, eight. That doubling is the standard retransmission backoff. Nothing in the other direction is the signature.
- Compare with the REJECT capture, which is two packets and over.

```
0.000 192.168.0.24.54318 > 203.0.113.9.443: Flags [S]
0.028 203.0.113.9.443 > 192.168.0.24.54318: Flags [R.]
```

■ 34.4.4 PLAIN – what is really happening inside

- A REJECT is an active act. The firewall generates a brand-new packet and sends it back to the source address.
- For TCP it usually sends a reset, which is a TCP packet with the RST flag set. The client's kernel sees it and immediately fails the `connect()` call with "connection refused".
- Alternatively it sends an ICMP error, a small control message. The most honest one is "communication administratively prohibited", which literally means "a policy stopped this".
- The client's kernel maps that ICMP error to an error code and the program reports something like "no route to host" or "host unreachable".
- A DROP does nothing at all. The packet is freed. No counter is incremented in anything the client can see. No log line reaches the client.
- So the client's TCP stack sits in a state called `SYN_SENT` and follows its retransmission schedule, doubling the wait each time.
- There is a second, subtler reason silence is not always deliberate. A device under heavy load may rate-limit the errors it generates, so a genuine REJECT can be swallowed and look like a DROP.
- And a black hole, where a router has no route and the error is filtered on the way back, produces the same silence with nobody having intended it.
- That is why the honest reading of silence is: something on this path is not delivering, and it did not say why. Not: someone blocked me.

■ 34.4.5 TECHNICAL – the engineer's version

- The client-visible symptom table. Learn this and you can diagnose from the error message alone.

Cause	Client sees	Typical time
Port closed, host up	ECONNREFUSED	under 50 ms
REJECT with tcp-reset	ECONNREFUSED	under 50 ms
REJECT with ICMP 3/13	EHOSTUNREACH	under 100 ms
DROP, silent	ETIMEDOUT	15 to 130 s
No route locally	ENETUNREACH	instant
Wrong host, alive	TLS or 404 error	one RTT

- The ICMP codes that carry a refusal are all type 3, destination unreachable: code 0 network unreachable, code 1 host unreachable, code 3 port unreachable, code 9 network administratively prohibited, code 10 host administratively prohibited, code 13 communication administratively prohibited. In ICMPv6 the equivalent is type 1 code 1.
- `iptables -j REJECT` defaults to `--reject-with icmp-port-unreachable`, which is type 3 code 3. `--reject-with tcp-reset` sends an RST instead, and is the politest option for TCP because it is indistinguishable from a closed port. `--reject-with icmp-admin-prohibited` is the most informative and the least used.

4. In pf the two spellings are `block drop` and `block return`, and `block return-rst` or `block return-icmp` name the reply explicitly.
5. nmap names the three outcomes directly, and this is the cleanest vocabulary there is: `open` means a SYN-ACK came back, `closed` means an RST came back, `filtered` means nothing came back. A `filtered` result is a DROP by definition.
6. Timeouts differ by system, and the reader's 15 seconds came from curl, not from the kernel:

Layer	Setting	Default
Linux kernel	<code>tcp_syn_retries</code>	6, about 127 s
macOS kernel	<code>net.inet.tcp.keepinit</code>	75000 ms
curl	<code>-connect-timeout</code>	none by default
Go net.Dialer	<code>Timeout</code>	none unless set

7. Why operators choose DROP, stated fairly. It denies a scanner confirmation that a host exists. It forces the scanner to wait for a timeout on every probe, turning a one-second sweep into a multi-hour job. It generates no traffic, so it cannot be used to reflect or amplify an attack at a third party. And it does not reveal that a filtering policy exists at all.
8. Why DROP is expensive for everyone else. Every failure mode collapses into one symptom. A dropped SYN, a routing black hole, an asymmetric return path, a full conntrack table, a dead server behind a load balancer and a national filter all produce the identical client experience.
9. Cloud platforms make this worse by having no choice. AWS security groups and network ACLs both drop silently. There is no reject option. So a one-line security group mistake presents to the developer as an unexplained 15-second hang, which they will spend an hour blaming on DNS.
10. Now the tie back, stated precisely, separating proof from suggestion.

Observation	What it proves
DNS returned 20.207.73.82	resolution worked
SYNs sent, nothing back	silent drop on the path
Same request fine on mobile	not the server, not DNS
Traceroute ends in stars	nothing on its own

11. What is proven: packets left the machine towards 20.207.73.82:443 and nothing at all came back for 15 seconds, and the same request over a different access network succeeded. That is the DROP signature and it rules out the destination being down.
12. What is not proven: which device dropped, whether it was on the forward or the return path, whether it was policy or fault, and whether it was permanent. A silent drop leaves no evidence of its author, which is exactly what its author intended.

■ 34.4.6 WORDS – remember these

1. DROP — discard and say nothing — a filtering action that frees the packet without generating any response.
2. REJECT — refuse and say so — a filtering action that generates a TCP RST or an ICMP destination-unreachable message.
3. Administratively prohibited — a policy said no — ICMP type 3 code 13, or ICMPv6 type 1 code 1.
4. Filtered — the scanner's word for silence — an nmap port state meaning no response was received to any probe.

5. Black hole — traffic that vanishes with no error — a path that discards packets and returns no ICMP, whether by policy or by fault.

34.5 Middleboxes, and the end of end-to-end

■ 34.5.1 PLAIN – in simple words

1. The simple story of the internet says two computers talk directly, and the routers in between only forward.
2. That story is no longer true anywhere. Between you and any server sits a row of machines that change, inspect or block what passes.
3. These machines have a collective name: **middleboxes**.
4. Some rewrite addresses so many homes can share one public address. That is NAT, and the reader's own trace showed private ISP addresses at hops 2 to 6.
5. Some quietly capture your web requests and answer from a local copy, without you configuring anything. Those are transparent proxies.
6. Some look inside packets to identify what application you are using, and act on it. That is deep packet inspection.
7. Some break your encryption on purpose, read the contents, and re-encrypt with their own certificate. That is TLS interception, common inside companies.
8. Some hijack your very first web request and show you a login page. That is a captive portal, the thing at every airport and hotel.
9. The pattern is always the same: something in the middle knows more about your traffic than the simple model allows.

■ 34.5.2 PLAIN – a picture in your head

1. You post a letter to a friend and imagine it travelling sealed from your hand to theirs.
2. In reality the sorting office takes your envelope, puts it in a new envelope with the office's own return address, and forwards it. That is NAT.
3. Another office keeps photocopies of popular letters and, if someone asks for one, sends the copy instead of forwarding the request. That is a transparent cache.
4. A third reads the address, the size, the thickness and the postmark to guess what the letter contains, without opening it. That is deep packet inspection on encrypted traffic.
5. A fourth actually opens the envelope, reads it, copies it into a fresh envelope, and reseals it with its own wax stamp. Your friend sees a sealed letter and may not notice the stamp changed. That is TLS interception.
6. A fifth holds every letter until you sign a form at the counter. That is a captive portal.

Where this comparison breaks:

7. In the post, opening a sealed letter is illegal and visible. On a network, TLS interception is legal, routine on company laptops, and technically invisible unless you look at the certificate chain.
8. And a letter passes maybe three offices. A packet from an Indian flat to a server in the United States passes twenty or more devices, several of which have inspection features switched on.

■ 34.5.3 PLAIN – a worked example

1. Here is the real path from the reader's session, annotated with which kind of box each hop is likely to be. Everything marked "likely" is inference, not proof.

hop	address	likely role
1	192.168.0.1	home router: NAT, DHCP, DNS, firewall
2	172.31.0.17	ISP access, private address
3	137.97.29.249	ISP public edge
4	172.26.22.235	ISP core, private address
5-6	172.16.18.33 etc	ISP core, load balanced links
7	104.44.196.187	Microsoft edge, Delhi
8-9	104.44.55.163 etc	Microsoft, Mumbai
10-12	104.44.31.62 etc	Microsoft, Pune
13+	* * *	no replies

2. Two NATs are visible in that list, not one. The home router translates 192.168.0.24 to whatever address it holds, and the presence of RFC 1918 private addresses in the ISP core is consistent with carrier-grade NAT translating again.
3. So the source address that Microsoft's edge saw was not the reader's home router address either. Two rewrites, neither of them visible from the laptop.
4. To detect TLS interception on any machine, look at who signed the certificate. A public site should chain to a public certificate authority.

```
$ openssl s_client -connect example.com:443 </dev/null 2>/dev/null \
| openssl x509 -noout -issuer
issuer=C=US, O=DigiCert Inc, CN=DigiCert TLS RSA SHA256 2020 CA1

# On an intercepted network you would instead see something like:
issuer=C=IN, O=ExampleCorp IT, CN=ExampleCorp Root CA
```

5. If the issuer is your employer, every HTTPS page you load on that machine is readable by your employer's proxy. This is not a hack. It is a purchased product, and the certificate was installed by the device management tool.

■ 34.5.4 PLAIN – what is really happening inside

1. A transparent proxy is put in the path by routing, not by configuration. The network is arranged so packets to port 80 or 443 arrive at the proxy instead of the real destination.
2. The proxy then opens its own connection onward and copies bytes between the two. Your operating system never knew.
3. Deep packet inspection on unencrypted traffic simply reads the payload. On encrypted traffic it cannot, so it reads what is still visible: the destination address, the port, the packet sizes and timing, and above all the server name sent in the clear at the start of a TLS handshake.
4. That server name field is the single most used censorship and filtering hook in the world today. It is plain text, it names the site, and it arrives before any encryption is in force.
5. TLS interception works by installing an extra root certificate on your machine. The proxy then generates a fresh certificate for every site you visit, signed by that root, and your browser trusts it because the root is in the trust store.
6. A captive portal usually does two things. It answers every DNS query with its own address, and it answers every plain HTTP request with a redirect to the login page.
7. That is why HTTPS sites fail with a certificate warning on a hotel network instead of showing the portal. The portal cannot forge a valid certificate, so the browser correctly refuses.

8. Modern operating systems work around this by deliberately making one plain HTTP request to a known address after joining a network, and checking whether the expected answer comes back.
9. If the answer is wrong, the system concludes a portal is present and opens the little login window automatically.

■ 34.5.5 TECHNICAL – the engineer’s version

1. RFC 3234, February 2002, “Middleboxes: Taxonomy and Issues”, is the document that named and catalogued them. It lists twenty-two kinds and warns about exactly the failure modes we now live with.
2. The end-to-end argument comes from the 1984 paper “End-to-End Arguments in System Design” by Jerome Saltzer, David Reed and David Clark. Its claim is that functions belong at the endpoints unless the network can do them materially better.
3. Middleboxes cause **ossification**: the network becomes unable to accept new protocols because boxes drop what they do not recognize. Attempts to deploy new TCP options routinely fail for this reason.
4. This is a large part of why QUIC, standardized in RFC 9000 in May 2021, runs over UDP and encrypts almost its entire header. It is designed so middleboxes cannot parse it and therefore cannot ossify it.
5. Server Name Indication, the field carrying the site name in the clear, is defined in RFC 6066. Encrypted Client Hello is the proposed fix; it was still working through the IETF at the time of writing, and Cloudflare enabled it broadly in 2023. Treat it as emerging, not settled.
6. TLS interception at scale is a product category. Its risks are documented: a 2017 study by Durerumeric and others, “The Security Impact of HTTPS Interception”, found many interception products weakened the connection they proxied.
7. Two famous incidents are worth knowing. Lenovo shipped consumer laptops with Superfish adware carrying a private root certificate, discovered in February 2015. Kazakhstan required citizens to install a state root certificate in July 2019, and browser vendors blocked it.
8. Captive portal standards are recent. RFC 8910 and RFC 8908, both September 2020, define a DHCP and Router Advertisement option pointing at a machine readable portal API, so the operating system can be told rather than having to guess.
9. The probes each system uses are an implementation detail but worth knowing: Apple requests a tiny page from a Apple-operated host and expects the word “Success”. Android requests a URL expecting HTTP status 204 with an empty body. Windows uses a similar text probe.
10. Filtering techniques, and how each one looks from the client, which is how you tell them apart:

Technique	Client symptom
DNS blocking	wrong or empty answer
IP or port blocking	timeout, no response
SNI-based reset	RST after ClientHello
HTTP redirect	portal page appears

11. Applying that table honestly to the reader’s session: DNS returned a real address, so DNS blocking is ruled out. Nothing came back at all, and the failure was at the SYN, before any TLS ClientHello existed, so an SNI-based reset is ruled out too. What remains is address-level or port-level discarding, or a plain path fault. The evidence cannot separate those two.

■ 34.5.6 WORDS – remember these

1. Middlebox — a device in the path that does more than forward — any on-path element that inspects, modifies or filters traffic.
2. Transparent proxy — an intercepting cache you did not configure — a proxy inserted by routing rather than by client settings.
3. Deep packet inspection — looking inside, not just at the label — examination of packet payloads or metadata to classify or control traffic.
4. TLS interception — legal, installed man-in-the-middle — decryption and re-encryption using a locally trusted root certificate.
5. Captive portal — the hotel login page — a network that intercepts the first request and redirects to an authentication page.
6. Ossification — the network refusing anything new — the inability to deploy new protocol behaviour because middleboxes drop unfamiliar traffic.

34.6 VPNs, what they do and what they do not

■ 34.6.1 PLAIN – in simple words

1. A VPN, a virtual private network, does one mechanical thing: it wraps your traffic inside another connection to a server somewhere else, and lets that server send it onward.
2. Your packets go out looking like ordinary encrypted traffic to one address. Inside them are your real packets to their real destinations.
3. The VPN server unwraps them, sends them on with its own address, receives the replies, wraps them, and sends them back to you.
4. Two things change. First, your local network and your internet provider can no longer see where you are going. They only see the VPN server.
5. Second, the websites you visit see the VPN server's address, not yours, and your path to them now starts wherever that server is.
6. There are two main uses. One joins two whole offices into one network. The other lets a single laptop join a network it is not physically on.
7. The important honest point comes early: a VPN moves your trust. It does not remove it. Whoever runs the VPN can see what your provider used to see.
8. It is not anonymity. It does not stop tracking. It does not make you safe. It changes who is in a position to watch.

■ 34.6.2 PLAIN – a picture in your head

1. Imagine you write letters, but you do not want your local post office to know who you write to.
2. So you put your addressed letter inside a second envelope and post it to a forwarding service in another city. The local post office only sees “letter to the forwarding service”.
3. The service opens the outer envelope, posts your real letter with its own return address, receives the reply, and posts it back to you inside another outer envelope.
4. Your local post office learns nothing about your correspondents. Your correspondents learn nothing about your address.
5. The forwarding service, of course, learns everything. It sees every real address and, if the inner letter is not itself sealed, every word.
6. That is the trade in one paragraph. You have not become invisible. You have chosen a different party to be visible to.

Where this comparison breaks:

7. The inner letter is usually already sealed. Almost all web traffic is HTTPS, so the VPN provider sees which sites you visit and how much you transfer, but not the contents of pages.
8. And the forwarding service adds distance. If you are in India and the service is in Germany, every letter now travels to Germany first, which is why a VPN often makes things slower even when it works perfectly.

■ 34.6.3 PLAIN – a worked example

1. Here is the reader's failing path, and the same request with a full-tunnel VPN to a server in Singapore.

```
Without a VPN:
laptop -> home router -> ISP core -> ISP edge ->
Microsoft Delhi -> Mumbai -> Pune -> nothing

With a full-tunnel VPN:
laptop -> home router -> ISP core -> ISP edge ->
VPN server in Singapore -> Microsoft Singapore edge ->
github.com (works)
```

2. Everything after the VPN server is a different path across a different set of networks. If the device that dropped the packets sat on the old path and not the new one, the problem simply does not occur.
3. That is why “turn on a VPN” so often appears to fix an unexplained network fault. It is not magic. It is a different route.
4. Now look at what the tunnel looks like on the machine.

```
$ ifconfig utun4
utun4: flags=8051<UP,POINTOPOINT,RUNNING,MULTICAST> mtu 1420
    inet 10.7.0.6 --> 10.7.0.6 netmask 0xffffffff00

$ netstat -rn | head -6
Destination    Gateway         Flags   Netif
0/1            10.7.0.5        UGSc    utun4
default        192.168.0.1     UGScg   en0
128.0/1        10.7.0.5        UGSc    utun4
203.0.113.77   192.168.0.1     UGHS    en0
```

5. Read those routes carefully, because they contain the cleverest trick in VPN software. The original default route through 192.168.0.1 is still there, untouched.
6. Instead, two routes were added: 0.0.0.0/1 and 128.0.0.0/1. Together they cover the entire address space, and both are more specific than the default route, so both win.
7. The single exception is the fourth line: a direct route to the VPN server's own public address through the real gateway. Without that, the tunnel's own packets would try to go through the tunnel, which cannot work.
8. Note the MTU of 1420 rather than 1500. The wrapping costs bytes, so the inner packets must be smaller.

■ 34.6.4 PLAIN – what is really happening inside

1. The VPN client creates a virtual network interface. On macOS these are called utun, on Linux tun or wg0, on Windows a virtual adapter.
2. To the operating system it looks like an ordinary network card, so ordinary routing rules can send traffic to it.
3. Anything the kernel routes into that interface is handed to the VPN software as a raw IP packet.

4. The software encrypts that packet, wraps it in a normal UDP or TCP packet addressed to the VPN server, and sends it out of the real interface.
5. So the same physical link now carries an outer packet, whose payload is a whole inner packet. This is **encapsulation**, and it is the same idea as putting one envelope inside another.
6. At the far end, the server decrypts, extracts the inner packet, and forwards it into the internet from its own address, keeping a translation record so it can return the reply to you.
7. Split tunnelling means only some routes point at the virtual interface. Work traffic goes through the tunnel, everything else goes out normally.
8. Full tunnelling means every route points at the virtual interface, so all traffic goes through the tunnel.
9. Split is faster and cheaper. Full is safer for a hostile local network, because nothing at all leaks to the local link.

■ 34.6.5 TECHNICAL – the engineer’s version

1. Deployment shapes: **site-to-site** joins two networks with a permanent gateway-to-gateway tunnel, usually IPsec, and users are unaware of it. **Remote access** connects one device into a network, and the user runs a client.
2. The protocol comparison, with the facts that decide real choices:

Protocol	Year	Transport	Note
PPTP	1999	TCP 1723, GRE	broken, do not use
L2TP/IPsec	1999–2001	UDP 500, 4500	slow, widely built in
IPsec IKEv2	2005–2014	UDP 500, 4500	best for roaming
OpenVPN	2001	UDP 1194 or TCP	flexible, userspace
WireGuard	2016–2020	UDP, one port	smallest and fastest

3. IPsec is a family, not one protocol. RFC 4301, December 2005, gives the architecture. ESP, RFC 4303, does the encryption. IKEv2, RFC 7296, October 2014, negotiates keys. NAT traversal wraps ESP in UDP port 4500 per RFC 3948. MOBIKE, RFC 4555, lets a tunnel survive an address change, which is why IKEv2 is the usual choice on phones.
4. OpenVPN was first released on 13 May 2001 by James Yonan. It runs in user space over OpenSSL or mbedTLS, defaults to UDP port 1194, and can be forced over TCP port 443 to look like ordinary web traffic, at the cost of TCP inside TCP, which behaves badly under loss.
5. WireGuard was written by Jason A. Donenfeld, first released publicly in 2016 and merged into the Linux kernel in version 5.6, released 29 March 2020. It is roughly 4,000 lines of kernel code, against hundreds of thousands for a full IPsec plus OpenVPN stack. Small code means fewer places for bugs and an auditable design.
6. WireGuard’s speed comes from three choices. It runs in the kernel. It has no cipher negotiation at all: the algorithms are fixed at Curve25519, ChaCha20-Poly1305, BLAKE2s and HKDF, using the Noise protocol framework. And it is stateless in the sense that there is no session setup handshake per connection, only a periodic key exchange.
7. The trade-off is real and worth stating: with no negotiation, upgrading the cryptography means replacing the protocol version everywhere at once. The designers consider that a feature. Some enterprise architects consider it a problem. Both positions are defensible.
8. PPTP should never be used. RFC 2637, July 1999, is informational, and its authentication, MS-CHAPv2, was broken publicly by Moxie Marlinspike and David Hulton at Defcon in July 2012,

reducing the effective work to a single DES key. Apple removed PPTP support in macOS Sierra 10.12 and iOS 10 in 2016.

9. L2TP alone, RFC 2661, August 1999, provides no encryption whatsoever. It is always paired with IPsec, per RFC 3193, and the pairing is often misconfigured.
10. About the reader's `utun` interfaces. `utun` is macOS's user tunnel kernel control interface, and a `utun` device is created by any process that asks the kernel for one. macOS itself creates several for its own features, so seeing `utun0` through `utun3` on a stock machine with no VPN is normal. Their presence is not evidence that a VPN was running.
11. To tell a real VPN from a system interface: a live tunnel has a routable address, at least one route pointing at it, and traffic counters that move. Check with `ifconfig utunN`, `netstat -rn`, `scutil --nc list` for configured VPN services, and `netstat -ibn` for byte counters.
12. What a commercial VPN genuinely protects against: a hostile or badly-run local network, such as a shared cafe or hotel WiFi; your access provider's visibility into which sites you reach; and geographic restrictions based on the apparent country of your address.
13. What it does not do, stated without softening. The provider now sees exactly what your provider used to see, and you have less legal recourse against them. It is not anonymity, because your account, your payment and your behaviour identify you. It does not stop cookies, browser fingerprinting, logins or advertising identifiers, which are all above the network layer. It does not protect a compromised device. And a "no logs" claim is a promise, not a mechanism.
14. Applied to the reader's fault: a full-tunnel VPN would probably have made the symptom disappear, because the path after the VPN server is entirely different. That is a workaround. It adds latency, costs money, moves trust to a third party, and leaves the broken path broken for everybody else on that provider. The fix is to report the fault with the evidence from Chapter 29.

■ 34.6.6 WORDS – remember these

1. VPN — a private path built over a public one — an encrypted tunnel encapsulating IP traffic between a client or site and a gateway.
2. Encapsulation — a packet inside a packet — carrying a complete inner datagram as the payload of an outer one.
3. Split tunnelling — only some traffic goes through — a route configuration sending selected prefixes over the tunnel interface.
4. Full tunnelling — everything goes through — installing `0.0.0.0/1` and `128.0.0.0/1` so all traffic prefers the tunnel.
5. `utun` — the macOS virtual tunnel device — a kernel control interface providing point-to-point tunnel interfaces to user-space processes.
6. MOBIKE — a tunnel that survives changing networks — IKEv2 mobility and multihoming, RFC 4555.

34.7 Proxies, forward and reverse

■ 34.7.1 PLAIN – in simple words

1. A proxy is a machine that makes a request on your behalf, or answers one on somebody else's behalf. Two jobs, two names, endlessly confused.
2. A **forward proxy** sits next to the clients. You send it your request, it fetches the page, it gives you the result.
3. A **reverse proxy** sits next to the servers. The public sends requests to it, and it decides which real server behind it should handle each one.

4. The memory rule, worth writing down: a forward proxy hides the client from the server. A reverse proxy hides the server from the client.
5. Forward proxies exist for filtering, caching, and control. Schools and companies use them.
6. Reverse proxies exist for spreading load, terminating encryption, caching, and routing by path or hostname. Almost every website has one.
7. You have used a reverse proxy every day of your life without knowing. You have probably only used a forward proxy at work or at school.
8. A proxy is not a VPN. A proxy handles particular connections for particular programs. A VPN handles all traffic for the whole machine.

■ 34.7.2 PLAIN – a picture in your head

1. A forward proxy is a purchasing department. You are not allowed to buy directly. You submit a request, they buy the item, they hand it to you.
2. They can refuse certain purchases, keep a record, and reuse a bulk item they already have in stock instead of buying it again.
3. The supplier only ever deals with the purchasing department, and never learns which employee wanted the item.
4. A reverse proxy is a receptionist at a large company. Every visitor comes to the desk. The receptionist decides which department handles the visitor and walks them there.
5. Visitors never learn how many departments exist, where they sit, or that one of them was replaced last night.

Where this comparison breaks:

6. A receptionist handles one visitor at a time. A reverse proxy handles tens of thousands of connections at once and can send one visitor's request to two departments and merge the answers.
7. And the purchasing department knows what you bought. A forward proxy handling HTTPS traffic mostly does not, because it only opens a blind tunnel, unless it is also intercepting your encryption.

■ 34.7.3 PLAIN – a worked example

1. The two shapes, drawn.

```
FORWARD PROXY (near the client)

you -> proxy -> internet -> the site
    ^ you configured this
    ^ the site sees the proxy's address

REVERSE PROXY (near the server)

you -> internet -> reverse proxy -> app server 1
                                   -> app server 2
    ^ you never configured this
    ^ you see one address and one name
```

2. Now a forward proxy in action for a plain HTTP request. The request line carries the whole address, which it never does otherwise.

```
GET http://example.com/page.html HTTP/1.1
Host: example.com
Proxy-Connection: keep-alive
```


- For an encrypted request the proxy cannot do that, so the client asks for a tunnel instead, using a special method.

```
CONNECT example.com:443 HTTP/1.1
Host: example.com

HTTP/1.1 200 Connection established

(from here the proxy copies bytes blindly in both
directions; the TLS handshake happens end to end)
```

- The proxy therefore learns the hostname and port from the CONNECT line, and the number of bytes and the timing, but not the page contents.
- Configuring one from the shell uses environment variables, which are a convention that most tools follow and a few ignore.

```
export http_proxy=http://proxy.corp.example:3128
export https_proxy=http://proxy.corp.example:3128
export no_proxy=localhost,127.0.0.1,.corp.example
curl -v https://example.com/      # now goes via CONNECT
curl --proxy socks5h://127.0.0.1:1080 https://example.com/
```

- The h in socks5h matters: it means the proxy resolves the hostname, so your DNS queries do not leak to your local resolver.

■ 34.7.4 PLAIN - what is really happening inside

- With a forward proxy, your program opens a TCP connection to the proxy, not to the destination. There are two separate connections and the proxy joins them.
- For plain HTTP the proxy fully understands the request and can cache, rewrite or block it.
- For HTTPS the proxy is asked to become a pipe. After it answers “200 Connection established” it stops understanding anything and just copies bytes.
- A SOCKS proxy is simpler and lower down. It does not understand HTTP at all. It just carries any TCP connection, and optionally UDP, to wherever you say.
- That is why SOCKS works for SSH, databases, game protocols and anything else, while an HTTP proxy only works for HTTP and CONNECT tunnels.
- With a reverse proxy, the public name and address belong to the proxy. It terminates the TCP connection and the encryption, reads the request, and decides.
- It can then route on the hostname, on the path, on a header, or on a cookie, and it opens its own connection to a chosen backend server.
- Because it terminates encryption, it can add caching, compression, request limiting and attack filtering in one place, for every backend behind it.
- The cost is that the backend no longer sees the real client address, so the proxy must pass it on in a header, and the backend must be configured to trust that header only from the proxy.

■ 34.7.5 TECHNICAL - the engineer’s version

- The distinction in one table:

Property	Forward proxy	Reverse proxy
Sits near	the client	the server
Configured by	the client	the site owner
Hides	the client	the servers

Property	Forward proxy	Reverse proxy
Typical software	Squid, tinyproxy	nginx, HAProxy

- The CONNECT method is defined in RFC 9110, June 2022, section 9.3.6, and was earlier described in RFC 2817, May 2000. A proxy that permits CONNECT to arbitrary ports is an open relay and will be abused, so real proxies restrict it to 443 and a short list.
- SOCKS version 4 was written by David Koblas around 1992. SOCKS5 is RFC 1928, March 1996, and adds authentication, IPv6, UDP association and, critically, remote name resolution via the domain-name address type.
- Proxy auto-configuration files are a JavaScript function named `FindProxyForURL(url, host)` returning a string such as `DIRECT` or `PROXY host:port`. Netscape introduced them in 1996. Web Proxy Auto-Discovery locates the file via DHCP option 252 or a DNS name, and has been a recurring security weakness because whoever answers first wins.
- Proxy versus VPN, at the packet level. A proxy operates at layer 5 to 7: your application makes a new TCP connection to the proxy, and the proxy makes a separate one onward. Only applications that know about the proxy use it. ICMP, DNS and everything else bypass it. A VPN operates at layer 3: whole IP packets are encapsulated, so every protocol and every program is covered whether it knows or not.
- Reverse proxy software, with real origins. nginx was written by Igor Sysoev and first released publicly on 4 October 2004, to solve the C10K problem of ten thousand concurrent connections. HAProxy was written by Willy Tarreau, first released in 2001, and is still the reference for layer 4 and layer 7 load balancing. Envoy was open-sourced by Lyft on 20 September 2016 and became the data plane of service meshes such as Istio. Caddy, by Matt Holt, first released in 2015, was the first popular server to obtain and renew TLS certificates automatically.
- Squid, the classic forward caching proxy, dates from 1996 and descends from the Harvest project.
- Where reverse proxies sit in a real deployment, outermost first: a CDN edge, then a cloud load balancer, then an ingress reverse proxy such as nginx or Envoy, then the application. Four layers of proxy is normal, and each one can add its own timeout, its own header rewriting and its own failure mode.
- Client address propagation uses X-Forwarded-For, a convention from Squid, or the standardized Forwarded header from RFC 7239, June 2014. Trusting these headers from anything other than your own proxy is a well-known vulnerability, because a client can simply send them.
- Observation: `curl -v --proxy ...` shows the CONNECT exchange, `nginx -T` prints the full resolved configuration, and `echo "show stat" | socat /var/run/haproxy.sock stdio` dumps HAProxy counters.

■ 34.7.6 WORDS – remember these

- Forward proxy — fetches things for clients — a proxy configured by the client that hides clients from origin servers.
- Reverse proxy — answers for servers — a proxy owned by the site that terminates client connections and dispatches to backends.
- CONNECT — ask a proxy for a blind pipe — the HTTP method that establishes a TCP tunnel through a proxy, used for HTTPS.
- SOCKS — a general-purpose connection relay — a layer 5 proxy protocol carrying arbitrary TCP and UDP, defined in RFC 1928.
- PAC file — a script that picks the proxy — a JavaScript `FindProxyForURL` function evaluated by the client per request.

6. X-Forwarded-For — the original client address, passed along — a convention header, standardized as Forwarded in RFC 7239.

34.8 Tunnelling in general

■ 34.8.1 PLAIN – in simple words

1. Tunnelling means putting one packet inside another packet.
2. The outer packet is addressed to a machine that knows how to unwrap it. The inner packet is addressed to where you actually wanted to go.
3. The network in between reads only the outer packet. It never looks at the inner one and does not need to understand it.
4. That is the whole trick. Everything called a tunnel is a version of it.
5. There are three honest reasons to tunnel. To carry something the network refuses to carry. To hide what you are carrying. To make two far-apart networks behave like one.
6. Because the outer packet needs its own header, a tunnel always makes packets bigger, so every tunnel eats a little of your usable space.
7. A tunnel is not automatically private. Wrapping and encrypting are different jobs. Some tunnels encrypt, many do not.
8. SSH gives you three ready-made tunnels for free. They are among the most useful tools a developer owns and almost nobody is taught them.
9. There are also services whose only job is to give a machine behind a home router a public address, by having that machine dial out first.

■ 34.8.2 PLAIN – a picture in your head

1. You want to send a letter into a building that does not accept post from outside.
2. So you seal your letter, write your colleague's internal room number on it, and put it inside a bigger envelope addressed to the reception desk.
3. The postal service only reads the outer envelope. It delivers to reception.
4. Reception opens the outer envelope, sees an internal room number, and passes the inner letter along the internal mail system.
5. Reception is the tunnel endpoint. The outer envelope is the encapsulation. The inner letter is your real packet.
6. The outer envelope also weighs something, so a fixed weight limit now holds slightly less of your actual letter.

Where this comparison breaks:

7. The post office does not silently destroy an envelope that is one gram too heavy. Networks do exactly that when a packet is too big and the message saying so is filtered. You get silence, not a bounce.
8. And reception can read your inner letter unless you sealed it. Wrapping alone gives you no privacy at all. Only encryption does.

■ 34.8.3 PLAIN – a worked example

1. SSH has three forwarding modes. Learn the letters and you own them for life.
2. -L is local forwarding. It opens a port on your machine and sends anything arriving there out through the SSH server.

```
# reach a database that only the bastion host can see
ssh -N -L 5433:db.internal:5432 user@bastion.example.com
# now, in another shell, on your own laptop:
psql -h 127.0.0.1 -p 5433 -U app appdb
```

3. Read -L 5433:db.internal:5432 as: listen on my port 5433, and from the SSH server open a connection to db.internal port 5432.
4. -R is remote forwarding. It is the mirror image. It opens a port on the SSH server and sends anything arriving there back to you.

```
# let a colleague see the site running on your laptop
ssh -N -R 9000:localhost:3000 user@public.example.com
# they open port 9000 on public.example.com and reach
# port 3000 on your laptop
```

5. By default that remote port only listens on the server's own loopback. To let other people reach it, the server needs GatewayPorts yes in its sshd_config. That is a deliberate safety default.
6. -D is dynamic forwarding. It turns your SSH connection into a SOCKS proxy, so any program can send any connection through it.

```
ssh -N -D 1080 user@bastion.example.com
curl --proxy socks5h://127.0.0.1:1080 https://intranet.example/
```

7. -N means "do not run a command on the far end, just forward". -f puts it in the background. Together they are the usual pair.
8. The three modes drawn, because the arrows are the whole point.

```
-L LOCAL FORWARD    (you open the door on your side)

  your app -> 127.0.0.1:5433 -[ssh]-> bastion -> db:5432

-R REMOTE FORWARD  (you open the door on their side)

  visitor -> public:9000 -[ssh]-> your laptop -> :3000

-D DYNAMIC          (a SOCKS proxy, any destination)

  any app -> 127.0.0.1:1080 -[ssh]-> bastion -> anywhere
```

9. Now the outside-in version. ngrok and Cloudflare Tunnel do the -R trick as a product, with a public name and a certificate.

```
ngrok http 3000
cloudflared tunnel --url http://localhost:3000
```

10. Both print a public hostname. Both work from behind a home router with no port forwarding, because the agent dialled out and the router allowed that outbound connection, exactly as it allows any web request.

■ 34.8.4 PLAIN - what is really happening inside

1. SSH forwarding is not really packet-in-packet. It is connection-in-connection, and the difference matters.
2. Your ssh client opens a normal listening socket on your machine, on the port you asked for.
3. When a program connects to that port, the client opens a new logical channel inside the one existing SSH connection and says "please connect to this host and port".

4. The SSH server makes a brand new TCP connection from itself to the target, and then copies bytes between that connection and the channel.
5. So the target sees a connection coming from the SSH server, not from you. Its logs show the bastion's address. Its firewall rules must allow the bastion.
6. Many channels share one SSH connection, so many forwards cost one TCP connection and one handshake.
7. A real network tunnel such as VXLAN or GRE works one level lower. It takes a whole packet, headers and all, and puts it in the payload of a new packet.
8. The receiving end strips the outer header and injects the inner packet as if it had just arrived on a wire. Nothing above it knows a tunnel happened.
9. That is why a layer 3 tunnel carries every protocol, and an SSH forward carries only the TCP connections you named.
10. In both cases the outer header takes space. If the original packet was already full size, the wrapped packet is now too big for the path, and it must be fragmented, refused, or dropped.

■ 34.8.5 TECHNICAL – the engineer's version

1. Encapsulations you will meet, with their real identifiers and overhead over IPv4.

Tunnel	Carried as	Overhead
IP-in-IP, RFC 2003	IP protocol 4	20 bytes
GRE, RFC 2784	IP protocol 47	24 bytes
IPsec ESP tunnel	IP protocol 50	50 to 73 bytes
VXLAN, RFC 7348	UDP port 4789	50 bytes
Geneve, RFC 8926	UDP port 6081	50 bytes plus
WireGuard	UDP, chosen port	60 bytes

2. GRE, Generic Routing Encapsulation, is RFC 2784, March 2000, updated by RFC 2890, and replaced the earlier RFC 1701 of October 1994. It has no encryption and no authentication at all. It is a plain wrapper, still used between routers and by DDoS scrubbing providers to return cleaned traffic.
3. VXLAN, Virtual eXtensible LAN, is RFC 7348, August 2014. It puts a whole Ethernet frame inside UDP. Its 24-bit VXLAN Network Identifier gives 16,777,216 segments, against 4,094 usable VLANs, which is the reason it exists. Cloud overlays and Kubernetes network plugins use it constantly.
4. Geneve, RFC 8926, November 2020, is the newer extensible version, and is what the AWS Gateway Load Balancer speaks.
5. MTU arithmetic, the part that bites. On a 1500-byte Ethernet path, a VXLAN tunnel leaves 1450 bytes for the inner frame. WireGuard's wg-quick sets the tunnel interface MTU to 1420 by default for the same reason.
6. When encapsulation shrinks the path and the ICMP "fragmentation needed" message is filtered, Path MTU Discovery fails silently and you get the classic signature: the handshake works, small responses arrive, and the first large response hangs forever. The fix is usually TCP MSS clamping, for example `iptables -t mangle -A FORWARD -p tcp --syn -j TCPMSS --clamp-mss-to-pmtu`.
7. SSH port forwarding is defined in RFC 4254, January 2006, the SSH Connection Protocol, as channels of type `direct-tcpip` for `-L` and `forwarded-tcpip` for `-R`. SOCKS5 dynamic forwarding is RFC 1928, March 1996. OpenSSH itself was forked from Tatu Ylonen's SSH 1.2.12 by the OpenBSD project and first released in December 1999.

8. Do not tunnel TCP inside TCP if you can avoid it. Two retransmission timers stack, and a lost packet triggers both, which is why `ssh -w` and TCP-based VPNs collapse on lossy links. UDP-based tunnels do not have this problem.
9. Outbound-dialled tunnel services, with real dates. ngrok was created by Alan Shreve and released in 2013. Cloudflare launched Argo Tunnel in 2018 and renamed it Cloudflare Tunnel in 2021; its agent is `cloudflared` and it connects outbound over QUIC or HTTP/2. Tailscale added Funnel in 2022.
10. The security consequence is blunt and worth stating once. A public tunnel to `localhost` exposes that service to the entire internet, including automated scanners that find new hostnames within minutes. Put authentication in front of it before you start it, not after.
11. Observation commands: `ssh -v` prints each channel as it opens, `ip link show type vxlan` and `bridge fdb show` inspect a VXLAN interface on Linux, `tcpdump -ni any udp port 4789` shows the encapsulated frames, and `ip route get 10.0.0.5` tells you whether a packet will take the tunnel.

■ 34.8.6 WORDS – remember these

1. Tunnel — a packet carried inside another packet — encapsulation of one protocol’s payload within another protocol for transit.
2. Encapsulation — the wrapping itself — prepending an outer header so intermediate routers forward on the outer address only.
3. Local forward — a door on your side — `ssh -L`, a listening socket on the client relayed to a target reached from the server.
4. Remote forward — a door on their side — `ssh -R`, a listening socket on the server relayed back to a target reached from the client.
5. Dynamic forward — a proxy, not a fixed pair — `ssh -D`, a local SOCKS5 listener whose destinations are chosen per connection.
6. VXLAN — Ethernet inside UDP — RFC 7348 overlay with a 24-bit VNI and 50 bytes of overhead, UDP port 4789.
7. GRE — a plain wrapper with no encryption — RFC 2784, IP protocol 47, 24 bytes of overhead.
8. MSS clamping — telling both ends to use smaller pieces — rewriting the TCP maximum segment size option to fit the tunnel’s path MTU.

34.9 CDNs and anycast in depth

■ 34.9.1 PLAIN – in simple words

1. A CDN, a content delivery network, is a large set of computers placed close to users, holding copies of somebody else’s files.
2. The point is distance. A file served from a machine 40 kilometres away arrives far sooner than the same file from 13,000 kilometres away, and no amount of money changes that.
3. The site that owns the content is called the **origin**. The CDN machines are called **edges** or points of presence.
4. A CDN can copy anything that is the same for everybody: images, style sheets, scripts, fonts, video pieces, downloads.
5. A CDN cannot copy things that are different for each person, such as your account page or your shopping basket, unless the site takes special care.
6. There are two ways to fill a CDN. Push, where you upload files to it in advance. Pull, where it fetches a file from the origin the first time anyone asks, and keeps it.

7. Nearly everything today is pull, because it needs no work from you.
8. **Anycast** is a separate idea that CDNs also use. It means one address exists in many places at once, and the network delivers you to whichever copy is closest by its own reckoning.
9. Anycast and DNS steering both send different users to different places, and people mix them up constantly. One works in the routing system, the other in the naming system. We will separate them properly at the end.

■ 34.9.2 PLAIN – a picture in your head

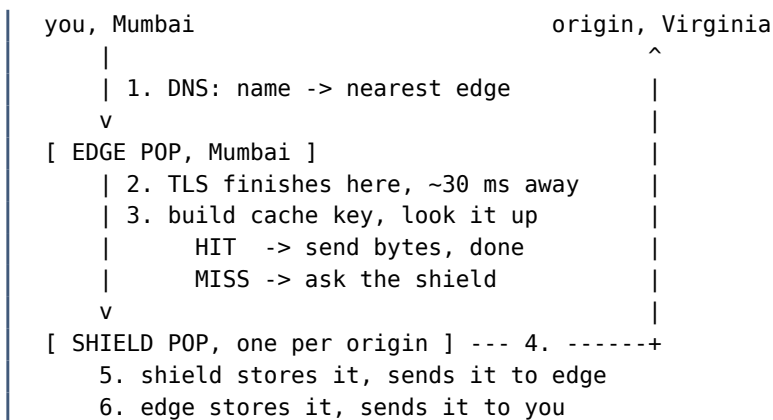
1. Think of a popular book and a national library system.
2. The origin is the one library that owns the original manuscript.
3. Rather than making every reader travel to that one building, the system prints copies and puts them in every town branch.
4. The first reader in your town asks for the book, the branch does not have it, so the branch orders one copy from the central library and keeps it on the shelf.
5. Every reader after that is served in two minutes from the local shelf. That is a cache hit.
6. The branch is told how long the copy stays valid. After that it must check with the central library before lending it again.
7. Now the anycast part. Imagine every branch has the same street address painted on the door, and the road signs in each town point to the nearest one. You always drive to “1 Library Road” and always arrive somewhere local.

Where this comparison breaks:

8. Books do not change under you. Web content does, and a stale copy on a shelf can be actively wrong, which is why purging and versioned file names exist.
9. Road signs do not change while you are driving. Internet routes do, and if they change mid-journey an anycast connection can arrive at a branch that has never heard of you and be turned away.
10. And a library branch does not run your code. A modern edge does, which is the part of this comparison that has aged the worst.

■ 34.9.3 PLAIN – a worked example

1. One request for one image, walked end to end, with a cold cache.



2. Step 3 needs a cache key. By default it is built like this.

```
key = scheme + host + path + query string
    = https + img.example.com + /logo.avif + ?v=7
```


3. Step 4 only happens on a miss. With a shield in front of the origin, 200 edges missing the same file cause one origin request, not 200.
4. What the origin sends back, and what each header does.

```
HTTP/1.1 200 OK
Content-Type: image/avif
Cache-Control: public, s-maxage=600, max-age=60
ETag: "9f2a1c"
Vary: Accept-Encoding
```

5. max-age=60 tells your browser to keep it for 60 seconds. s-maxage=600 tells shared caches such as the CDN to keep it for 600 seconds. Shared caches prefer s-maxage when both are present.
6. Vary: Accept-Encoding adds that request header to the cache key, so the compressed and uncompressed versions are stored separately.
7. What the edge sends you, with two extra headers of its own.

```
HTTP/1.1 200 OK
Age: 143
X-Cache: HIT
ETag: "9f2a1c"
```

8. Age: 143 means this copy has been in caches for 143 seconds, so it has 457 seconds of its 600 left. X-Cache: HIT is a convention, not a standard; every CDN spells it differently.
9. Now the timing, using round-trip times we will justify in section 34.10.

Case	Where served	Time to first byte
Cache hit, Mumbai edge	30 ms away	about 45 ms
Cache miss, shield hit	30 ms plus	about 250 ms
Cache miss to origin	Virginia	about 600 ms

10. That table is the entire business case for CDNs in three rows.

■ 34.9.4 PLAIN – what is really happening inside

1. The name you type does not resolve to the origin. It resolves, one way or another, to a CDN address near you.
2. There are only two mechanisms for that, and this is the crux of the whole section.
3. Mechanism one, DNS steering. The authoritative name server looks at who is asking and returns a different address to different askers. Different users get different addresses.
4. Mechanism two, anycast. Every user gets the same address, and the routing system itself carries each user to a different physical machine.
5. Most large CDNs use both: anycast for the edge addresses, and DNS to hand out which anycast address family or which service you land in.
6. Once you reach the edge, the edge terminates your TCP connection and your encryption. That is why the handshake feels fast: it finished 30 ms away, not 200 ms away.
7. The edge computes the cache key, looks in memory, then on local flash. A hit is answered immediately.
8. A miss is not sent straight to the origin. It goes to a parent or shield node, and identical concurrent misses are collapsed into one upstream request so the origin is never stampeded.
9. The connection from edge to origin is usually already open, already warmed up, and often runs over the CDN's own private network rather than the public internet.

10. Stale content is handled by time, not by magic. When the timer expires the edge revalidates with a conditional request, and a 304 Not Modified reply costs one small round trip instead of a whole file.
11. If you need something gone before its timer expires, you purge it, which is a message pushed out to every edge holding it.
12. Edge compute means the edge can also run small programs of yours before or after the cache lookup, so decisions that used to need the origin now happen 30 ms from the user.

■ 34.9.5 TECHNICAL – the engineer’s version

1. Cacheability rules, from RFC 9111, June 2022, which replaced RFC 7234. A shared cache may store a response if the method is cacheable, the status is understood, and no directive forbids it.
2. The directives that decide it, and what they really mean.

Directive	Effect on a CDN
public	may be stored by shared caches
private	browser only, never the CDN
no-store	nobody stores it at all
no-cache	store, but always revalidate
s-maxage=N	lifetime for shared caches
immutable	never revalidate before expiry

3. stale-while-revalidate and stale-if-error come from RFC 5861, May 2010. The first serves the old copy while fetching a new one in the background. The second serves the old copy when the origin is down, and it has saved more outages than any dashboard.
4. Validators: ETag with If-None-Match, and Last-Modified with If-Modified-Since, producing 304 Not Modified with no body.
5. Cache key control is where real hit rates are won. Strip tracking query parameters, normalize the order of the remaining ones, and never send Vary: User-Agent, which multiplies your object count by thousands.
6. Purging, in increasing order of danger: by exact URL, by surrogate key or tag, by prefix, and purge-everything. Fastly calls the tag header Surrogate-Key, Cloudflare calls it Cache-Tag, Akamai uses Edge-Control and its own cache tags. Purge-everything on a busy site sends every edge to the origin at once and can take the origin down.
7. The better pattern is to never purge. Put a content hash in the file name, serve it with max-age=31536000, immutable, and change the name when the content changes. One year is the maximum value RFC 9111 encourages.
8. Edge compute, with real dates. Cloudflare Workers were announced on 29 September 2017, running V8 isolates rather than containers. AWS Lambda@Edge became generally available in July 2017 and CloudFront Functions in May 2021. Fastly’s Compute product, built on WebAssembly, reached general availability in 2020.
9. Deployment shapes differ, and the industry genuinely disagrees about which is better. These are company-published figures as of 2026, which is to say marketing claims rather than audited counts, and they change constantly.

CDN	Shape	Published claim
Akamai	many small POPs	around 4,000 locations

CDN	Shape	Published claim
Cloudflare	mid-size POPs	over 300 cities
Fastly	few large POPs	roughly 100 POPs

10. The argument: many small POPs sit closer to users; few large POPs hold more objects each, so they have higher hit rates and better shielding. Both camps are right about their own metric.
11. History: Akamai was founded in 1998 by Tom Leighton and Daniel Lewin at MIT, out of work on consistent hashing, and launched service in 1999. Cloudflare was founded in 2009 by Matthew Prince, Lee Holloway and Michelle Zatlyn and launched publicly at TechCrunch Disrupt in September 2010. Fastly was founded in 2011 by Artur Bergman.
12. Now anycast, properly. Anycast was described in RFC 1546, November 1993, by Craig Partridge, Trevor Mendez and Walter Milliken. Operational guidance is RFC 4786, December 2006, which is BCP 126.
13. The mechanism is entirely BGP. The same prefix, say a /24, is announced from many autonomous systems or many locations of one autonomous system. Every router in the world independently picks one best path to that prefix using its usual decision process: local preference, then AS path length, then origin, then MED, then eBGP over iBGP, then IGP cost.
14. Therefore “nearest” in anycast means nearest in BGP policy terms, not in kilometres. A site two countries away with a shorter AS path will win over a site in your city that your ISP reaches through three transit providers.
15. Why UDP and DNS love anycast: a DNS query is one packet out, one packet back, with no state at the server. If the next query lands at a different site, nothing is lost. The 13 root server identities are served this way from more than 1,900 instances worldwide, a figure that changes monthly.
16. Why TCP needs care: every packet of a flow must reach the same site, because only that site holds the transmission control block. A packet arriving at a site with no matching socket gets a RST and the connection dies.
17. In practice it works because routes are stable for minutes to hours, and because equal-cost multi-path hashing inside a network is per-flow, keyed on the five tuple, so all packets of one connection follow one path.
18. A session sticks to one site for exactly as long as the routing decision at every hop stays the same. It is not pinned, tracked or guaranteed. It is merely stable, which is a weaker and more honest word.
19. When routing changes mid-connection, the flow is delivered to a different POP, which resets it. The user sees a broken download or a stalled upload. Short connections almost never notice. Long ones do.
20. Mitigations used in production: keep announcements stable and drain sites gracefully rather than withdrawing routes instantly, use consistent hashing in the load balancers behind the anycast address, and for QUIC use the connection ID, which lets a new front end recognize an existing connection and route it correctly, as specified in RFC 9000, May 2021.
21. A common hybrid is anycast for the first contact and a unicast address for the long transfer, which gives fast site selection and stable long flows.
22. Anycast inherits BGP’s weaknesses. A route hijack sends your users to somebody else’s machine. Real cases: Pakistan Telecom’s accidental global hijack of YouTube on 24 February 2008, and the deliberate hijack of Amazon Route 53 address space on 24 April 2018 used to steal cryptocurrency from MyEtherWallet users. RPKI route origin validation exists to reduce this.
23. The final separation, which is the most confused pair in networking.

Question	Anycast	DNS steering
Which layer	routing, BGP	naming, DNS
Addresses	one, everywhere	different per user
Who decides	the internet's routers	the authoritative server
Failover time	seconds, BGP reconverges	one TTL, 30 to 300 s

24. Two more differences that matter in an incident. DNS steering can be overridden by a client that ignores TTLs or pins an address, and there are many such clients. Anycast cannot be overridden by the client at all.
25. And DNS steering sees the resolver, not the user. The reader's own session used 1.1.1.1, Cloudflare's public resolver, rather than the ISP's. With DNS steering that can put you in the wrong region, unless the resolver forwards a truncated client subnet using EDNS Client Subnet, RFC 7871, May 2016. Cloudflare's resolver deliberately does not send client subnet by default, for privacy; Google's public resolver does.
26. In the reader's case the answer was still local: `github.com` resolved to `20.207.73.82`, an address in a Microsoft-owned range serving India, and the traceroute entered Microsoft's network at a Delhi site named `ae66-0.del01-96cbe-1b.ntwk.msn.net`. The evidence supports local steering working correctly. It does not prove which mechanism produced it.
27. Observation: `dig +short github.com` from two networks shows steering, `dig CHAOS TXT id.server @1.1.1.1` or `hostname.bind` reveals which anycast instance answered you, and `curl -sI` plus a look at Age, X-Cache or CF-Cache-Status tells you whether you got a cached copy.

■ 34.9.6 WORDS – remember these

1. CDN — machines near you holding copies of a site's files — a distributed caching and delivery network fronting an origin.
2. Origin — the real server that owns the content — the authoritative upstream a CDN fetches from on a miss.
3. Pull versus push — fetch on first request versus upload in advance — origin pull caching versus pre-positioned distribution.
4. Cache key — the label a cached copy is filed under — normally scheme, host, path and query, extended by Vary headers.
5. Origin shield — one CDN node that speaks to the origin — a mid-tier cache collapsing edge misses into a single upstream request.
6. Purge — delete a copy before it expires — an invalidation pushed to every edge, by URL, tag or prefix.
7. Edge compute — your code running in the CDN — isolate or WebAssembly runtime executing per request at the point of presence.
8. Anycast — one address in many places — the same prefix announced by BGP from multiple sites, each router choosing its best path.
9. DNS steering — a different answer per asker — authoritative DNS returning addresses chosen by resolver location, latency or health.
10. EDNS Client Subnet — telling the server roughly where the user is — RFC 7871 option carrying a truncated client prefix to the authoritative server.

34.10 Latency versus bandwidth

■ 34.10.1 PLAIN – in simple words

1. Latency is the waiting time before anything arrives. Bandwidth is how much arrives per second once it starts.
2. They are different quantities with different units, and they have almost nothing to do with each other.
3. Latency is measured in milliseconds. Bandwidth is measured in megabits per second. You cannot convert one into the other.
4. Your internet plan sells you bandwidth. Your experience of a slow website is almost always latency.
5. Latency has a hard floor set by physics. Nothing travels faster than light, and light in glass is slower than light in air.
6. Bandwidth has no such floor. You can always lay another cable, or send more colours of light down the same one.
7. So bandwidth is an engineering and money problem, and latency, past a point, is a geography problem.
8. The single most useful sentence in this chapter: **you can buy more bandwidth, you cannot buy a shorter distance.**
9. A page that makes 60 small requests to a server 200 milliseconds away is slow no matter how fast your connection is, because it spends its life waiting, not transferring.

■ 34.10.2 PLAIN – a picture in your head

1. Think of a road between two cities.
2. Latency is how long the road is, divided by the speed limit. It is the time for one car to make the trip.
3. Bandwidth is how many lanes the road has. It is how many cars can be moving at once.
4. Adding lanes does not shorten the road. A ten-lane motorway and a two-lane road of the same length take one car the same time to drive.
5. If you need to move ten thousand cars, lanes are what you want. If you need to move one car and get an answer back, length is what you want.
6. That is the whole difference, and most complaints about “slow internet” are complaints about the length of the road.

Where this comparison breaks:

7. Cars do not wait for permission. Data does. A sender may only put a limited amount on the road before it must wait for confirmation from the far end, so the length of the road actually limits how many lanes you can use.
8. On a real motorway, adding cars makes everyone slower. On a fibre link, the cable’s capacity is fixed, and what actually slows down is the queue at the entrance, which is a different mechanism with a name: *bufferbloat*.
9. And the road is not straight. Cables follow coasts, avoid mountains and share trenches, so the real distance is commonly 1.3 to 1.6 times the map distance.

■ 34.10.3 PLAIN – a worked example

1. Start with physics. Light in vacuum travels at 299,792 kilometres per second.
2. Light in optical fibre travels slower, because glass has a refractive index of about 1.47. That gives roughly 204,000 kilometres per second.
3. Round it to 200,000 kilometres per second, which is 200 kilometres per millisecond, and you get a rule of thumb worth memorizing.
4. **Ideal round trip in milliseconds is roughly the one-way distance in kilometres divided by 100.** The 100 rather than 200 is because a round trip covers the distance twice.
5. Mumbai to Ashburn, Virginia, where a large part of the United States east coast cloud sits, is about 12,850 kilometres in a straight line.
6. So the ideal round trip is about 12,850 divided by 100, which is about 128 milliseconds. That is with a perfectly straight cable and zero equipment.
7. Real measurements from an Indian home connection to that region are around 185 to 200 milliseconds. The extra 60 or so milliseconds is cable routing, plus every router, queue and last-mile hop on the way.
8. No purchase, no protocol and no optimization will ever take that number below 128 milliseconds, short of digging a straighter trench.
9. Now the second calculation, the one that explains slow pages. Suppose a 100 megabit per second line and a 200 millisecond round trip.
10. How much data can be in flight at once? That is the bandwidth-delay product.

```
BDP = bandwidth x round-trip time
    = 100,000,000 bits/s x 0.200 s
    = 20,000,000 bits
    = 2,500,000 bytes
    = about 2.5 megabytes
```

11. To use the full 100 megabits per second, the sender must be allowed to have 2.5 megabytes unacknowledged at all times.
12. The original TCP window field is 16 bits, so without the window scaling option the largest window is 65,535 bytes.
13. That caps throughput at 65,535 bytes divided by 0.2 seconds, which is about 328 kilobytes per second, or about 2.6 megabits per second.
14. On a 100 megabit line. That is 2.6 percent of what you pay for, and the cause is entirely the round trip.
15. Finally, why more bandwidth does not help a page. Take a 100 kilobyte response over that same 200 millisecond link.

Step	At 10 Mbit/s	At 100 Mbit/s
DNS, TCP, TLS, request	800 ms	800 ms
Sending 100 KB	80 ms	8 ms
Total	880 ms	808 ms

16. Ten times the bandwidth bought 8 percent. Halving the round trip to 100 milliseconds would have given 480 milliseconds, a 45 percent saving.

■ 34.10.4 PLAIN – what is really happening inside

1. A round trip is not one thing. It is four things added together, and only one of them is physics.
2. Propagation delay: the time for the signal to travel the distance. Fixed by the speed of light in the medium. You cannot improve it.
3. Serialization delay: the time to push the bits of a packet onto the wire. A 1500-byte packet takes 1.2 milliseconds at 10 megabits per second and 12 microseconds at 1 gigabit per second.
4. Queuing delay: time spent waiting in a buffer behind other packets. This is the one that varies wildly and the one that badly sized buffers make worse.
5. Processing delay: the time each router takes to look up and forward. On modern hardware this is microseconds.
6. On a long path, propagation dominates. On a congested last mile, queuing dominates. Both are latency, and neither is fixed by more bandwidth.
7. Now the feedback loop. TCP does not send everything at once. It starts small and grows, because it does not know what the path can take.
8. It starts with an initial window of about ten packets, roughly 14 kilobytes, and it doubles that each round trip while nothing is lost.
9. So a 100 kilobyte file takes about three round trips of growth, whatever the line speed is. At 200 milliseconds each, that is 600 milliseconds of waiting, not transferring.
10. Every extra round trip in your protocol costs the full latency, every time. DNS is one. The TCP handshake is one. Modern encryption is one more. A redirect is another full set.
11. That is why page load time is latency-bound. A typical page is not one transfer. It is dozens of small transfers, each one paying the round trip again, and many of them cannot start until an earlier one finished.
12. Reducing the number of round trips is therefore the single highest-value performance work there is, and it is usually free.

■ 34.10.5 TECHNICAL – the engineer's version

1. Typical round-trip times from a home broadband connection in India, against the theoretical floor. Measured values are approximate, vary by provider and by time of day, and are given as ranges for that reason.

From India to	Distance	Ideal RTT	Typical RTT
Chennai	1,030 km	10 ms	25 to 35 ms
Bahrain, Gulf	2,420 km	24 ms	35 to 45 ms
Singapore	3,900 km	38 ms	55 to 65 ms
Frankfurt	6,560 km	64 ms	105 to 120 ms
Tokyo	6,720 km	66 ms	120 to 135 ms
London	7,190 km	70 ms	110 to 125 ms
Sydney	10,160 km	99 ms	150 to 165 ms
Oregon, US West	12,690 km	124 ms	215 to 235 ms
Virginia, US East	12,850 km	126 ms	185 to 200 ms
Sao Paulo	13,770 km	135 ms	320 to 360 ms

2. Distances are great-circle from Mumbai. Ideal round trip assumes 204,000 kilometres per second in fibre, which is the speed of light divided by a refractive index of 1.4675, a normal figure for standard single-mode fibre.

3. Note Tokyo and Oregon. Both are further by cable than by map, because the cable routes run through Singapore and around the Pacific rather than over central Asia. The ratio of real to ideal is the honest measure of a route.
4. Bandwidth-delay product, formally: BDP in bytes equals bandwidth in bits per second times round-trip time in seconds, divided by eight. It is the amount of data that must be in flight to keep the pipe full.
5. TCP window scaling is RFC 7323, September 2014, which replaced RFC 1323 of May 1992. The scale factor shifts the 16-bit window left by up to 14 bits, giving a maximum window of 1 gigabyte. It is negotiated in the SYN, so both ends must offer it, and a middlebox that strips the option silently caps your throughput.
6. Initial congestion window of 10 segments is RFC 6928, April 2013. Before it, the standard was 2 to 4 segments, and a small page took twice as many round trips.
7. Loss matters more than bandwidth on long paths. The Mathis equation approximates a single TCP flow's rate as MSS divided by RTT, times one over the square root of the loss probability.
8. Worked: 1,460-byte segments, 200 millisecond round trip, 0.1 percent loss. That gives about 1.85 megabits per second for one connection, on a line of any speed. This is why long-distance transfers use many parallel connections or a loss-tolerant protocol.
9. Queuing delay and bufferbloat: the term was popularized by Jim Gettys in 2010 and 2011. Over-sized, unmanaged buffers in home routers add hundreds of milliseconds of latency under load. The modern fixes are CoDel, RFC 8289, and fq_codel, RFC 8290, both January 2018, plus BBR congestion control from Google in 2016.
10. Protocol round-trip costs, cold start, one round trip written as 1 RTT.

Step	Cost	Note
DNS lookup	1 RTT	0 if cached
TCP handshake	1 RTT	0 with TCP Fast Open
TLS 1.2	2 RTT	RFC 5246, 2008
TLS 1.3	1 RTT	RFC 8446, 2018
QUIC, new	1 RTT	RFC 9000, 2021
QUIC, resumed	0 RTT	replay risk

11. Page weight for scale: the HTTP Archive's public data has put the median page in the mid-2020s at roughly 2.5 megabytes across about 70 requests. Approximate, and it drifts upward every year.
12. Measurement tools, and what each actually measures. `ping` gives ICMP round trip, which some networks deprioritize. `mtr` and `traceroute` give per-hop round trip, which is the time to that hop and back, not a segment time. `iperf3` measures achievable throughput, not latency. `curl -w` measures the phases of a real request, which is what section 34.13 uses.
13. One caution from the reader's own session. Round-trip time to a hop tells you nothing about whether traffic passes through it. The reader's trace reached `ae106-0.rwa04.pnq20.ntwk.msn.net` at hop 12 and then went silent, and the silence was not a latency problem at all.

■ 34.10.6 WORDS - remember these

1. Latency — the wait before anything arrives — one-way or round-trip delay, measured in milliseconds.
2. Round-trip time — there and back — RTT, the interval from sending a packet to receiving its acknowledgement.

3. Bandwidth — how much per second — capacity in bits per second, distinct from achieved throughput.
4. Throughput — what you actually got — measured delivery rate, always less than or equal to bandwidth.
5. Propagation delay — travel time over the distance — distance divided by the signal velocity, about 200,000 km/s in fibre.
6. Serialization delay — time to push bits onto the wire — packet size divided by link rate.
7. Bandwidth-delay product — how much can be in flight — bandwidth times RTT, the window size needed to fill a path.
8. Window scaling — permission to have more in flight — RFC 7323 TCP option raising the maximum window to 1 GiB.
9. Bufferbloat — queues so long they add delay — excessive unmanaged buffering, addressed by CoDel and fq_codel.

34.11 IPv6 properly

■ 34.11.1 PLAIN - in simple words

1. Every device on the internet needs an address. The old kind, IPv4, is four numbers such as 20.207.73.82, and there are about 4.3 billion of them.
2. The world has far more than 4.3 billion connected devices, so we ran out. That was known to be coming in the early 1990s.
3. IPv6 is the replacement. Its addresses are four times longer, written in hexadecimal and separated by colons, like 2001:db8::1.
4. The number of IPv6 addresses is about 340 undecillion, a 3 followed by 38 digits. It is not a bigger pool. It is a pool of a completely different kind.
5. Because addresses are no longer scarce, every device can have its own public address, and the address-sharing trick called NAT becomes unnecessary.
6. IPv6 is not a newer version of IPv4 that IPv4 machines can talk to. It is a separate, parallel internet. A machine with only IPv6 cannot reach a machine with only IPv4.
7. That single fact explains why adoption took thirty years: everyone had to run both, which is more work, not less, until almost everyone has moved.
8. Most machines therefore run **dual stack**: both address families at once, choosing per connection.
9. The reader's own session printed IPv6: (none). We will say exactly what that means, and exactly what it did and did not cause.

■ 34.11.2 PLAIN - a picture in your head

1. IPv4 is a country that issued eight-digit phone numbers, then grew, and ran out of numbers.
2. Its answer was to give one real number to a whole building and put a switchboard in the lobby. Everyone inside dials out through the switchboard, and nobody outside can dial a specific flat directly. That is NAT.
3. IPv6 is a new numbering plan with numbers so long that every room, every appliance and every future appliance can have its own.
4. So the switchboard is no longer needed. Every flat can be dialled directly.
5. But the two numbering plans are not interchangeable. A phone that can only dial old numbers cannot reach a new number at all, and there is no prefix that converts one to the other.
6. So for a long transition every building keeps both systems, and every phone tries the new number first, falling back to the old one.

Where this comparison breaks:

7. Direct dialling is not the same as being open to callers. Removing NAT does not remove the front door. IPv6 home routers still block unsolicited incoming connections by default; the address is reachable, the firewall still decides.
8. And phone numbers are handed out by an office. IPv6 addresses are very often built by the device itself from a prefix the router announces, with no central office involved at all.

■ 34.11.3 PLAIN - a worked example

1. An IPv6 address is 128 bits, written as eight groups of four hexadecimal digits separated by colons.
2. There are exactly two compression rules, and they are always applied in this order.
3. Rule one: in each group, drop leading zeros. Never drop trailing zeros.
4. Rule two: replace one run of consecutive all-zero groups with `::`. You may do this only once in an address, because otherwise it would be ambiguous.

```
full      2001:0db8:0000:0000:0008:0800:200c:417a
rule 1    2001:db8:0:0:8:800:200c:417a
rule 2    2001:db8::8:800:200c:417a
```

5. A second example, the loopback address, which is what `127.0.0.1` is in the new world.

```
full      0000:0000:0000:0000:0000:0000:0000:0001
rule 1    0:0:0:0:0:0:0:1
rule 2    ::1
```

6. A third, the unspecified address, which is what `0.0.0.0` is in the new world. All 128 bits are zero, so the whole thing compresses to `::`.
7. A fourth, showing the “only once” rule. This address has two separate zero runs.

```
full      2001:0db8:0000:0000:0001:0000:0000:0001
right     2001:db8::1:0:0:1
also ok   2001:db8:0:0:1::1
wrong     2001:db8::1::1      (ambiguous, illegal)
```

8. The preferred form, from RFC 5952, says compress the longest run, use lowercase letters, and do not use `::` for a single zero group. So `2001:db8::1:0:0:1` is the correct spelling of that one.
9. In a web address you must put the address in square brackets, because the colon already means “port”.

```
https://[2606:4700:4700::1111]/      port 443
https://[2606:4700:4700::1111]:8443/  port 8443
```

10. Prefix lengths work exactly as in IPv4. /64 is the normal size of one local network, and it contains 18.4 quintillion addresses. Your home is usually delegated a /56, which is 256 such networks.

■ 34.11.4 PLAIN - what is really happening inside

1. When an IPv6 machine joins a network, it does not wait to be given an address. It makes one, checks it, and starts using it.
2. First it builds a link-local address beginning `fe80::`. Every IPv6 interface always has one, and it works even with no router present.
3. Then it checks nobody else has that address, by asking the network directly. This is duplicate address detection.
4. Then it sends a router solicitation, a short message meaning “is there a router here, and what prefix do we use?”.

5. The router replies with a router advertisement carrying the prefix, for example 2401:4900:1f3f:abcd::/64, and some flags.
6. The machine appends its own 64-bit interface part to that prefix and now has a global address. This is SLAAC, stateless address autoconfiguration, and no server was involved.
7. Because that interface part used to be derived from the hardware address, it followed you between networks and could track you. Modern systems therefore generate random, periodically changing addresses instead.
8. A machine normally ends up with several addresses at once: one link-local, one stable global, one or more temporary globals, and possibly one from DHCPv6. This is normal and not a fault.
9. ARP does not exist in IPv6. Its job, turning an address into a hardware address on the local wire, is done by neighbour discovery, which uses ICMPv6 messages sent to a multicast group rather than a broadcast.
10. That is a real improvement. An ARP broadcast interrupts every device on the network. A neighbour solicitation is sent to a solicited-node multicast group that typically only the target listens to.
11. A dual-stack machine looking up a name asks for both an A record, which is IPv4, and an AAAA record, which is IPv6, at the same time.
12. If both come back, it starts connecting over IPv6, and if that has not succeeded within a short delay it starts an IPv4 attempt in parallel and uses whichever finishes first. That is Happy Eyeballs.
13. The user never sees which family won. That is the entire point of the design, and it is why broken IPv6 in 2012 caused visible page hangs and broken IPv6 today usually does not.

■ 34.11.5 TECHNICAL – the engineer’s version

1. IPv6 was first specified in RFC 1883, December 1995, by Steve Deering and Bob Hinden, revised as RFC 2460 in December 1998, and is now RFC 8200, July 2017, an Internet Standard, STD 86.
2. Address architecture is RFC 4291, February 2006. Canonical text representation is RFC 5952, August 2010. The documentation prefix 2001:db8::/32 is RFC 3849 and is the only prefix you should use in examples.
3. Address types you must recognize on sight.

Type	Prefix	IPv4 counterpart
Global unicast	2000::/3	public address
Link-local	fe80::/10	169.254.0.0/16
Unique local	fc00::/7	10.0.0.0/8
Multicast	ff00::/8	224.0.0.0/4
Loopback	::1/128	127.0.0.1
Unspecified	::/128	0.0.0.0

4. There is no broadcast address in IPv6 at all. Everything broadcast used to do is done with multicast groups such as ff02::1, all nodes on the link, and ff02::2, all routers on the link.
5. Unique local addresses are RFC 4193, October 2005. In practice only fd00::/8 is used, with a randomly generated 40-bit global identifier, so that two merged networks do not collide. They are not routable on the public internet.
6. Link-local addresses need a zone index when you use them, because fe80::1 is ambiguous across interfaces. Write fe80::1%en0 on macOS or fe80::1%eth0 on Linux.

7. SLAAC is RFC 4862. Router advertisement is ICMPv6 type 134, router solicitation is type 133. The A flag on a prefix option permits autoconfiguration; the M flag says use DHCPv6 for addresses; the O flag says use DHCPv6 only for other information such as DNS servers.
8. Interface identifiers were originally modified EUI-64, built by splitting the 48-bit MAC address, inserting ff:fe in the middle and flipping the seventh bit. Because that leaks the hardware address, RFC 8981, February 2021, which replaced RFC 4941, defines temporary addresses, and RFC 7217 defines stable opaque identifiers that differ per network.
9. DHCPv6 is RFC 8415, November 2018. It provides stateful assignment and, more importantly for home networks, prefix delegation, by which a router requests a /56 or /48 for the networks behind it. A long-standing practical problem is that Android has never implemented DHCPv6 address assignment, so any network that offers only DHCPv6 leaves Android phones without IPv6.
10. Neighbour discovery is RFC 4861, September 2007. Neighbour solicitation is ICMPv6 type 135, advertisement is 136. The solicited-node multicast address is ff02::1:ff followed by the low 24 bits of the target address. It also provides duplicate address detection and neighbour unreachability detection, which ARP never had.
11. Because so much of IPv6 depends on ICMPv6, blanket-blocking ICMP breaks IPv6 completely. RFC 4890 lists which types must be permitted. This is a common self-inflicted outage.
12. Happy Eyeballs is RFC 6555, April 2012, by Dan Wing and Andrew Yourtchenko, replaced by RFC 8305, December 2017, by David Schinazi and Tommy Pauly. The recommended resolution delay is 50 milliseconds and the recommended connection attempt delay is 250 milliseconds, with a 2 second minimum fallback. A version 3 is being written in the IETF HAPPY working group; revision 04 is dated 2 July 2026 and it is not an RFC yet.
13. Transition and exhaustion dates, which explain the pace better than any argument.

Event	Date
IANA free pool exhausted	3 February 2011
APNIC exhausted, Asia	15 April 2011
World IPv6 Launch	6 June 2012
RIPE NCC exhausted	14 September 2012
ARIN exhausted	24 September 2015
Google IPv6 passes 50%	28 March 2026

14. The honest reasons adoption took thirty years, in order of weight. IPv6 is not backward compatible, so the first mover pays and gains nothing. NAT and CIDR, RFC 1519 in 1993, relieved the pressure that was supposed to force the move. Carrier-grade NAT let ISPs grow without addresses. Every firewall, monitoring tool, log parser and internal script had to be redone. And a resale market in IPv4 addresses, at roughly 30 to 50 US dollars each in recent years, made staying put affordable.
15. Where it did happen, it happened fast and for a reason: mobile networks built after 2012 had no address supply, so they went IPv6-first with translation at the edge. Reliance Jio in India, launched in 2016, is the largest example, which is why India's IPv6 capability is around 72 percent, near the top of the world table, while its fixed broadband lags.
16. Now the reader's own line, IPv6: (none). What it means precisely: the machine had no usable global IPv6 address and therefore no IPv6 default route. Almost certainly the router advertised no IPv6 prefix, which is normal on IPv4-only home broadband behind carrier-grade NAT. The trace's private hops 172.31.0.17, 172.26.22.235 and 172.16.18.33 are consistent with exactly that kind of network.

17. What it did not cause: the failure. `github.com` publishes no AAAA record. Checked again on 13 August 2026, `dig AAAA github.com` returns nothing while `dig A github.com` returns an address. So even a perfect dual-stack machine would have used IPv4 for that request.
18. What it did remove: a second chance. On a dual-stack host talking to a dual-stack site, Happy Eyeballs races both families, and a silent drop on one is often invisible because the other completes. With no IPv6 there was exactly one path, and when that path went silent there was nothing to fall back to.
19. It also simplified the diagnosis, which is a genuine benefit. There was one address, one path, one failure, and `curl -v` naming `20.207.73.82` was the complete story.
20. Observation commands: `ip -6 addr` and `ip -6 route` on Linux, `ifconfig` and `netstat -rn -f inet6` on macOS, `ip -6 neigh` or `ndp -an` for the neighbour cache, `ping6 -c3 ff02::1%en0` to see every IPv6 host on the link, `dig AAAA example.com` for records, and `curl -6` or `curl -4` to force a family.

■ 34.11.6 WORDS – remember these

1. IPv6 — the long-address internet — 128-bit addressing defined by RFC 8200, not backward compatible with IPv4.
2. Dual stack — both at once — a host running IPv4 and IPv6 simultaneously and selecting per connection.
3. Link-local — an address that works with no router — `fe80::/10`, valid only on one link, requiring a zone index.
4. Unique local — private addressing for IPv6 — `fc00::/7`, in practice `fd00::/8` with a random global identifier, RFC 4193.
5. SLAAC — the device builds its own address — stateless autoconfiguration from a router-advertised prefix, RFC 4862.
6. Router advertisement — the router announcing the prefix — ICMPv6 type 134, carrying prefix, flags and lifetimes.
7. Neighbour discovery — IPv6's replacement for ARP — ICMPv6 types 133 to 137, using solicited-node multicast instead of broadcast.
8. AAAA record — the IPv6 version of an A record — a DNS record mapping a name to a 128-bit address.
9. Happy Eyeballs — try both families and take the winner — the connection racing algorithm of RFC 8305.
10. Prefix delegation — your router being given a block — DHCPv6-PD handing a site a /56 or /48 to subnet internally.

34.12 Multicast, broadcast and mDNS

■ 34.12.1 PLAIN – in simple words

1. There are three ways to address a message on a network, and every beginner meets all three without being told they are different.
2. **Unicast** is one to one. You address one machine, and only that machine receives it. This is almost all traffic.
3. **Broadcast** is one to everybody on this local network. Every device receives it and must look at it, even if it does not care.
4. **Multicast** is one to a named group. Only devices that joined the group receive it. Everyone else ignores it, ideally without even being interrupted.

5. Broadcast never crosses a router. It stops at the edge of your local network, and that is by design, or the internet would drown.
6. Multicast can in principle cross routers, and inside company and provider networks it does. Across the public internet it essentially never does.
7. Broadcast is how your machine gets an address when it joins a network, and how it finds the hardware address of its neighbours.
8. Multicast is how devices announce what they are: printers, speakers, televisions, game consoles, other laptops.
9. **mDNS** is ordinary DNS, done over multicast, with no server anywhere. It is why names ending in `.local` work, and why your phone finds a printer you never configured.

■ 34.12.2 PLAIN – a picture in your head

1. Picture a classroom full of people.
2. Unicast is walking over and speaking to one person quietly.
3. Broadcast is standing at the front and shouting to the whole room. Everybody must stop and listen long enough to decide whether it concerns them.
4. Multicast is saying “this is for the chess club only”. You still say it out loud, but only the chess club is expected to pay attention.
5. mDNS is somebody shouting “does anyone here have a printer?”, a printer shouting back “I do, and here is where to find me”, and everyone else in the room writing that down for later because they heard it anyway.
6. That last detail is real and clever. Answers are shouted, not whispered, so every device updates its notes from one exchange.

Where this comparison breaks:

7. A shout does not leave the room, and neither does broadcast. But many cheap network switches treat multicast exactly like a shout and send it to every port anyway, which defeats the point. Better switches watch who joined which group and only forward accordingly.
8. And a classroom has one room. A home with a guest network, or an office with several VLANs, is several rooms with a closed door between them, which is why discovery mysteriously stops working the moment somebody adds a second network.

■ 34.12.3 PLAIN – a worked example

1. Finding a printer on a home network, with no configuration at all.

```

1 laptop asks   PTR  _ipp._tcp.local
2 printer says  PTR  Office Printer._ipp._tcp.local
3 laptop asks   SRV  and TXT for that name
4 printer says  SRV  port 631, host prn9010.local
5 printer says  TXT  rp=ipp/print, pdl=image/urf
6 laptop asks   A    for prn9010.local
7 printer says  A    192.168.0.31
8 laptop prints to 192.168.0.31 port 631

```

2. Every one of those questions and answers went to the multicast group `224.0.0.251` on UDP port `5353`. No DNS server was involved at any point.
3. You can watch all of it happen. On macOS the tool is built in.

```

dns-sd -B _services._dns-sd._udp local.  # what kinds exist
dns-sd -B _ipp._tcp                      # printers
dns-sd -B _airplay._tcp                  # AirPlay devices

```



```
dns-sd -L "Office Printer" _ipp._tcp      # full details
ping prn9010.local                        # name to address
```

4. On Linux the equivalent comes from Avahi.

```
avahi-browse -art
avahi-resolve -n prn9010.local
```

5. Service names follow a fixed shape: an instance name, an underscore service type, an underscore protocol, then local.

Service type	What it is
_ipp._tcp	printing
_airplay._tcp	AirPlay video
_raop._tcp	AirPlay audio
_googlecast._tcp	Chromecast
_ssh._tcp	SSH servers
_hap._tcp	HomeKit accessories

6. So “it just works” is not magic. It is four record types, one multicast group and one well-known port.

■ 34.12.4 PLAIN – what is really happening inside

- Broadcast at the hardware level means sending to the Ethernet address `ff:ff:ff:ff:ff:ff`. Every network card accepts a frame with that destination and passes it up.
- That is why ARP and the first steps of DHCP use broadcast: at that moment the sender does not know who to ask, or does not have an address yet.
- Multicast at the hardware level uses a different set of Ethernet addresses, derived from the group address. A card is told which of them to accept, and it silently discards the rest without waking the processor.
- That is the real saving. Broadcast costs every device time. Multicast costs only the members, provided the switch and the network card do their part.
- Devices announce group membership so switches and routers know where to send. In IPv4 this is IGMP; in IPv6 it is MLD. Switches that listen in on those messages are said to do snooping, and that is what stops multicast becoming broadcast.
- mDNS uses this. Every participating device joins one group, listens on one port, and answers questions about itself only.
- Answers are sent to the group, not to the asker, so every listener can cache them. A busy network therefore needs far fewer queries than you would expect.
- Names are claimed, not assigned. A device probes its desired name three times, and if someone else answers, it renames itself, typically by adding a number. That is where `MacBook-2.local` comes from.
- Records carry a flag meaning “replace what you have”, so corrections propagate. A device leaving politely sends its records with a lifetime of zero, which is a goodbye.
- The packets are deliberately built so they do not travel. Their time-to-live is set so a router will not forward them, and the group is a link-local group. mDNS is not designed to cross networks, and does not.
- That is the single biggest source of “it worked at home and not at the office”. Two subnets, or a guest network with client isolation turned on, and discovery stops instantly with no error message.

12. A full-tunnel VPN can do the same thing, by capturing name lookups or the default route. On the reader's macOS machine, `utun` interfaces were present, and that class of interface is exactly where such interference would appear.

■ 34.12.5 TECHNICAL – the engineer's version

1. The addresses worth memorizing.

Address	Meaning
255.255.255.255	limited broadcast
224.0.0.1	all hosts on this link
224.0.0.2	all routers on this link
224.0.0.251	mDNS over IPv4
ff02::fb	mDNS over IPv6
239.255.255.250	SSDP and WS-Discovery

2. IPv4 multicast is `224.0.0.0/4`. Within it, `224.0.0.0/24` is link-local control traffic that routers never forward, and `239.0.0.0/8` is administratively scoped for private use, defined in RFC 2365, July 1998.
3. IPv6 has no broadcast at all. `ff02::1` is all nodes on the link and `ff02::2` is all routers, and scope is encoded in the address itself.
4. Ethernet mapping: an IPv4 group maps to a MAC address of `01:00:5e` plus the low 23 bits of the group. Because 32 different groups share one MAC address, a card can accept traffic it did not want, and software must filter again.
5. Group membership signalling: IGMPv1 is RFC 1112, August 1989, IGMPv2 is RFC 2266, November 1997, IGMPv3 is RFC 3376, October 2002, which added source-specific multicast. For IPv6 the equivalent is MLDv2, RFC 3810, June 2004.
- IGMP snooping guidance is RFC 4541, May 2006, and is informational, which is why switch behaviour varies so much.
6. Discovery protocols on a normal home network.

Protocol	Port	Group
mDNS	UDP 5353	224.0.0.251
LLMNR	UDP 5355	224.0.0.252
SSDP	UDP 1900	239.255.255.250
WS-Discovery	UDP 3702	239.255.255.250

7. mDNS is RFC 6762 and DNS Service Discovery is RFC 6763, both February 2013, both by Stuart Cheshire and Marc Krochmal of Apple. Apple shipped the implementation as Rendezvous in Mac OS X 10.2 in August 2002 and renamed it Bonjour in 2005 after a trademark dispute. Avahi is the Linux equivalent, first released in 2005. Windows has shipped a native mDNS responder since Windows 10.
8. `.local` is reserved by RFC 6762 for mDNS. Using `.local` as an internal Active Directory domain name, which was once common advice, produces a permanent low-level conflict on any network with Apple devices.
9. Service enumeration uses the meta-query `_services._dns-sd._udp.local`, which returns a PTR record per service type present. Then PTR gives instances, SRV gives host and port, TXT gives

key-value parameters, and A or AAAA gives the address. Four record types, in that order, every time.

10. SSDP is part of UPnP, which Microsoft introduced in 1999. It uses HTTP-like text over UDP with M-SEARCH and NOTIFY verbs. It has been widely abused for reflection and amplification attacks, which is why UDP 1900 should never be reachable from the internet.
11. WS-Discovery on UDP 3702 is what modern Windows and modern network printers use after Microsoft deprecated the old NetBIOS browsing and SMBv1.
12. The Wi-Fi cost, which surprises people. Multicast and broadcast frames are normally sent at the lowest basic rate so every client can hear them, and they are not acknowledged. On a busy access point, chatty discovery traffic therefore consumes disproportionate airtime. Enterprise access points respond by converting multicast to unicast, proxying mDNS, or dropping it, and each of those changes behaviour in a different way.
13. Multicast routing across the public internet was specified, deployed experimentally as the MBone from 1992, and never became general service. PIM Sparse Mode is RFC 7761, March 2016, and is used inside provider networks for IPTV and inside exchanges for market data, but you cannot assume any multicast reaches the internet.
14. Observation: `tcpdump -ni en0 udp port 5353` shows the traffic, `dns-sd -B` browses, `avahi-browse -art` dumps everything, `netstat -gn` or `ip maddr` show lists joined groups, and `ping -c3 224.0.0.1` reveals which hosts on the link answer at all.

■ 34.12.6 WORDS – remember these

1. Unicast — one to one — a packet addressed to a single interface.
2. Broadcast — one to all on this link — IPv4 only, 255.255.255.255 or the subnet's all-ones address, never forwarded by routers.
3. Multicast — one to a group — delivery to hosts that joined a group address in 224.0.0.0/4 or ff00::/8.
4. IGMP — how a host joins a group — Internet Group Management Protocol, RFC 3376 for version 3; MLD is the IPv6 equivalent.
5. Snooping — the switch learning who joined — IGMP or MLD snooping, so multicast is not flooded to every port.
6. mDNS — DNS with no server — RFC 6762 multicast DNS on UDP 5353 serving the .local namespace.
7. DNS-SD — how services are named and found — RFC 6763 conventions using PTR, SRV and TXT records over unicast or multicast DNS.
8. Bonjour — Apple's name for it — the Apple implementation of mDNS and DNS-SD, shipped as Rendezvous in 2002.
9. SSDP — the older discovery protocol — UPnP's search mechanism on UDP 1900, a frequent amplification-attack vector.

34.13 Network performance for developers

■ 34.13.1 PLAIN – in simple words

1. When a page feels slow, “the network is slow” is not a diagnosis. It is a place to start looking.
2. Every web request is made of five separate waits, and they have completely different causes and fixes.
3. **DNS**: turning the name into an address. Slow if the resolver is far away or the record is not cached.
4. **Connect**: the handshake that opens the connection. Costs one round trip, always, no exceptions.

5. **TLS:** agreeing on encryption. Costs one more round trip on modern versions, two on older ones.
6. **Time to first byte:** you asked, and you are waiting for the server to think. This includes one more round trip plus the server's own work.
7. **Transfer:** the bytes arriving. This is the only part that bandwidth actually improves.
8. Most slow requests are the first four. Most "optimizations" people try improve only the fifth.
9. The good news is that you can measure all five with one command, on any machine, in one line, and stop guessing forever.

■ 34.13.2 PLAIN - a picture in your head

1. Think of telephoning a shop in another city to ask whether they have an item.
2. Looking up the number is DNS.
3. Dialling and waiting for someone to pick up is the connection handshake.
4. Agreeing that you will both speak in a private code, and checking the other side really is that shop, is TLS.
5. Asking your question and waiting while they walk to the back room is time to first byte.
6. Them reading the answer aloud is the transfer.
7. If the shop is far away, every single exchange in that conversation takes longer, because every sentence travels the distance twice.
8. So the way to make the call quick is to ask fewer questions, not to speak faster.

Where this comparison breaks:

9. A phone call is one conversation. Loading one page opens dozens, and older protocols could only ask one question per line at a time, which is exactly why browsers used to open six lines to the same shop.
10. And you can keep a phone line open for the next question. Programs often fail to, hanging up and redialling for every item, which is the single most common self-inflicted performance bug.

■ 34.13.3 PLAIN - a worked example

1. curl can print the exact time each phase finished. Put the format in a file, because it is easier to read.

```
cat > fmt.txt <<'EOF'
  dns  %{time_namelookup}s
 conn  %{time_connect}s
  tls  %{time_appconnect}s
 send  %{time_pretransfer}s
 ttfb  %{time_starttransfer}s
 total  %{time_total}s  code %{http_code}
EOF
curl -sS -o /dev/null -w @fmt.txt https://github.com/
```

2. A healthy result from an Indian connection to a nearby edge looks like this.

```
  dns  0.028s
 conn  0.081s
  tls  0.190s
 send  0.191s
 ttfb  0.352s
 total 0.404s  code 200
```

3. Those numbers are cumulative, all measured from the start. The useful quantities are the differences.

Phase	Arithmetic	Time
DNS	namelookup	28 ms
TCP handshake	conn minus dns	53 ms
TLS handshake	tls minus conn	109 ms
Server thinking	ttfb minus send	161 ms
Download	total minus ttfb	52 ms

4. Read it. The round trip to this server is about 53 milliseconds, because that is what a TCP handshake costs. TLS cost roughly two of those. Actual data transfer was 52 milliseconds out of 404.
5. So 87 percent of that request was setup and waiting. Buying a faster line improves the 52 milliseconds and nothing else.
6. Now the same command during the reader's own outage. The shape of the failure is the diagnosis.

```

dns 0.026s
conn 0.000s
tls 0.000s
ttfb 0.000s
total 15.001s code 000

```

7. DNS succeeded, so the name resolved to 20.207.73.82 correctly. Then every later phase is zero, because none of them was ever reached.
8. A zero in a later phase does not mean it was fast. It means it never happened. code 000 means no HTTP response existed at all.
9. That is the signature of a silent drop: the connection phase never completed and nothing came back, not a refusal, not an error. Compare it with a refused connection, which fails in a few milliseconds with code 000 and a different curl exit code, and with a slow server, which shows a large ttfb and a normal code 200.
10. Three different faults, three different shapes, one command.

■ 34.13.4 PLAIN – what is really happening inside

1. The first fix is to stop paying the setup cost repeatedly. If you close the connection after each request, the next request pays DNS, TCP and TLS again.
2. Keeping the connection open is called keep-alive, and it has been the default in HTTP since version 1.1. But a program that creates a new client object per request defeats it, and this is very common in application code.
3. The second fix is to stop queuing. In HTTP/1.1, one connection carries one request at a time. Browsers worked around it by opening about six connections per host, which multiplies handshakes and confuses congestion control.
4. HTTP/2 solves this properly by carrying many requests inside one connection as interleaved streams. One handshake, one congestion window, dozens of requests in flight.
5. Header compression helps too. The same cookies and headers repeat on every request; HTTP/2 sends them once and refers back to them afterwards.
6. But HTTP/2 has one honest weakness. All those streams still ride one TCP connection, so a single lost packet stalls every stream until it is resent. That is head-of-line blocking, moved down a layer rather than removed.
7. HTTP/3 fixes that by running over QUIC on UDP, where each stream is recovered independently, and a loss affecting one stream does not stall the others.
8. The third fix is compression. Text compresses enormously, so a compressed response transfers in a fraction of the time. It costs a little processor time on both ends and nothing else.

9. Compression only shortens the transfer phase. It does not remove a single round trip, which is why it helps large responses and barely touches small ones.
10. The fourth fix is doing setup earlier. A browser can be told to resolve a name, or open a connection, before it knows it will need it, so the cost is paid during time the user was going to spend anyway.
11. The last and largest fix is asking for less. A request you do not make costs nothing and cannot fail.

■ 34.13.5 TECHNICAL – the engineer’s version

1. The curl write-out variables, all in seconds from the start of the transfer.

Variable	Marks the moment
time_namelookup	name resolution done
time_connect	TCP handshake done
time_appconnect	TLS handshake done
time_pretransfer	about to send request
time_starttransfer	first response byte
time_total	transfer complete

2. Useful companions: `%{http_version}, %{num_connects}, %{num_redirects}, %{time_redirect}, %{size_download}, %{speed_download}` and `%{remote_ip}`. `curl -w` also accepts `%{json}` in version 7.70.0 and later, which prints every variable as one JSON object.
3. A one-line version worth keeping in your shell history: `curl -so /dev/null -w 'dns %{time_namelookup} tls %{time_appconnect} ttfb %{time_starttransfer} tot %{time_total}\n' https://example.com/.`
4. Connection reuse. HTTP/1.1, RFC 2068 in January 1997, made persistent connections the default. In practice the wins come from client configuration: reuse one HTTP client object, set a connection pool size, and keep idle connections alive for longer than your request interval.
5. HTTP/2 is RFC 7540, May 2015, restated as RFC 9113, June 2022. It requires ALPN negotiation during the TLS handshake, uses HPACK header compression, RFC 7541, and multiplexes streams over one connection. Server push was part of it, was found not to help, and Chrome removed support in version 106 in October 2022.
6. HTTP/3 is RFC 9114, June 2022, over QUIC, RFC 9000, May 2021, with QPACK header compression, RFC 9204. It removes transport-level head-of-line blocking, survives a change of network address through connection identifiers, and includes the TLS 1.3 handshake in its own setup.
7. Compression, with real behaviour. gzip is RFC 1952 and is universal. Brotli is RFC 7932, July 2016, and typically produces files 15 to 20 percent smaller than gzip on JavaScript and CSS at its highest setting, at a much higher compression cost, so it suits static assets compressed once. Zstandard is RFC 8878, February 2021, and its value is speed at a similar ratio, which suits dynamic responses; browser support arrived in Chrome 123 in March 2024 and Firefox 126 in May 2024.
8. Never compress content that is already compressed, such as images, video or archives. You spend processor time to add a few bytes.
9. TLS costs, precisely. TLS 1.2, RFC 5246, needs two round trips. TLS 1.3, RFC 8446, August 2018, needs one, and a resumed session can send data in the first flight, which is 0-RTT, at the price of replay exposure for non-idempotent requests. Session resumption tickets are what make the second visit cheap.
10. Browser hints that buy round trips back, all standard HTML today: `dns-prefetch` resolves a name early, `preconnect` completes DNS, TCP and TLS early, `preload` fetches a known-critical asset early, and `Early Hints`, status code 103 from RFC 8297, December 2017, lets the server send those hints before the real response is ready.

11. Server-side measurement should be exported, not guessed. The Server-Timing response header, a W3C specification, carries named durations from the server into the browser's own performance timeline.
12. Rough real-world weights for a typical page on a 200 millisecond path, approximate and workload-dependent.

Change	Typical effect
Connection reuse	removes 2 RTT per request
HTTP/1.1 to HTTP/2	removes most queueing
gzip or brotli on text	60 to 80 percent smaller
CDN edge for static	removes most of the RTT
One fewer redirect	removes a full request

13. Measurement discipline. Test from where the users are, not from the datacentre. Test with a cold cache and a warm cache separately. Test the 99th percentile, because the mean hides the failures. And prefer real user monitoring over one laptop's opinion.

■ 34.13.6 WORDS - remember these

1. Time to first byte — how long before anything arrives — the interval from request sent to first response byte, including one round trip.
2. Keep-alive — do not hang up — HTTP persistent connections, default since HTTP/1.1.
3. Multiplexing — many requests, one connection — interleaved streams within a single HTTP/2 or HTTP/3 connection.
4. Head-of-line blocking — one stuck item stalls the rest — a lost segment delaying all streams sharing a TCP connection.
5. ALPN — deciding which HTTP version during the handshake — Application-Layer Protocol Negotiation, a TLS extension, RFC 7301.
6. Brotli — a stronger text compressor — RFC 7932 encoding, advertised as br in Accept-Encoding.
7. Preconnect — pay the setup before you need it — a resource hint that performs DNS, TCP and TLS for a host in advance.
8. 0-RTT — sending data in the first packet — TLS 1.3 early data on a resumed session, with replay risk.

34.14 Cloud networking preview

■ 34.14.1 PLAIN - in simple words

1. Everything in this chapter also exists inside a cloud account, with different names and a control panel instead of a cable.
2. A **VPC**, a virtual private cloud, is your own private network inside the provider's network. Nobody else's machines are in it.
3. A **subnet** is a slice of that network, usually tied to one physical zone.
4. A **route table** decides where traffic from a subnet goes. A subnet is called public or private purely because of what its route table says.
5. A **security group** is a firewall attached to a machine, and it remembers connections it allowed.
6. A **network ACL** is a firewall attached to a subnet, and it remembers nothing, so you must write both directions yourself.
7. A **load balancer** is a managed reverse proxy: one address in front of many machines.

8. **Private link** lets you reach a service without your traffic ever touching the public internet.
9. Chapter 43 builds all of this properly. This section is only so the names are not new when you get there.

■ 34.14.2 PLAIN – a picture in your head

1. A VPC is a floor of an office building that you have rented.
2. Subnets are the rooms on that floor, each in a different fire zone.
3. The route table is the sign at each room door saying which corridor to use to leave.
4. A security group is a lock on each desk. A network ACL is a lock on the room door.

Where this comparison breaks:

5. A door lock does not care whether you are leaving or entering as a reply to something. A security group does, and a network ACL does not, and that single difference causes most cloud firewall confusion.

■ 34.14.3 PLAIN – a worked example

1. A minimal two-tier layout, written as a plan.

```
VPC          10.0.0.0/16
public subnet 10.0.1.0/24 -> route 0.0.0.0/0 to IGW
private subnet 10.0.2.0/24 -> route 0.0.0.0/0 to NAT
load balancer in public subnet, port 443
app servers in private subnet, port 8080
database in private subnet, port 5432
```

2. The security groups, written as sentences rather than tables.

```
sg-lb  allow in 443 from 0.0.0.0/0
sg-app allow in 8080 from sg-lb
sg-db  allow in 5432 from sg-app
```

3. Note what is missing. No outbound rules were needed, because security groups are stateful, so the reply to an allowed request is allowed automatically.
4. Note also that rules refer to other groups, not to addresses. That is the feature worth learning first, because it survives machines being replaced.

■ 34.14.4 PLAIN – what is really happening inside

1. None of this is a physical network. It is a software overlay running on the provider's real network, and the encapsulation is exactly the tunnelling from section 34.8.
2. Your packets are wrapped, carried across shared hardware, and unwrapped, and the wrapping is what keeps your VPC separate from everyone else's.
3. A security group is enforced at the virtual network interface of the machine itself, before the packet reaches the guest, which is why the machine's own firewall never sees a blocked packet.
4. A network ACL is enforced at the subnet boundary, so traffic between two machines in the same subnet never meets it.
5. Because security groups only allow and never deny, they cannot express "allow everyone except this address". Network ACLs can, which is the main reason they still exist.
6. A private link endpoint puts an address from your own subnet in front of somebody else's service, so the traffic stays inside the provider's network and never needs an internet gateway at all.

■ 34.14.5 TECHNICAL – the engineer’s version

1. Amazon launched Virtual Private Cloud in August 2009, and made a default VPC automatic for new accounts in 2013. Azure calls it a virtual network, and Google calls it a VPC network, which is global rather than regional.
2. The comparison that matters most in interviews and in incidents.

Property	Security group	Network ACL
Attached to	an interface	a subnet
Stateful	yes	no
Rule types	allow only	allow and deny
Evaluation	all rules	numbered, first match

3. AWS reserves five addresses in every subnet: the network address, the VPC router, the DNS address at the second host address, one reserved for future use, and the broadcast address. So a /24 gives 251 usable addresses, not 254.
254. VPC CIDR blocks may be /16 to /28.
4. Load balancer types, with dates. The Classic Load Balancer arrived in 2009, the Application Load Balancer, which is layer 7 and understands HTTP, in August 2016, the Network Load Balancer, layer 4, in September 2017, and the Gateway Load Balancer, which uses Geneve encapsulation, in November 2020.
5. AWS PrivateLink was announced in November 2017. Gateway endpoints for object storage pre-date it. The distinction to remember is that an interface endpoint consumes an address in your subnet and a gateway endpoint is a route table entry.
6. Connectivity between networks: peering is non-transitive, so three peered VPCs do not form a mesh; Transit Gateway, announced November 2018, exists precisely to solve that.
7. Egress costs money and ingress usually does not, which is a commercial fact with architectural consequences, and it is why cross-zone chatter shows up on bills.
8. Observation: `aws ec2 describe-security-groups`, `aws ec2 describe-route-tables`, and VPC Flow Logs, which record accepted and rejected flows and are the cloud equivalent of the packet capture you would otherwise take.
9. Chapter 43 covers the whole of this properly, including NAT gateways, IPv6-only subnets, and how a request reaches a container.

■ 34.14.6 WORDS – remember these

1. VPC — your own private network in the cloud — a logically isolated virtual network with your own address range.
2. Subnet — a slice of that network in one zone — a CIDR range bound to an availability zone with its own route table association.
3. Route table — the sign saying which way out — a set of destination-to-target rules applied to a subnet.
4. Security group — a stateful firewall on the machine — allow-only rules evaluated at the elastic network interface.
5. Network ACL — a stateless firewall on the subnet — numbered allow and deny rules evaluated in order, in both directions.
6. Load balancer — one public address, many servers — a managed layer 4 or layer 7 reverse proxy with health checks.

7. Private link — reach a service without the internet — an endpoint placing a provider service on an address inside your own subnet.

34.98 Common wrong ideas

1. Wrong: 127.0.0.1 and 0.0.0.0 are two ways of saying the same thing. Right: 127.0.0.1 means this machine only, and listening on 0.0.0.0 means every interface the machine has, which is the opposite of private.
2. Wrong: a firewall makes a machine secure. Right: a firewall only decides which packets are allowed to arrive. The service listening behind it must still be safe on its own, because allowed traffic is not checked.
3. Wrong: DROP and REJECT are the same, since both block the packet. Right: REJECT sends a refusal and fails in milliseconds, DROP says nothing and fails at your timeout. The reader's 15 seconds of silence was the DROP signature.
4. Wrong: * * * at the end of a traceroute proves the traffic is blocked there. Right: many routers deprioritize or never send ICMP time-exceeded replies, so silence at the end of a trace is normal and proves nothing.
5. Wrong: a VPN makes you anonymous. Right: it moves the point where your traffic becomes visible, from your ISP to your VPN provider, and it changes nothing about cookies, logins or browser fingerprints.
6. Wrong: a proxy handling your HTTPS traffic can read it. Right: after a CONNECT it copies bytes blindly and sees the hostname, sizes and timing only, unless it has been made a trusted certificate authority on your machine.
7. Wrong: a CDN makes everything faster. Right: it makes cacheable things faster and a first miss slightly slower, and it does nothing at all for a response that is different for every user.
8. Wrong: anycast is a kind of DNS load balancing. Right: anycast is one address announced from many places by BGP, decided by routers. DNS steering is different addresses given to different askers, decided by a name server.
9. Wrong: a slow website is fixed by buying more bandwidth. Right: page load is usually latency-bound, so it is fixed by removing round trips, moving content closer, and reusing connections.
10. Wrong: IPv6 is IPv4 with more numbers, so machines can talk to each other across the two. Right: they are separate protocols with no interoperability, which is why dual stack and Happy Eyeballs exist.

34.99 Chapter summary in 20 lines

1. 127.0.0.1 is an address meaning this machine, localhost is a name for it, and 0.0.0.0 is a placeholder that means all interfaces when listening and unknown when sending.
2. A service bound to loopback is invisible from every other machine, and that is the address behaving correctly, not a firewall.
3. Ports are 16-bit numbers, split by convention into well known below 1024, registered to 49151, and ephemeral above that, and a busy machine exhausts the ephemeral range long before it runs out of anything else.
4. A firewall decides which packets are allowed. It does not inspect what an allowed packet then does, and it is not a substitute for a safe service.
5. Stateless filtering judges each packet alone; stateful filtering remembers connections, which is why you write one rule instead of two.

6. REJECT refuses immediately and produces an instant error. DROP says nothing and produces a timeout, which is why the reader's `curl` sat silent for 15 seconds against `20.207.73.82` and then gave up.
7. The same request succeeding over mobile data proved the server was up and the fault lay on one path, while the trace ending in `* * *` after `ae106-0.rwa04.pnq20.ntwk.msn.net` proved nothing at all on its own.
8. Middleboxes sit between every client and server, and each one can see a different amount: addresses, ports, hostnames from SNI, or full content if it terminates encryption.
9. A VPN encapsulates whole IP packets at layer 3, so every program is covered, and macOS shows those tunnels as `utun` interfaces.
10. A proxy works at layer 5 to 7 for the connections that were configured to use it, so a forward proxy hides clients and a reverse proxy hides servers.
11. Tunnelling is putting one packet inside another, and it always costs overhead, which is why broken path MTU discovery produces the classic small-pages-work, large-pages-hang fault.
12. SSH gives three tunnels: `-L` opens a port on your side, `-R` opens a port on theirs, and `-D` gives a SOCKS proxy for anything.
13. Services such as ngrok and Cloudflare Tunnel expose a local port publicly by dialling outward first, which is why they work behind a home router with no port forwarding.
14. A CDN caches what is identical for everybody, uses a cache key built from scheme, host, path and query, and turns a 600 millisecond origin fetch into a 45 millisecond edge hit.
15. Anycast is one address announced from many sites by BGP; it suits DNS and UDP perfectly, works for TCP because routes are stable rather than pinned, and breaks a live connection when routing changes mid-flow.
16. DNS steering is the other mechanism entirely: different answers to different askers, limited by TTLs and by the fact that the server sees the resolver, such as the reader's `1.1.1.1`, and not the user.
17. Latency is bounded by the speed of light in fibre, about 200,000 kilometres per second, so India to United States East cannot go below about 128 milliseconds and measures 185 to 200 in practice.
18. Bandwidth-delay product is bandwidth times round-trip time, so a 100 megabit link at 200 milliseconds needs 2.5 megabytes in flight, and page load time is dominated by round trips rather than by capacity.
19. IPv6 addresses are 128 bits with two compression rules, use SLAAC and neighbour discovery instead of DHCP and ARP, and the reader's IPv6: (none) cost them nothing directly because `github.com` publishes no AAAA record, though it removed the second path Happy Eyeballs would have raced.
20. Broadcast reaches everyone on a link, multicast reaches a group, mDNS on `224.0.0.251` port 5353 makes `.local` discovery work with no server, and `curl -w` measures DNS, connect, TLS, first byte and transfer separately so that "the network is slow" becomes an actual number.



PART G

Building and Shipping Software

The web, APIs, version control, automation, the cloud and
keeping it all safe.

Web and App Development

35.0 What this chapter gives you

1. You will be able to draw the line between frontend and backend, say exactly what runs on the user's device and what runs on a server you control, and state the security rule that follows from it.
2. You will be able to say what HTML, CSS and JavaScript each are, why they are three separate things, and why only one of them is a programming language.
3. You will be able to explain the CSS cascade, specificity with the real calculation, the box model, flexbox and grid, and stop guessing at layout.
4. You will be able to describe the JavaScript event loop with the call stack, the task queue and the microtask queue, and predict the output order of a piece of asynchronous code by hand.
5. You will be able to trace an HTML file all the way to lit pixels: parse, DOM, CSSOM, render tree, layout, paint, composite, and say what reflow costs.
6. You will be able to say what a frontend framework actually solves, how a virtual DOM and reconciliation work, and when you do not need one at all.
7. You will be able to write a minimal web server in two languages, explain routing and middleware, and compare Node, Python, Java, Go, Ruby, PHP and .NET on honest grounds rather than fashion.
8. You will be able to design a small relational schema, write SQL with joins and grouping, explain how a B-tree index turns a scan into a lookup, read an EXPLAIN plan, and define ACID one letter at a time.
9. You will be able to store a password correctly, choose between sessions and tokens, describe the OAuth 2.0 authorization code flow step by step, and separate authentication from authorization.
10. You will be able to replace the "180-day roadmap" with an order of learning that matches how long these things really take.

35.1 Frontend, backend and the line between them

■ 35.1.1 PLAIN – in simple words

1. A web application lives in two places at once.
2. Part of it runs on the user's own device, inside the browser. That part is called the **frontend**.
3. Part of it runs on a computer you own or rent, far away. That part is called the **backend**.
4. The frontend is what the user sees and touches: text, pictures, buttons, boxes to type in.
5. The backend is what the user never sees: the stored data, the rules about who may do what, the money, the emails, the search.
6. The two talk to each other over the network, using the request and reply pattern from the networking chapters.

7. The device asks, the server answers. The device asks again, the server answers again. That is the whole shape of it.
8. This is called **client-server**. The client is the asking side. The server is the answering side.
9. Here is the single most important rule in this whole chapter. Everything on the user's device can be changed by the user.
10. The user can edit the page, change the numbers you sent, delete your checks, and send you anything at all.
11. So the frontend can be polite, helpful and fast. It can never be trusted.
12. Every rule that matters must be checked again on the server, where the user cannot reach it.

■ 35.1.2 PLAIN – a picture in your head

1. Think of a bank branch.
2. The counter, the forms, the pen on a chain and the queue barrier are the frontend. They are in the public part of the building.
3. The vault, the ledgers and the staff-only room are the backend.
4. The form asks you to write your account number in the right shape. That is helpful. It stops honest people making honest mistakes.
5. But nobody thinks the form is security. A person can write anything on a form. They can bring their own form. They can shout their request instead.
6. So the clerk checks the request again, against the real ledger, behind the glass, before any money moves.
7. That second check is the backend check. The form was the frontend check.
8. Both are useful. Only one of them is protection.

Where this comparison breaks:

9. In a bank you can see the customer. On the web you cannot. The “customer” may be a program sending ten thousand requests a second from another country.
10. In a bank the forms live in the branch. On the web, you hand the customer a complete copy of your entire form-processing machine and let them run it at home. They can take it apart first.
11. And a bank branch serves one person at a time at each counter. A server handles thousands of half-finished conversations at once.

■ 35.1.3 PLAIN – a worked example

1. A shop page shows a jacket at 4,000 rupees. The page has a discount box.
2. The frontend code checks the discount code, works out 10 percent off, and shows 3,600.
3. The user then presses Buy. The browser sends a message to the server.
4. A careless developer sends the price in that message:

```
| { "item": "jacket-01", "price": 3600, "code": "SAVE10" }
```
5. The user opens the browser's developer tools, changes one number, and sends this instead:

```
| { "item": "jacket-01", "price": 1, "code": "SAVE10" }
```
6. If the server saves that order, the shop has just sold a jacket for one rupee. Nothing was hacked. The user simply typed a different number.
7. The correct message contains no price at all:

```
| { "item": "jacket-01", "code": "SAVE10", "qty": 1 }
```


8. The server looks up the real price of jacket - 01 in its own database, checks whether SAVE10 is real, unexpired and allowed for this user, and computes the total itself.
9. The frontend showed 3,600 so the user was not surprised. The server decided 3,600 because it is the only side that can decide.
10. This exact mistake has cost real companies real money many times.

■ 35.1.4 PLAIN – what is really happening inside

1. The browser is a program on the user's machine. Chapter 18 called such a program a process: code plus memory, run by the operating system.
2. When you visit a site, the browser makes a network connection to a server, as in Chapters 28 and 32, and asks for a document.
3. The server sends back text. Usually HTML, which describes the content.
4. The browser reads that text and builds an internal model of the page in its own memory. Then it draws that model on the screen.
5. The HTML normally refers to more files: stylesheets, scripts, images, fonts. The browser fetches each of those with more requests.
6. Once a script is running, it can make further requests on its own, without loading a new page. That is how modern pages update in place.
7. Every one of those requests is a separate arrival at the server, and the server has no memory of the previous one unless you build that memory.
8. That last point is what "HTTP is stateless" means, and it is why sessions and tokens exist at all, which is section 35.12.
9. The split exists for three reasons. Speed, because a round trip to a server costs tens or hundreds of milliseconds and a local reaction costs none.
10. Control, because your data and your rules must live where users cannot edit them.
11. Scale, because one server can serve many clients only if the clients do their own drawing.

■ 35.1.5 TECHNICAL – the engineer's version

1. The client-server model is an architectural style, not a protocol. HTTP is one instance of it. It was covered from the wire upward in Chapters 23 to 33.
2. HTTP/1.1 is defined by RFC 9110 to RFC 9112 (June 2022), which replaced the older RFC 7230 series. HTTP/2 is RFC 9113. HTTP/3 over QUIC is RFC 9114.
3. HTTP is stateless by design: each request carries everything needed to process it. State is reconstructed from cookies, headers or tokens.
4. The trust boundary sits at the network edge of your server. Everything on the far side of it is **untrusted input**, including your own JavaScript.
5. The formal rule is the end-to-end argument from Saltzer, Reed and Clark (1981, 1984): a check is only meaningful at the point that owns the outcome.

Concern	Frontend can	Server must
Field format	show hints early	validate again
Price, totals	display only	compute, authoritative
Permissions	hide buttons	enforce on every call
Rate limits	debounce clicks	count and reject
Secrets	never hold any	hold all of them

6. Client-side validation is a usability feature with a measurable benefit: it removes a network round trip, typically 30 to 300 ms depending on distance.

7. Anything shipped to the browser is public. API keys in a bundle, “hidden” admin routes and commented-out code are all readable. Minification is not obfuscation and obfuscation is not security.
8. Tools that make this concrete: browser DevTools Network and Sources panels, `curl -v`, Burp Suite and mitmproxy for intercepting and editing requests.
9. Server-side rendering and client-side rendering shift where HTML is produced, but they do not move the trust boundary at all. That is fixed.

■ 35.1.6 WORDS – remember these

1. Frontend — the part running on the user’s device — client-side code executing in the browser’s JavaScript engine and rendering engine.
2. Backend — the part running on your machine — server-side processes you control, holding data, secrets and authorization logic.
3. Client-server — one side asks, the other answers — an architectural style where a requesting process contacts a listening service process.
4. Stateless — the server forgets between requests — no server-held context is implied by the protocol; state must be carried or looked up.
5. Trust boundary — the line where you stop believing input — the point where data crosses from an environment you do not control into one you do.

35.2 HTML: the structure of a page

■ 35.2.1 PLAIN – in simple words

1. HTML stands for HyperText Markup Language.
2. A **markup language** is a way of adding labels to text to say what each part of the text is.
3. It does not say what the text should look like. It says what the text means.
4. You mark up a piece of text by wrapping it in a tag. A tag is a word inside angle brackets.
5. `<p>Hello</p>` says: the word Hello is a paragraph.
6. Most tags come in pairs. An opening tag, then the content, then a closing tag with a slash.
7. The opening tag, the content and the closing tag together are called an **element**.
8. Tags can carry extra information called **attributes**, written as `name="value"` inside the opening tag.
9. Elements go inside other elements. That is **nesting**, and it makes the page a tree, like folders inside folders.
10. HTML is not a programming language. It cannot add two numbers, make a decision, or repeat something. It only describes structure.
11. That is not an insult. Describing structure well is the whole job, and most accessibility and search problems come from doing it badly.

■ 35.2.2 PLAIN – a picture in your head

1. Think of a printed book being prepared before typesetting.
2. An editor goes through the manuscript with a pencil and writes in the margin: this line is a chapter title, this block is a quotation, this is a footnote, this is a list.
3. The editor does not choose the font. The editor does not choose the size. Those come later, from the designer.
4. The editor’s marks are the markup. They record what each piece of text *is*.
5. A different designer can take the same marked-up manuscript and produce a paperback, a large-print edition, or an audiobook script.

6. Because the marks say “chapter title”, the audiobook narrator knows to pause and change tone. The word “chapter title” carries meaning that “big bold text” does not.

Where this comparison breaks:

7. The editor’s marks are read by a human who can use judgement. HTML is read by a machine that will do exactly and only what the tags say.
8. A book is finished once. An HTML page is re-laid-out every time the window size changes, on screens the author never saw.
9. And HTML is more forgiving than any editor would be. Browsers repair broken markup silently, which hides mistakes rather than reporting them.

■ 35.2.3 PLAIN - a worked example

1. Here is a complete, valid page. Everything in it earns its place.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport"
    content="width=device-width, initial-scale=1">
  <title>Bookshop - Contact</title>
  <link rel="stylesheet" href="/style.css">
</head>
<body>
  <header>
    <h1>Ambleside Bookshop</h1>
    <nav aria-label="Main">
      <a href="/">Home</a>
      <a href="/contact" aria-current="page">Contact</a>
    </nav>
  </header>

  <main>
    <h2>Send us a message</h2>
    <form action="/contact" method="post">
      <label for="name">Your name</label>
      <input id="name" name="name" type="text" required>

      <label for="email">Email</label>
      <input id="email" name="email" type="email" required>

      <label for="msg">Message</label>
      <textarea id="msg" name="msg" rows="5"></textarea>

      <button type="submit">Send</button>
    </form>
  </main>

  <footer>
    <p>Open 9am to 6pm, closed Sunday.</p>
  </footer>
</body>
</html>
```

2. Line by line, the parts that people get wrong.

3. `<!DOCTYPE html>` is not a tag. It is a switch. Without it, browsers fall into “quirks mode” and copy 1990s bugs on purpose.
4. `lang="en"` tells screen readers which language to pronounce, and tells search engines who this page is for.
5. `<meta charset="utf-8">` says the bytes are UTF-8, from Chapter 7. Leave it out and names with accents may turn into rubbish.
6. The viewport meta tag tells a phone not to pretend it is a 980-pixel desktop. Without it, your responsive CSS does nothing on mobile.
7. `<label for="name">` is joined to `<input id="name">` by the matching value. Now clicking the label focuses the box, and a screen reader announces the right words.
8. `name="name"` is what the server receives. `id` is for the page. They are different jobs that beginners merge.
9. `type="email"` makes phone keyboards show the at sign, and lets the browser check the shape for free.

■ 35.2.4 PLAIN – what is really happening inside

1. The browser receives the file as a stream of bytes, not as a page.
2. It decodes those bytes into characters using the declared encoding, then splits the characters into tokens: start tag, end tag, text, comment.
3. It builds a tree from those tokens. Each element becomes a node with a parent and a list of children. That tree is the DOM, which is section 35.5.
4. **Semantic elements** are tags whose names describe the role of the content: `header`, `nav`, `main`, `article`, `section`, `aside`, `footer`, `h1` to `h6`, `figure`, `time`.
5. They matter for two audiences you cannot see.
6. Screen readers build a list of headings and landmarks from them, so a blind user can jump straight to the main content instead of hearing the menu on every page.
7. Search engines use them to work out what the page is about and which part is the real content.
8. A `<div>` says nothing. It is a plain box. A page built only from divs looks identical and is much harder to use without sight.
9. Forms are the one place plain HTML can send data. A form with `method="post"` packages the named fields and sends them to the server on its own, with no JavaScript at all.
10. The browser gives you validation, keyboard handling, autofill and mobile keyboards for free, but only if you use real form elements.

■ 35.2.5 TECHNICAL – the engineer’s version

1. HTML was created by Tim Berners-Lee at CERN; the first public description circulated in 1991. HTML 2.0 was standardized as RFC 1866 in November 1995.
2. HTML 4.01 became a W3C Recommendation in December 1999. XHTML 1.0 followed in 2000 and pushed strict XML syntax, which the web rejected.
3. WHATWG formed in 2004 in response. HTML5 became a W3C Recommendation on 28 October 2014. Since a 2019 agreement, the WHATWG HTML Living Standard is the single normative specification. There are no numbered versions now.
4. The parsing algorithm is fully specified, including error recovery. Two conforming browsers must build the same DOM from the same broken input. This is a standard, not a convention, and it is unusual in being so.
5. Elements are void (`img`, `br`, `input`, `meta`), raw text (`script`, `style`), or normal. Void elements have no closing tag and no children.

6. Attribute values should be quoted. Unquoted values are legal in HTML5 but break on spaces. Quoting is a convention with a good reason.
7. ARIA (Accessible Rich Internet Applications, WAI-ARIA 1.2, W3C Recommendation June 2023) adds roles and states for widgets HTML lacks.
8. The first rule of ARIA, stated in the specification's own authoring practices, is not to use ARIA when a native element exists. A real `<button>` beats `<div role="button" tabindex="0">` on every measure.

Input type	Added in	Mobile keyboard	Free validation
text	HTML 2.0	standard	none
email	HTML5	at sign	shape check
number	HTML5	digits	min, max, step
date	HTML5	date picker	range
tel	HTML5	phone pad	none

9. Tools: the W3C Nu HTML Checker (validator.w3.org/nu), axe DevTools, and the accessibility tree view in Chrome and Firefox DevTools.

■ 35.2.6 WORDS - remember these

1. Markup — labels added to text — a syntax that annotates content with structural and semantic roles rather than presentation.
2. Element — a tag plus its content — a node in the document tree with a tag name, attribute set and child node list.
3. Attribute — extra detail on a tag — a name-value pair on an element, exposed in the DOM as content attributes and IDL properties.
4. Semantic HTML — tags that say what content is — markup chosen so the accessibility tree and machine readers infer correct roles.
5. Void element — a tag with no closing half — an element defined as having no end tag and no permitted children, such as `img` or `input`.
6. Doctype — the switch that turns on modern rules — a legacy string that selects no-quirks mode in the HTML parser.

35.3 CSS: presentation, and the cascade that confuses everyone

■ 35.3.1 PLAIN - in simple words

1. CSS stands for Cascading Style Sheets.
2. HTML says what things are. CSS says what they look like.
3. Keeping those two apart is the idea called **separation of content and presentation**.
4. The benefit is real. Change one stylesheet and every page changes. Give the same HTML a different stylesheet and you get a different design.
5. A CSS rule has two halves. A **selector** picking which elements to affect, and a block of **declarations** saying what to do to them.
6. A declaration is a property and a value: `color: black;`.
7. Many rules can hit the same element. When two rules disagree, the browser has to pick one. The picking procedure is called the **cascade**.
8. The cascade is where almost everyone gets stuck, so we will do it properly and give you the real calculation.

9. Every element is drawn as a rectangle. Understanding that rectangle, the box model, removes most layout confusion.
10. Two layout systems, flexbox and grid, do nearly all modern layout. Learn those two well and you can stop guessing.

■ 35.3.2 PLAIN – a picture in your head

1. Imagine a school with a stack of rulebooks about uniform.
2. The national rule says shoes must be black. The school rule says shoes must be brown. The class teacher's note says shoes must be blue.
3. A pupil arrives. Which colour wins?
4. You need an agreed order for settling arguments, or the pupil cannot dress.
5. The school's order is: a note about *this named pupil* beats a note about *this class*, which beats a note about *all pupils*.
6. If two notes are equally specific, the one written most recently wins.
7. And a rule marked "no exceptions, headmaster" beats everything, which is why the headmaster should use it almost never.
8. That is the cascade. Origin, then importance, then specificity, then order.

Where this comparison breaks:

9. Rules about pupils apply to whole pupils. CSS settles each property separately. Colour can come from one rule and size from another, in the same element, at the same time.
10. And some CSS properties are inherited by children automatically, like text colour, while others are not, like border. No school rule works that way.

■ 35.3.3 PLAIN – a worked example

1. Here is a real box, measured.

```
.card {
  width: 300px;
  padding: 20px;
  border: 2px solid black;
  margin: 16px;
  box-sizing: content-box;
}
```

2. With content-box, which is the original CSS behaviour, width means the content area only.
3. Total horizontal space the box occupies on screen:

Part	Left	Right	Total
content	-	-	300
padding	20	20	40
border	2	2	4
margin	16	16	32

4. Drawn width is $300 + 40 + 4 = 344$ px. Space consumed in the flow, including margin, is 376 px.
5. That surprise is why almost every project now writes:

```
| *, *::before, *::after { box-sizing: border-box; }
```

6. With `border-box, width: 300px` means the border edge is 300 px, so the content area shrinks to $300 - 40 - 4 = 256$ px. What you type is what you measure.
7. Now specificity, calculated properly. Specificity is a triple: (A, B, C).

Selector	A ids	B classes	C elements
p	0	0	1
.note	0	1	0
p.note	0	1	1
#main p.note	1	1	1
:is(#a, p)	1	0	0

8. Compare left to right. (1,0,0) beats (0,9,9). A single id beats nine classes. The columns are not digits of one number, they are separate ranks.
9. Attribute selectors and pseudo-classes count in column B. Pseudo-elements such as `::before` count in column C. The universal selector `*` counts zero.
10. `:where(...)` always contributes zero, which makes it the correct tool for library defaults you want authors to override easily.

■ 35.3.4 PLAIN - what is really happening inside

1. The browser collects every declaration that could apply to one element and one property, then sorts them by these steps in order.
2. Step one, origin and importance. Order from weakest to strongest: browser defaults, then user settings, then your stylesheet, then your `!important` rules, then user `!important`, then browser `!important`.
3. Note the flip. Among `!important` rules the order reverses, so a user's accessibility override can beat an author's. That is deliberate.
4. Step two, since 2023, cascade layers declared with `@layer`. Later layers beat earlier ones, and this is decided before specificity.
5. Step three, specificity, using the (A, B, C) triple above.
6. Step four, source order. If everything else ties, the declaration that appears later in the stylesheet wins.
7. Inline `style="..."` is not a selector. It sits above all normal author rules and is only beaten by `!important`.
8. If no rule sets a property, the element may still get a value by **inheritance** from its parent. Text properties inherit. Box properties do not.
9. If nothing inherits either, the property takes its **initial value** from the specification.
10. The honest version: people say "the most specific rule wins". That is only step three of six, and it is not even the first tiebreaker. The full order is origin, importance, layer, specificity, then source order.

■ 35.3.5 TECHNICAL - the engineer's version

1. CSS was proposed by Hakon Wium Lie on 10 October 1994 at CERN, with Bert Bos. CSS Level 1 became a W3C Recommendation on 17 December 1996. CSS 2.1 became a Recommendation in June 2011.
2. There is no "CSS3" specification. Since Level 2 the language is split into independently versioned modules: Selectors Level 4, Grid Level 2, and so on. Saying "CSS3" is a marketing habit, not a standard.

3. Display types: block, inline, inline-block, flex, grid, none, and the two-value syntax display: inline flex. none removes the element from layout entirely; visibility: hidden keeps its space.
4. Flexbox is one-dimensional: items are distributed along a main axis. It reached W3C Candidate Recommendation status in 2012 and shipped broadly from 2013.

```
.toolbar {
  display: flex;
  justify-content: space-between;
  align-items: center;
  gap: 12px;
  flex-wrap: wrap;
}
.toolbar .spacer { flex: 1 1 auto; }
```

5. Grid is two-dimensional: rows and columns declared together. It shipped in Chrome 57 and Firefox 52 in March 2017, which is the date modern CSS layout really begins.

```
.page {
  display: grid;
  grid-template-columns: 240px minmax(0, 1fr);
  grid-template-rows: auto 1fr auto;
  gap: 16px;
  min-height: 100vh;
}
```

6. The fr unit means one share of the leftover space. minmax(0, 1fr) is the standard fix for grid children that refuse to shrink below their content.
7. Positioning: static (default), relative (offset from its own place, keeps its space), absolute (removed from flow, positioned against the nearest positioned ancestor), fixed (against the viewport), sticky (relative until a scroll threshold, then fixed).

Unit	Relative to	Typical use
px	device-independent pixel	borders, hairlines
em	own font-size	spacing inside a component
rem	root font-size (16 px)	type scale, layout
%	parent's same dimension	fluid widths
vh / vw	1% of viewport	full-screen sections
ch	width of "0" glyph	text measure limits

8. One CSS px is defined as 1/96 inch at a nominal viewing distance, not one hardware pixel. On a 3x phone screen, one CSS px covers nine device pixels.
9. Media queries adapt to the environment:

```
@media (min-width: 48rem) { .page { --cols: 2; } }
@media (prefers-reduced-motion: reduce) {
  * { animation-duration: 0.01ms !important; }
}
```

10. Mobile-first means writing the small-screen rules first and using min-width queries to add complexity. It is a convention, and a good one.
11. Custom properties, usually called CSS variables, are real cascading values, live at runtime, and are readable by JavaScript. They shipped in 2016.

```
:root { --gap: 12px; --ink: #111; }
.card { padding: var(--gap); color: var(--ink); }
```

12. Container queries let a component respond to its own container rather than the viewport. They reached cross-browser support in February 2023.

```
.sidebar { container-type: inline-size; }
@container (min-width: 30rem) {
  .card { display: grid; grid-template-columns: 1fr 2fr; }
}
```

13. Native CSS nesting reached cross-browser support during 2023 and 2024 and removes one of the main historical reasons to use Sass.
14. `:has()`, the long-requested parent selector, shipped across browsers by December 2023. `.card:has(img)` styles a card because of what it contains.
15. Tools: DevTools Styles pane shows the winning declaration and strikes out losers; the Layout pane overlays grid and flex lines; `@supports` queries let you ship progressive enhancement safely.

■ 35.3.6 WORDS – remember these

1. Cascade — the argument-settling procedure — the ordered algorithm of origin, importance, layer, specificity and source order.
2. Specificity — how targeted a selector is — the (id, class, type) triple compared left to right, never summed.
3. Box model — every element is a rectangle — content, padding, border and margin boxes, with box-sizing choosing what width measures.
4. Flexbox — line things up along one direction — CSS Flexible Box Layout, one-dimensional distribution along a main and cross axis.
5. Grid — a real table-free grid — CSS Grid Layout, two-dimensional placement with explicit and implicit tracks.
6. rem — a size tied to the page's base text — the computed font-size of the root element, 16 px unless changed.
7. Container query — react to my own box, not the window — `@container` rules evaluated against a queried ancestor's size.

35.4 JavaScript: the language and its event loop

■ 35.4.1 PLAIN – in simple words

1. JavaScript is the programming language that runs inside web browsers.
2. Unlike HTML and CSS, it really is a programming language. It can decide, repeat, calculate, store and react.
3. It was written in 1995 at a company called Netscape, by one person, Brendan Eich, in about ten days.
4. Ten days is not enough time to design a language carefully. Many of its odd corners come straight from that deadline.
5. It was never removed, because the web could not break old pages. So the world's most used language is one that was rushed and then frozen.
6. Since 1997 the language has had a written standard, and it now gets a new edition every June, so it improves without breaking what came before.
7. The single hardest idea in JavaScript is **asynchrony**: starting something slow, not waiting for it, and dealing with the answer when it arrives.

8. Asynchrony exists because JavaScript in a browser has only one worker. If that worker waits, the whole page freezes: no scrolling, no clicking.
9. So instead of waiting, it writes a note saying what to do when the answer comes, and gets on with other work.
10. The machine that manages those notes is called the **event loop**, and it is the thing you must understand to write correct JavaScript.

■ 35.4.2 PLAIN – a picture in your head

1. Picture a single cook in a small kitchen with a strict rule: finish the task in your hands before starting the next one.
2. An order arrives: boil rice, twenty minutes. The cook does not stand and watch the pot. That would block every other order.
3. The cook puts the rice on, sets a timer, writes “when the timer rings, drain the rice” on a slip, and puts the slip in a tray.
4. Then the cook takes the next order and starts chopping.
5. When the cook finishes a task and has nothing in hand, they look in the tray, take the oldest slip, and do it.
6. That is the event loop. One cook, no waiting, a tray of pending slips.
7. There is a second, smaller tray, right next to the cook, checked before the big one and emptied completely each time.
8. Anything urgent, like “the sauce reduced, add salt now”, goes in the small tray and is done before the next big order.
9. The small tray is the microtask queue. The big tray is the task queue.

Where this comparison breaks:

10. A real cook can glance at the pot while chopping. JavaScript truly cannot. If a function loops for three seconds, nothing else happens for three seconds, including the timer that was due one second ago.
11. And a timer set for 100 ms does not mean the slip is done at 100 ms. It means the slip is put in the tray at 100 ms. If the cook is busy, it waits.

■ 35.4.3 PLAIN – a worked example

1. Predict the output of this. Most people get it wrong the first time.

```
console.log("1");

setTimeout(() => console.log("2"), 0);

Promise.resolve().then(() => console.log("3"));

queueMicrotask(() => console.log("4"));

console.log("5");
```

2. The answer is 1, 5, 3, 4, 2.
3. Here is why, step by step.
4. The whole script is itself one task. It runs to the end before anything else is considered. So “1” prints, then “5” prints.
5. Along the way, `setTimeout` handed a callback to the timer system with a delay of zero. When zero milliseconds pass, that callback is placed in the **task queue**.

6. `Promise.resolve().then(...)` placed its callback in the **microtask queue** immediately, because the promise was already settled.
7. `queueMicrotask` placed its callback in the microtask queue too, after the promise one.
8. The script ends. The call stack is now empty.
9. Before touching the task queue, the engine drains the microtask queue completely. So “3” prints, then “4”.
10. Only now does it take one item from the task queue. “2” prints.

Moment	Stack	Microtasks	Tasks
script running	script	3, 4	(timer pending)
script ends	empty	3, 4	2
drain micro	3 then 4	empty	2
next task	2	empty	empty

11. The rule to memorize: microtasks run to exhaustion after every task, before the next task, and before the browser gets a chance to paint.
12. That gives you a real bug. A microtask that queues another microtask forever starves the page. The browser never repaints. Nothing on screen moves.

■ 35.4.4 PLAIN - what is really happening inside

1. The **call stack** is a pile of function calls. Calling a function pushes a frame on. Returning pops it off. When the pile is empty, the engine is idle.
2. Slow things are not done by JavaScript at all. Network requests, timers, disk access and user input are handled by the browser’s own code, in other threads, written in C++ or Rust.
3. When one of those finishes, the browser does not interrupt your code. It places your callback in a queue.
4. The event loop is a simple repeating check: is the stack empty? If yes, drain all microtasks, then take one task, run it, then possibly render.
5. Promises, `async/await`, `queueMicrotask` and `MutationObserver` use the microtask queue. Timers, network events, clicks and messages use task queues.
6. `await` is not magic. It splits your function in two at that point. The rest of the function becomes a microtask attached to the awaited promise.
7. Rendering happens between tasks, not during one. That is why a long loop freezes the screen even if you change styles inside it.
8. Three ways to write the same wait, in historical order.

```
// 1. Callbacks (1995 onward)
getUser(id, (err, user) => {
  if (err) return done(err);
  getOrders(user, (err, orders) => {
    if (err) return done(err);
    done(null, orders);
  });
});

// 2. Promises (standardized in ES2015)
getUser(id)
  .then(user => getOrders(user))
  .then(orders => done(null, orders))
  .catch(done);
```

```
// 3. async / await (ES2017)
try {
  const user = await getUser(id);
  const orders = await getOrders(user);
  return orders;
} catch (err) {
  handle(err);
}
```

9. All three do the same thing. The third is easiest to read and easiest to get right, because errors travel through ordinary try and catch.
10. The trap in version three: two sequential await calls that do not depend on each other waste time. Use Promise.all to run them together.

```
const [user, rates] = await Promise.all([
  getUser(id),
  getRates(),
]);
```

■ 35.4.5 TECHNICAL - the engineer's version

1. Brendan Eich wrote the first prototype in May 1995 at Netscape Communications. It was called Mocha, renamed LiveScript in September 1995, then JavaScript in December 1995 as part of a marketing deal with Sun.
2. The name is the source of endless confusion. JavaScript and Java are unrelated languages.
3. Netscape submitted it to Ecma International; ECMA-262 1st edition was published in June 1997. The standard language is called **ECMAScript**; JavaScript is a trademarked implementation name.
4. ES5 (December 2009) added strict mode and JSON. ES2015, also called ES6, was the large modernization: let, const, arrow functions, classes, modules, promises, template literals, destructuring.
5. Since 2015 there is one edition each June. ECMAScript 2025 was the 16th edition (June 2025) and ECMAScript 2026 the 17th (June 2026).
6. Types: number (IEEE 754 double, so integers are exact only to $2^{53} - 1$), bigint, string (UTF-16 code units), boolean, undefined, symbol, null, and object.
7. Coercion is the automatic conversion between types, and it is the source of the famous oddities.

Expression	Result	Reason
0.1 + 0.2	0.30000000000000004	binary floats
[] + {}	"[object Object]"	both to string
"5" - 2	3	minus forces number
"5" + 2	"52"	plus prefers string
NaN === NaN	false	IEEE 754 rule
typeof null	"object"	1995 bug, frozen

8. `0.1 + 0.2` is not a JavaScript flaw. It is IEEE 754 binary floating point, from Chapter 7, and it happens in Python, Java and C too.
9. Use `===` not `==`. `==` applies the coercion table and produces surprises such as `"" == 0` being true.
10. **Closures**: a function keeps a live reference to the variables of the scope where it was defined, even after that scope returns.

```
function counter() {
  let n = 0;
  return () => ++n;
}
```

```

    }
    const next = counter();
    next(); // 1
    next(); // 2

```

11. JavaScript uses **prototypal inheritance**. Every object has a hidden link to another object; property lookup walks that chain. `class` syntax, added in ES2015, is sugar over prototypes, not a separate system.
12. Modules: ES modules (`import`, `export`) are the standard, defined in ES2015 and shipped in browsers from 2017. CommonJS (`require`, `module.exports`) is Node's older system and is still everywhere.
13. Engines: V8 (Chrome, Node, Edge, Deno), SpiderMonkey (Firefox), JavaScriptCore (Safari, Bun). All are JIT compilers with multiple tiers, described in Chapter 16.
14. Error handling rules that matter: never swallow an error silently; attach a `.catch` or `try/catch` to every promise chain; an unhandled rejection terminates a Node process by default since Node 15 (2020).
15. Tools: `node --prof`, Chrome DevTools Performance panel for long tasks, `performance.now()` for sub-millisecond timing.

■ 35.4.6 WORDS – remember these

1. Event loop — the one-worker queue system — the algorithm that drains microtasks and dispatches one task per turn on a single thread.
2. Call stack — the pile of calls in progress — the LIFO structure of execution contexts; overflow raises `RangeError`.
3. Microtask — an urgent note done before anything else — a job queued to the microtask queue, drained fully after each task.
4. Promise — a receipt for a future value — an object in pending, fulfilled or rejected state, with then scheduling microtasks.
5. Closure — a function that remembers where it was born — a function object plus a reference to its defining lexical environment.
6. Prototype — the object I fall back to — the internal `[[Prototype]]` link traversed during property lookup.
7. ECMAScript — the written standard — ECMA-262, published annually each June since 2015; JavaScript is an implementation of it.

35.5 The DOM and how a browser turns HTML into pixels

■ 35.5.1 PLAIN – in simple words

1. HTML arrives as text. Text cannot be clicked, moved or measured.
2. So the browser converts the text into a live tree of objects held in memory. That tree is the **DOM**, the Document Object Model.
3. Every tag becomes a node. Every node knows its parent and its children.
4. JavaScript does not edit your HTML file. It edits this tree. Change the tree and the screen follows.
5. Your CSS is turned into a second structure with the same idea, called the CSSOM.
6. The browser combines the two into a **render tree**: only the things that will actually be drawn, each with its final style.
7. Then it works out where every box goes, which is **layout**. Then it fills in the colours, which is **paint**. Then it stacks the layers, which is **composite**.

8. If you change something that affects size or position, the browser must redo layout. That is called **reflow** and it is the expensive one.
9. If you change only a colour, it can skip layout and just repaint. Cheaper.
10. If you change only a transform or opacity, it can often skip both and just re-stack existing layers. Cheapest of all.
11. Knowing which of the three you triggered is most of frontend performance.

■ 35.5.2 PLAIN – a picture in your head

1. Think of a newspaper being made in a print room, before computers.
2. The reporters' typed pages are the HTML. Just text with notes on it.
3. The editor's style guide, saying headlines are 36 point and captions are italic, is the CSS.
4. The layout artist takes both and builds a paste-up board: which story goes in which column, how many lines it runs to, where the picture sits. That is layout.
5. Then the ink goes on: black type, grey photographs. That is paint.
6. Then the transparent overlay sheets are stacked in order. That is composite.
7. Now the editor says one caption should be bold. The word gets wider. The line rewraps. The column gets longer. The story below moves down. The whole board must be redone.
8. That is a reflow, and you can see why it costs.
9. But if the editor only says "print this headline in a darker grey", the shapes do not move. Only the ink changes. That is a repaint.

Where this comparison breaks:

10. A newspaper is laid out once. A browser may redo layout sixty times a second while you drag a window.
11. And in a print room the artist works on the whole board. Browsers do partial invalidation: they mark a subtree as dirty and try to redo only that. The cost still spreads upward more often than you would like.

■ 35.5.3 PLAIN – a worked example

1. Here is the classic layout-thrashing bug, in ten lines.

```
const items = document.querySelectorAll(".row");

// BAD: read, write, read, write, ...
for (const el of items) {
  el.style.height = el.offsetHeight + 10 + "px";
}
```

2. `offsetHeight` is a **read** of a geometric value. `style.height` is a **write** that invalidates geometry.
3. The browser batches writes and flushes them lazily. But a read forces it to flush immediately, so the answer is correct. That forced flush is a reflow.
4. With 200 rows, that loop causes 200 forced synchronous layouts. On a mid phone that can be 200 ms of frozen page.
5. The fix is to separate all reads from all writes.

```
const items = document.querySelectorAll(".row");

// GOOD: read everything, then write everything
const heights = [...items].map(el => el.offsetHeight);
items.forEach((el, i) => {
```



```
el.style.height = heights[i] + 10 + "px";
});
```

- Now there is one layout pass, not 200. Same output, roughly one hundredth of the cost.
- Properties that force a synchronous layout when read include `offsetTop`, `offsetHeight`, `scrollTop`, `clientWidth`, `getBoundingClientRect()` and `getComputedStyle()`.
- Now events. Suppose a list has 500 rows and each needs a click handler.

```
// 500 listeners, 500 objects, slow to attach
rows.forEach(r => r.addEventListener("click", onClick));

// 1 listener, using bubbling. This is delegation.
list.addEventListener("click", (e) => {
  const row = e.target.closest(".row");
  if (row) onClick(row.dataset.id);
});
```

- This works because a click on a row also fires on its parents, in order, outward. That travelling is **bubbling**, and using it deliberately is **event delegation**.
- Delegation also handles rows added later, which the first version cannot.

■ 35.5.4 PLAIN - what is really happening inside

- The full sequence, in order, for a first page load.

```
bytes -> characters -> tokens -> nodes -> DOM tree
CSS bytes -----> CSSOM tree
DOM + CSSOM -> render tree -> layout -> paint -> composite
```

- Parsing is incremental. The browser starts building the DOM before the whole file has arrived, which is why a slow page still shows its header early.
- A plain `<script>` tag in the middle of the body stops parsing dead. The browser must fetch and run it before continuing, because the script might write into the document.
- `defer` says: fetch it now in parallel, run it after parsing finishes, in order. `async` says: fetch now, run the moment it arrives, order not guaranteed. Use `defer` for application code.
- CSS is **render-blocking**. The browser will not paint until it has the stylesheets, because painting with the wrong styles then correcting would flash. That flash has a name, FOUC.
- Layout computes a box for every rendered node: x, y, width, height, in a single tree walk, honouring the box model and the layout mode of each parent.
- Paint turns boxes into a list of drawing commands: fill this rectangle, draw this glyph run, clip here.
- Composite runs those command lists into layers, often on the GPU from Chapter 22, and stacks them. Scrolling and CSS transforms usually only touch this stage, which is why they are smooth.
- An event has three phases: capture, travelling down from the document to the target; target; then bubble, travelling back up. Most code only ever uses the bubble phase.
- `stopPropagation()` halts travel. `preventDefault()` cancels the browser's built-in reaction, such as following a link. They are different and beginners confuse them constantly.

■ 35.5.5 TECHNICAL - the engineer's version

- DOM Level 1 became a W3C Recommendation in October 1998. The DOM is now a WHATWG Living Standard, and it is a language-independent API, not a JavaScript feature.
- The **critical rendering path** is the minimum set of resources needed for first paint: the HTML, the render-blocking CSS, and any parser-blocking JavaScript.

3. Rendering engines in production: Blink (Chrome, Edge, Opera, forked from WebKit on 3 April 2013), WebKit (Safari), Gecko (Firefox).
4. Style resolution is per element per property; matching selectors right to left is the standard implementation trick that makes descendant selectors cheap enough.
5. Reflow cost is superlinear in tree depth and node count. Modern engines do partial layout, but a change to a flow-root ancestor still invalidates its whole subtree.

Change	Triggers	Relative cost
width, top, font-size	layout, paint, composite	highest
background-color, color	paint, composite	medium
transform, opacity	composite only	lowest
adding a class with all three	full pipeline	highest

6. Animate transform and opacity. Animating left and width forces layout on every frame and will not hold 60 fps on a mid-range phone.
7. A frame budget at 60 Hz is 16.7 ms; at 120 Hz it is 8.3 ms. Chapters 21 and 22 traced that budget from key press to photons.
8. `requestAnimationFrame` schedules a callback just before the next paint, which is the correct hook for animation. `setTimeout(fn, 16)` is not.
9. `content-visibility: auto` and `contain` let you tell the engine that a subtree cannot affect the outside, allowing it to skip work entirely.
10. The DOM API is genuinely slow relative to plain objects, mostly because each call crosses from the JavaScript engine into the C++ engine. Batch changes with `DocumentFragment` or by building a string once.
11. Tools: Performance panel with the Frames and Main tracks, the “Layout Shift Regions” and “Paint flashing” overlays in Rendering, and the `PerformanceObserver` API for long-animation-frame entries.

■ 35.5.6 WORDS – remember these

1. DOM — a live tree of the page in memory — the Document Object Model, a language-neutral object representation defined by WHATWG.
2. CSSOM — the same idea for styles — the parsed stylesheet object model used during style resolution.
3. Render tree — only the things that get drawn — the tree of styled boxes, excluding `display: none` and including generated content.
4. Reflow — redo the positions — layout invalidation and recomputation of box geometry; also called relayout.
5. Repaint — redo the colours — rasterization of the display list without a geometry pass.
6. Composite — stack the finished layers — GPU-assisted assembly of rasterized layers into the final frame.
7. Bubbling — the event travels outward — phase 3 of DOM event dispatch, from target up to the root.
8. Delegation — one listener for many children — attaching a handler to a common ancestor and using `event.target`.

35.6 The browser as a platform

■ 35.6.1 PLAIN – in simple words

1. A browser is not a document viewer any more. It is an operating system for web pages.
2. It gives your page a set of built-in services, the same way an operating system gives a program files, memory and the network.
3. It can store data on the user's disk in three different ways, for different sizes and purposes.
4. It can fetch more data from a server at any time, without loading a new page.
5. It can run your code on extra threads, so slow work does not freeze the screen.
6. It can keep a small program running in the background that answers requests even with no network, which is how a web app works offline.
7. It can hold a two-way conversation with a server, where the server speaks first, instead of only answering questions.
8. It can connect two users' browsers directly to each other, for calls and video.
9. It can draw arbitrary graphics and run code on the graphics chip.
10. And it does all this inside a **sandbox**: a locked room where the page can touch only what it has been given, and must ask permission for the rest.

■ 35.6.2 PLAIN – a picture in your head

1. Think of a serviced office you rent by the hour.
2. You get a desk, a locked drawer, a phone and a window. That is the platform: storage, network, screen.
3. You cannot walk into another tenant's office. The doors are locked and the locks are not yours to change. That is the same-origin rule.
4. You cannot open the fire door to the roof without asking the building manager first, and the manager asks you in front of the tenant. That is the permission prompt for camera, microphone and location.
5. There are two small back rooms you can hire someone to work in, out of sight of your desk. One does heavy calculation. One sits by the post slot and answers deliveries even when you have gone home.
6. Those are web workers and service workers.

Where this comparison breaks:

7. An office tenant can carry things down the corridor by hand. A page cannot reach another origin's storage at all, by any route, including through the user's own copy of the browser.
8. And a real building manager can be persuaded. The sandbox is enforced by the browser's own code, and cannot be talked around by the page.

■ 35.6.3 PLAIN – a worked example

1. A small task: fetch a list, cache it, and survive going offline.

```
// 1. Ask the server for JSON
const res = await fetch("/api/books", {
  headers: { "Accept": "application/json" },
});
if (!res.ok) throw new Error("HTTP " + res.status);
const books = await res.json();

// 2. Keep a copy the user can read while offline
localStorage.setItem("books", JSON.stringify(books));
```

2. Note `res.ok`. `fetch` does not reject on a 404 or a 500. It only rejects on a network failure. Forgetting this is one of the most common bugs.
3. Now the storage choices, with real limits.

Store	Holds	Typical limit	Sent to server
Cookie	small strings	4 KB each	yes, every request
localStorage	strings	about 5 MB	no
sessionStorage	strings, per tab	about 5 MB	no
IndexedDB	structured objects	large, quota-based	no
Cache Storage	whole responses	quota-based	no

4. `localStorage` is synchronous. A 2 MB read blocks the single thread, and the page freezes for that time. Do not use it for anything large.
5. `IndexedDB` is asynchronous and can hold hundreds of megabytes, subject to a quota that is usually a share of free disk space.
6. Cookies travel with every matching request automatically. That is why they suit session identifiers and suit nothing else.
7. A minimal service worker turns that `fetch` into an offline-capable one.

```
// sw.js
self.addEventListener("fetch", (event) => {
  event.respondWith(
    caches.match(event.request).then(hit =>
      hit || fetch(event.request)
    )
  );
});
```

8. Now the browser asks the service worker before it asks the network. If the worker has a copy, the page loads with the aeroplane mode switch on.

■ 35.6.4 PLAIN – what is really happening inside

1. The **same-origin policy** is the core rule. An origin is the triple of scheme, host and port. `https://a.com` and `https://a.com:8443` are different origins, and so are `http` and `https` versions of one host.
2. Code from one origin cannot read the DOM, cookies or storage of another. It can still cause requests to another origin, which is why `CSRF` exists.
3. `CORS` is the controlled exception. The *server* adds a response header saying which origins may read its answers. The browser enforces it. The rule lives with the server, not the page.
4. A **web worker** is a separate thread with its own memory and no access to the DOM. You talk to it by sending messages that are copied, not shared.
5. Use one for parsing a large file, image processing, or cryptography, so the main thread stays free to draw at 60 frames per second.
6. A **service worker** is a special worker that sits between the page and the network, like a programmable proxy on the user's own machine.
7. It survives page closes, wakes for push messages, and must be served over `HTTPS`, because a hostile one would be permanent.
8. **WebSockets** solve a real problem. Plain `HTTP` is ask-and-answer, so a server with news cannot speak first.

9. The old workaround was polling: ask every few seconds, usually to be told nothing happened. Wasteful in requests, battery and latency.
10. A WebSocket starts life as an ordinary HTTP request with an Upgrade header, then the same TCP connection becomes a two-way message pipe that stays open.
11. **WebRTC** goes further and connects two browsers directly, so audio and video do not pass through your server. Your server only helps them find each other.

■ 35.6.5 TECHNICAL - the engineer's version

1. Same-origin policy dates to Netscape Navigator 2.02 in 1995. CORS is defined in the WHATWG Fetch Standard. Preflight uses an OPTIONS request when the method or headers are non-simple.
2. `fetch()` shipped from 2015 and replaced XMLHttpRequest, which came from Microsoft Outlook Web Access around 1999 and reached IE 5 in 1999. The term AJAX was coined by Jesse James Garrett in February 2005.
3. WebSocket is RFC 6455, December 2011. Handshake is HTTP Upgrade: `websocket`; frames then carry a 2 to 14 byte header, versus roughly 500 to 800 bytes of HTTP headers per polled request.

Technique	Direction	Overhead	Best for
Polling	client asks	high, constant	rare updates
Long polling	client waits	medium	legacy fallback
SSE	server to client	low, text only	feeds, progress
WebSocket	both ways	lowest per message	chat, games
WebRTC data	peer to peer	lowest latency	calls, screen share

4. Server-Sent Events use the `text/event-stream` MIME type, reconnect automatically, and are part of the HTML Living Standard. They are one-way and much simpler to operate than WebSockets.
5. WebRTC was open-sourced by Google in May 2011 and became a W3C Recommendation on 26 January 2021. It needs STUN and usually TURN servers to traverse NAT, which Chapter 24 described. TURN relays cost real bandwidth.
6. Graphics: Canvas 2D is immediate-mode drawing. WebGL 1.0 shipped March 2011 and is based on OpenGL ES 2.0; WebGL 2.0 (2017) on OpenGL ES 3.0. WebGPU shipped in Chrome 113 in May 2023, in Safari 26 in 2025, and gives compute shaders and a modern binding model.
7. Storage quota is not fixed by specification. Chrome allows an origin up to about 60 percent of total disk in practice; Safari applies stricter eviction. This is an implementation detail and changes between versions.
8. Permissions requiring an explicit user prompt: camera, microphone, geolocation, notifications, clipboard read, MIDI, Bluetooth, USB, and persistent storage. Most also require a secure context (HTTPS or localhost).
9. Powerful features are gated three ways: secure context, transient user activation (a real click within a few seconds), and Permissions Policy headers set by the site.
10. Tools: DevTools Application panel shows every store, service worker state, and quota usage; `chrome://webrtc-internals` shows live peer connections.

■ 35.6.6 WORDS - remember these

1. Sandbox — a locked room for the page — an isolated execution environment with mediated access to host resources.
2. Origin — the identity of a site — the tuple of scheme, host and port used for all isolation decisions.
3. CORS — the server's note saying who may read this — Cross-Origin Resource Sharing headers defined by the Fetch Standard.

4. Service worker — a proxy of your own on the user's machine — an event-driven worker intercepting fetches, enabling offline and push.
5. Web worker — a second thread with no screen access — a dedicated worker with an isolated global scope and message-passing only.
6. WebSocket — a phone line instead of letters — RFC 6455 full-duplex framed messaging over a single upgraded TCP connection.
7. IndexedDB — a real database in the browser — a transactional object store with indexes and an asynchronous API.

35.7 Frontend frameworks, explained rather than listed

■ 35.7.1 PLAIN – in simple words

1. Here is the problem a framework exists to solve.
2. Your page shows information. That information changes. The page must change to match, everywhere it appears, every time.
3. Doing that by hand means writing instructions: find this element, change its text, add this row, remove that one, toggle this class.
4. That style is called **imperative**: you say the steps.
5. With ten pieces of information it is fine. With two hundred, in a page where any change can affect any other, it becomes impossible to keep straight.
6. A framework lets you write **declarative** code instead: you describe what the page should look like for the current data, and the framework works out the steps to get there.
7. The second big idea is the **component**: one self-contained piece of the page with its own markup, styling and behaviour, that you can use many times.
8. A page becomes a tree of components, the way a house is a set of rooms rather than one giant space.
9. That is all a framework is: a way to say “screen equals function of data”, plus a way to cut a screen into reusable pieces.

■ 35.7.2 PLAIN – a picture in your head

1. Imagine a departures board at a station, made of physical letter tiles.
2. The imperative way: a person with a ladder, told “change platform 4 from 12 to 9, and change the 14:05 to Delayed, and remove the 13:50 row”.
3. Every instruction must be right, in the right order, or the board lies.
4. The declarative way: you hand over a printed sheet showing what the board should say now. A machine compares it with what the board says and changes only the tiles that differ.
5. You never think about tiles. You think about the correct board.
6. The comparing step is called **reconciliation**, and the sheet you hand over is the idea behind a **virtual DOM**.

Where this comparison breaks:

7. The station board has fixed rows. A web page has a shifting tree, so the comparison needs hints about identity. That is what the key attribute is for, and getting keys wrong causes real bugs.
8. And comparison is not free. Some frameworks skip it entirely by working out at build time exactly which tiles a given change touches.

■ 35.7.3 PLAIN – a worked example

1. The same counter, three ways.

```
// Imperative: plain DOM
let n = 0;
btn.addEventListener("click", () => {
  n = n + 1;
  out.textContent = "Count: " + n;
});

// Declarative: React
function Counter() {
  const [n, setN] = useState(0);
  return (
    <button onClick={() => setN(n + 1)}>
      Count: {n}
    </button>
  );
}

<!-- Declarative: Svelte 5 -->
<script>
  let n = $state(0);
</script>
<button onclick={() => n++}>Count: {n}</button>
```

2. In the first, you own the update. In the other two, you own the description and the tool owns the update.
3. With one counter the first is shorter. With a form of forty fields that feed a summary, a chart and a save button, the first becomes a maze.
4. What React does when `setN` runs: it calls `Counter()` again, gets a new description, compares it with the old one, sees only the text differs, and changes that one text node. It does not rebuild the button.
5. What Svelte does: at build time the compiler already worked out that `n` appears in exactly one text position, so it emits code that assigns to that text node directly. There is no comparison at runtime.

■ 35.7.4 PLAIN – what is really happening inside

1. **State** is the data that can change. **Props** are the data a parent hands to a child. A component re-renders when its own state or its props change.
2. A **virtual DOM** is a plain tree of ordinary JavaScript objects describing the wanted output. Cheap to build, cheap to throw away.
3. Reconciliation walks the old tree and the new tree together. Same type at the same position means update in place. Different type means destroy and rebuild. Lists are matched by key.
4. Using an array index as a key is the classic bug: delete the first row and every key shifts, so the framework updates the wrong rows and typed input jumps to the wrong place.
5. **State management** is the question of where shared data lives. Local state inside a component is easy. Data that four distant components need is not.
6. The escalation ladder is: local state, then lift state to a common parent, then a context or provider, then a dedicated store, then a server-state library that caches responses.
7. **Hydration** happens when HTML is rendered on the server for speed, then the same components run in the browser to attach event handlers to the existing markup. The page looks ready before it is interactive.

8. The honest version: hydration is a workaround, not a triumph. It ships the data twice, once as HTML and once as JSON, and it is why some server-rendered pages feel dead for a second. Islands, partial hydration and React Server Components are all attempts to send less.

■ 35.7.5 TECHNICAL – the engineer’s version

1. React was created at Facebook by Jordan Walke and open-sourced in May 2013 at JSConf US. Facebook renamed itself Meta in October 2021. React 19 is current, with 19.2 published in 2026.
2. JSX is a syntax extension compiled to function calls; it is not HTML and not part of the language. `class` becomes `className` because `class` is reserved.
3. React uses one-way data flow: data goes down through props, changes go up through callbacks. Hooks arrived in React 16.8 (February 2019) and replaced class components as the normal style.
4. The core hooks are `useState`, `useEffect`, `useMemo`, `useCallback`, `useRef` and `useContext`. Rules: only at the top level, only in components or other hooks, because hook identity is positional.
5. `useEffect` is for synchronizing with something outside React, not for deriving data. Overuse of effects is the most common source of React bugs.

Framework	First release	Approach	Trade-off
React	May 2013, Meta	virtual DOM, hooks	huge ecosystem, more rerenders
Angular	Sept 2016 (v2)	full framework, DI	batteries included, steep start
Vue	Feb 2014, Evan You	reactive proxies	gentle curve, smaller job market
Svelte	Nov 2016, R. Harris	compiler, runes	least runtime code, smaller ecosystem
Solid	2018, R. Carniato	fine-grained signals	very fast, small community

6. AngularJS (October 2010, Misko Hevery at Google) and Angular 2 onward (14 September 2016) are different frameworks with a shared name. Angular uses TypeScript, RxJS and dependency injection, and now ships version 22.
7. Vue 3 (September 2020) uses ES Proxy-based reactivity and a composition API. Svelte 5 (19 October 2024) introduced runes, replacing its label-based reactivity. Solid uses signals with no component rerenders at all.
8. Signals, which Solid popularized, have now been adopted by Angular, Vue, Svelte and Preact, and a TC39 signals proposal exists. Experts disagree on whether React should adopt them; React’s team argues its compiler achieves the same result without changing the model.
9. Meta-frameworks add routing, server rendering and data loading: Next.js for React (version 16 in 2026), Nuxt for Vue, SvelteKit, Angular with its own SSR, Remix and Astro.
10. When you do not need a framework: content sites, marketing pages, blogs, documentation, forms with a handful of fields, and anything where the server can render the HTML and a hundred lines of plain JavaScript will do.
11. A framework costs 40 to 150 KB of JavaScript before your own code, plus a build step, plus upgrade work forever. Spend that only when interactivity genuinely earns it.

■ 35.7.6 WORDS – remember these

1. Declarative — describe the result — express the target state and delegate the transition to the runtime.

2. Component — one reusable piece of screen — an encapsulated unit of template, state and behaviour with a props interface.
3. Virtual DOM — a cheap paper copy of the page — an in-memory tree diffed against the previous render to compute minimal DOM mutations.
4. Reconciliation — comparing old and new — the diffing algorithm, keyed by element type and key attribute.
5. Hydration — waking up server HTML — attaching client event listeners and state to markup already present in the document.
6. Signal — a value that knows who reads it — a fine-grained reactive primitive that updates only its dependents.

35.8 Build tooling: why frontend code needs a build step

■ 35.8.1 PLAIN - in simple words

1. Twenty years ago you wrote a `.js` file and put it on a server. Done.
2. Today most projects run a program over the code first, and ship the result. That program is the **build step**.
3. There are four honest reasons for it.
4. One, your code is split across hundreds of small files, and loading hundreds of files is slow. So they are joined into a few. That is **bundling**.
5. Two, you write modern syntax, and some browsers do not understand all of it. So it is rewritten into older syntax. That is **transpiling**.
6. Three, some files are not JavaScript at all: JSX, TypeScript, Svelte components, Sass. Something must convert them.
7. Four, shipping fewer bytes makes pages faster, so names are shortened, spaces removed and unused code deleted.
8. Nothing here is magic. The build step is a translator and a packer.

■ 35.8.2 PLAIN - a picture in your head

1. Think of moving house.
2. Your belongings are spread through many rooms in many small containers. That is your source code: many files, each doing one thing, easy for you to find.
3. You would not carry two hundred small boxes to the van one at a time. You pack them into a few large cartons. That is bundling.
4. Anything broken or never used gets thrown out rather than moved. That is tree shaking.
5. Air is squeezed out and clothes are vacuum-packed. That is minification.
6. And you keep an inventory sheet saying which carton each item is in, so that when something is missing at the far end you can trace it back. That is the source map.

Where this comparison breaks:

7. When you move house, you unpack. The browser never unpacks: it runs the packed form. The source map only exists for you, in the debugger.
8. And you can add a room after moving in. Bundles are fixed at build time, which is why code splitting has to be planned, not discovered.

■ 35.8.3 PLAIN – a worked example

1. A tiny `package.json`, which is the project's identity card.

```
{
  "name": "bookshop-web",
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "test": "vitest run"
  },
  "dependencies": { "react": "^19.2.0" },
  "devDependencies": { "vite": "^8.2.0" }
}
```

2. `dependencies` are needed to run. `devDependencies` are needed only to build and test, and never reach the browser.
3. `^19.2.0` means “19.2.0 or any later 19.x”. That range is why two people can install the same project on the same day and get different code.
4. The **lock file**, `package-lock.json` or `pnpm-lock.yaml`, records the exact version and content hash of every package that was actually installed.
5. Commit the lock file. Without it your builds are not reproducible, and a working build today can break tomorrow with no change from you.
6. Tree shaking, shown concretely:

```
// utils.js exports 40 functions
import { formatDate } from './utils.js';
```

7. Because ES modules declare their imports statically, the bundler can prove the other 39 are unreachable and drop them. CommonJS require is dynamic, so it usually cannot.
8. Code splitting turns one big file into several loaded on demand:

```
const Chart = lazy(() => import('./Chart.jsx'));
```

9. Now the chart's code downloads the first time a user opens the chart, not on every page load.

■ 35.8.4 PLAIN – what is really happening inside

1. A bundler starts at an entry file, follows every import, and builds a graph of modules. Then it writes that graph out as one or more files.
2. A transpiler parses source into a syntax tree, rewrites nodes, and prints new source. Babel does this for JavaScript syntax.
3. Transpiling handles *syntax*. Missing *functions* need a **polyfill**, which is real code implementing the missing feature. These are different jobs.
4. TypeScript adds a type layer on top of JavaScript. It checks your code, then erases every type annotation. Nothing is checked at runtime.
5. That erasure is the key point people miss. TypeScript protects you while you write. It gives you nothing at all against bad data arriving from a network request. You still have to validate.
6. Minification renames local variables to single letters, removes whitespace and comments, and simplifies expressions. Typical saving is 30 to 60 percent before compression.
7. A source map is a separate file mapping every position in the output back to a line and column in your source, so the debugger can lie helpfully.

8. `node_modules` is where installed packages live, honestly: a very large folder containing your dependencies and all of their dependencies, often tens of thousands of files and hundreds of megabytes.
9. It is not a virus and it is not waste by design. It is the visible cost of a culture of very small packages. It should never be committed to git, and it is always rebuildable from `package.json` plus the lock file.

■ 35.8.5 TECHNICAL – the engineer’s version

1. npm launched in January 2010, written by Isaac Schlueter, and is now operated by GitHub. The registry holds well over three million packages.
2. Alternatives: Yarn (2016), pnpm (2017, which hard-links a global store and cuts disk use sharply), and Bun’s built-in installer (2022).
3. Semantic versioning is MAJOR.MINOR.PATCH. `^1.2.3` allows minor and patch updates, `~1.2.3` allows patch only, and an exact pin allows neither. This is a convention, and packages break it regularly.

Tool	First release	Written in	Role
webpack	2012, T. Koppers	JavaScript	bundler, loaders
Rollup	2015, R. Harris	JavaScript	library bundler
Babel	2014, S. McKenzie	JavaScript	transpiler
esbuild	2020, E. Wallace	Go	very fast bundler
Vite	2020, Evan You	JS plus Rust	dev server, build
SWC	2019	Rust	transpiler, minifier

4. Vite’s development server does not bundle. It serves native ES modules and transforms each file on request, so start-up is near instant regardless of project size. For production it bundles with Rollup.
5. Speed differences are large and measurable: esbuild and SWC are commonly 10 to 100 times faster than Babel and webpack on the same input, because they are compiled languages with parallelism rather than single-threaded JavaScript.
6. TypeScript was released by Microsoft on 1 October 2012, designed by Anders Hejlsberg. TypeScript 7, a native port of the compiler to Go, shipped in 2026 with roughly a tenfold type-checking speed-up.
7. Supply chain risk is real. Compromised packages have shipped malware to millions of installs. Mitigations: lock files, `npm audit`, `npm ci` in CI, `--ignore-scripts`, and provenance attestations.
8. Bundle budgets that hold up in practice: under 100 KB of compressed JavaScript for a content site, under 300 KB for a rich application. Measure compressed transfer size, not the raw file size.
9. Tools: `npx vite build --sourcemap`, `rollup-plugin-visualizer`, Chrome DevTools Coverage panel to find unused bytes, `npm ls <pkg>` to find who pulled a package in.

■ 35.8.6 WORDS – remember these

1. Bundler — the packer — a tool that resolves a module graph and emits fewer, larger output files.
2. Transpile — rewrite new syntax as old — source-to-source compilation between language versions.
3. Polyfill — supply a missing function — runtime code implementing a standard API absent in the host.
4. Tree shaking — drop code nobody uses — dead-code elimination over a static ES module graph.

5. Source map — the inventory sheet for debugging — a JSON mapping from generated positions back to original source.
6. Lock file — the exact recipe of what was installed — a resolved dependency tree with versions and integrity hashes.
7. Code splitting — download it when needed — emitting separate chunks loaded by `dynamic import()`.

35.9 The backend: what a server program actually does

■ 35.9.1 PLAIN - in simple words

1. A server program is an ordinary program with one unusual habit: it never finishes.
2. It opens a port, which is a numbered door on the machine, and waits.
3. When a request arrives, it reads it, decides what it means, does the work, writes an answer, and goes back to waiting.
4. Deciding what a request means is **routing**: matching the method and path, such as `GET /books/12`, to the piece of code that handles it.
5. The piece of code that handles it is called a **handler**, or in some traditions a **controller**.
6. Work that must happen on many requests, such as logging, checking who the user is, or parsing the body, is factored out into **middleware**: small steps a request passes through on the way in.
7. There are two shapes of backend. One builds finished HTML pages and sends them. The other sends only data, usually JSON, and lets the frontend draw.
8. The first is templating. The second is API-only. Both are fine, and mixing them is normal.

■ 35.9.2 PLAIN - a picture in your head

1. Think of a hospital reception.
2. The door is the port. Anyone may walk in.
3. Before anyone sees a doctor they pass a fixed sequence: security check, then registration, then the file is pulled, then vitals are taken. Every patient, every time, in that order. That is middleware.
4. Reception then reads the complaint and sends the patient to the right department. That is routing.
5. The specialist who actually treats them is the handler.
6. The discharge letter handed back is the response.
7. If registration fails, the patient never reaches the specialist, and is sent away at that step. That is middleware returning early, which is exactly how authentication is enforced.

Where this comparison breaks:

8. A hospital treats patients one at a time per doctor. A server holds thousands of half-finished requests at once, interleaved.
9. And a hospital remembers you between visits. HTTP does not. Every arrival is a stranger until a cookie or token proves otherwise.

■ 35.9.3 PLAIN - a worked example

1. The same tiny API, in Node with Express and in Python with FastAPI.

```
// server.js - Node with Express 5
import express from "express";
const app = express();

app.use(express.json());           // middleware
app.use((req, res, next) => {      // logging
```

```

    console.log(req.method, req.url);
    next();
  });

  app.get("/books/:id", async (req, res) => {
    const book = await db.findBook(req.params.id);
    if (!book) return res.status(404).json({ error: "no" });
    res.json(book);
  });

  app.listen(3000);

# main.py - Python with FastAPI
from fastapi import FastAPI, HTTPException

app = FastAPI()

@app.get("/books/{book_id}")
async def read_book(book_id: int):
    book = await db.find_book(book_id)
    if book is None:
        raise HTTPException(status_code=404)
    return book

```

2. The shapes are the same: declare a path, name the variable part, do the work, return data or an error code.
3. FastAPI reads the `book_id: int` annotation and rejects `/books/abc` with a 422 before your function runs. Express does not; you must check yourself.
4. The request lifecycle, drawn:

```

TCP accept
-> parse HTTP request line and headers
-> middleware 1 (parse body)
-> middleware 2 (log)
-> middleware 3 (authenticate) -- may stop here
-> router match: GET /books/:id
-> handler runs, queries database
-> serialize response, set status and headers
-> write bytes, maybe keep connection alive

```

■ 35.9.4 PLAIN – what is really happening inside

1. Under the routing there is a socket, from Chapter 28. `listen` puts the process into the accepting state; the kernel queues incoming connections.
2. Two families of design handle concurrency.
3. **Thread or process per request:** each request gets its own worker, which is allowed to block. Simple to reason about. Costs memory per worker, roughly 0.5 to 8 MB of stack, so thousands of workers is expensive.
4. **Event loop:** one thread handles many connections by never blocking, using the same idea as the browser event loop in section 35.4. Cheap per connection, but one slow synchronous function stalls everyone.
5. Node.js uses the second. So do FastAPI with async handlers, Go's runtime with very cheap goroutines, and modern Java with virtual threads.
6. This is why a Node server is excellent for many connections doing little work each, such as an API or a chat gateway, and a poor choice for heavy number-crunching in the request path.

7. The way out for Node is not to fight it: move heavy work to a worker thread, a separate service, or a background job queue.
8. In every design, the database is usually the real bottleneck, not the language. A 40 ms query dwarfs a 0.4 ms difference in framework overhead.

■ 35.9.5 TECHNICAL – the engineer’s version

1. Node.js was released by Ryan Dahl in 2009 and presented at JSConf EU in November 2009. It pairs V8 with libuv for asynchronous input and output. Node 24 “Krypton” is the active LTS as of 2026; Node 26 became the current line in May 2026.
2. Node is single-threaded for JavaScript but not single-threaded overall: libuv keeps a thread pool, four by default, for file system work, DNS and some crypto.
3. Express began in 2010 by TJ Holowaychuk; Express 5.0 was released in 2024 after a very long wait. Fastify and Hono are faster modern alternatives. Deno (2018) and Bun (2022) are competing runtimes.

Stack	Since	Concurrency	Strong at
Node.js	2009	event loop	APIs, realtime, one language
Django	2005	threads, async	admin, ORM, batteries
FastAPI	Dec 2018	async, ASGI	typed APIs, auto docs
Spring Boot	Apr 2014	threads, virtual	large enterprise systems
Go	2009, 1.0 2012	goroutines	throughput, single binary
Rails	July 2004	threads	fast product iteration
Laravel	June 2011	processes	shared hosting, PHP shops
ASP.NET Core	June 2016	async tasks	Windows shops, strong tooling

4. Django was created at the Lawrence Journal-World by Adrian Holovaty and Simon Willison and released publicly in July 2005; Django 6.1 is current. FastAPI was released by Sebastian Ramirez in December 2018.
5. Ruby on Rails was extracted from Basecamp by David Heinemeier Hansson in July 2004 and popularized convention over configuration. Rails 8.1 is current in 2026.
6. Go was announced by Google on 10 November 2009 (Robert Griesemer, Rob Pike, Ken Thompson) with 1.0 in March 2012; Go 1.26 is current. A goroutine starts at about 2 KB of stack, versus about 1 MB for an operating system thread.
7. Spring Framework began in 2003 from Rod Johnson’s book; Spring Boot arrived in April 2014 and Spring Boot 4.1 is current in 2026. Java 25, released 16 September 2025, is the newest long-term-support release.
8. PHP dates from 1995 (Rasmus Lerdorf); PHP 8.5 arrived November 2025. Laravel was released by Taylor Otwell in June 2011 and is now at version 13.
9. .NET Core 1.0 shipped June 2016 and unified with .NET 5 in November 2020; .NET 10, released 11 November 2025, is the current long-term-support version.
10. Honest guidance: pick the language your team already knows. Every framework in that table serves millions of users somewhere. The differences that matter are hiring, libraries and operational familiarity, not benchmarks.

11. Tools: `curl -i`, `httpie`, `ab` and `wrk` for load, `strace` or `dtruss` for syscalls, and `clinic.js` or `pprof` for profiling.

■ 35.9.6 WORDS – remember these

1. Route — the address-to-code map — a match on HTTP method and path pattern dispatching to a handler.
2. Middleware — a step every request passes through — a composable function in the request pipeline that may short-circuit the response.
3. Handler — the code that answers — the terminal function producing a status, headers and body; also called a controller action.
4. Templating — the server builds the HTML — server-side rendering of markup from data plus a template language.
5. Event loop server — one worker, never waiting — non-blocking I/O multiplexed with `epoll`, `kqueue` or `IOCP` through `libuv` or similar.
6. Port — a numbered door on a machine — a 16-bit TCP or UDP endpoint identifier bound by a listening socket.

35.10 Databases

■ 35.10.1 PLAIN – in simple words

1. Programs forget everything when they stop. A database is the place data goes so it survives, and so many users can share it safely.
2. You could use files. People do, and then discover the hard parts: two users writing at once, finding one record among ten million, and a crash halfway through an update.
3. A database solves exactly those three problems: concurrency, fast search, and never being left half-done.
4. The oldest and still best default is the **relational** database: data stored as tables of rows and columns, queried with a language called SQL.
5. A **table** is one kind of thing, such as books. A **row** is one of them. A **column** is one fact about it.
6. A **primary key** is the column that uniquely names a row. A **foreign key** is a column in one table holding the key of a row in another, which is how tables are joined together.
7. **NoSQL** is a loose family of databases that gave up some of those rules for speed, scale or a friendlier shape. Sometimes that is the right trade. Often it is not.
8. If you have no strong reason otherwise, start with PostgreSQL. This is not fashion; it is the advice most experienced engineers give.

■ 35.10.2 PLAIN – a picture in your head

1. Think of a library's paper records before computers.
2. One drawer holds a card per book: title, author number, year. Another drawer holds a card per author: number, name, country.
3. The author's name is written once, in the author drawer. The book card only stores the author's number.
4. That is **normalization**, and the reason for it is simple: if an author's name is wrong, you fix one card, not four hundred.
5. Looking up a book by title is fast because the drawer is sorted and you can jump to the right divider. That sorted arrangement is an **index**.
6. Without an index you would read every card in the drawer. With one you touch about twenty. That gap is the whole reason indexes exist.

7. And when a book moves between branches, both the “removed from A” and “added to B” cards must change together, or the book vanishes. Doing both or neither is a **transaction**.

Where this comparison breaks:

8. Library cards do not lie to each other. Two clerks writing at the same moment can produce a state neither intended, which is why isolation levels exist and why they are subtle.
9. And a card drawer holds thousands. A real index handles billions with roughly the same number of steps, because it is a tree, not a list.

■ 35.10.3 PLAIN – a worked example

1. Start with a badly designed table. One row per sale.

id	book	author	author_country	qty
1	Dune	F. Herbert	US	2
2	Emma	J. Austen	UK	1
3	Dune	F. Herbert	US	3

2. Three faults. The author’s country is repeated; if it is wrong it is wrong in many places. You cannot record an author with no sales. And updating one row can leave the data disagreeing with itself.
3. Normalized into three tables:

```
CREATE TABLE authors (
  id      SERIAL PRIMARY KEY,
  name    TEXT NOT NULL,
  country CHAR(2) NOT NULL
);

CREATE TABLE books (
  id      SERIAL PRIMARY KEY,
  title   TEXT NOT NULL,
  author_id INT NOT NULL REFERENCES authors(id)
);

CREATE TABLE sales (
  id      SERIAL PRIMARY KEY,
  book_id INT NOT NULL REFERENCES books(id),
  qty     INT NOT NULL CHECK (qty > 0),
  sold_at TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

4. Now a real query: total copies sold per country, best first, ignoring countries with fewer than 10 sales.

```
SELECT a.country,
       SUM(s.qty) AS copies
FROM sales s
JOIN books  b ON b.id = s.book_id
JOIN authors a ON a.id = b.author_id
WHERE s.sold_at >= '2026-01-01'
GROUP BY a.country
HAVING SUM(s.qty) >= 10
ORDER BY copies DESC
LIMIT 5;
```

5. JOIN follows the foreign keys to stitch the three tables back together. GROUP BY collapses many rows into one per country. WHERE filters rows before grouping; HAVING filters groups after. That difference is examined constantly and understood rarely.
6. Now indexes, with real arithmetic. A sales table with 10 million rows.
7. Without an index, finding rows for one book_id reads all 10,000,000 rows. At perhaps 2 million rows per second in memory, that is about 5 seconds.
8. With a B-tree index, the rows are held in a sorted tree with a high branching factor. If each node holds about 200 keys, then 200^3 is 8 million and 200^4 is 1.6 billion.
9. So four levels cover the table. The lookup reads four pages instead of ten million. That is roughly a million times less work, and it is why the answer comes back in under a millisecond.

```
CREATE INDEX idx_sales_book ON sales (book_id, sold_at);
EXPLAIN ANALYZE
SELECT * FROM sales WHERE book_id = 42;
```

10. EXPLAIN prints the plan the database intends to use. Seq Scan means it is reading everything; Index Scan means it is using the tree. Adding ANALYZE actually runs it and reports real timings and row counts.
11. The single most useful debugging habit in backend work is comparing the planner's estimated rows with the actual rows. A large gap means the statistics are stale, and the plan is probably bad.

■ 35.10.4 PLAIN - what is really happening inside

1. **ACID**, one letter at a time.
2. **A, Atomicity.** A transaction happens completely or not at all. Money leaving one account and arriving in another either both happen or neither does. A crash in between leaves no half-state.
3. **C, Consistency.** The database never ends a transaction in a state that breaks the rules you declared: foreign keys, uniqueness, checks.
4. **I, Isolation.** Concurrent transactions do not see each other's unfinished work. How strictly is a setting, and that is the interesting part.
5. **D, Durability.** Once the database says "committed", the data survives an immediate power cut, because it was written to a log on disk and flushed.
6. Isolation levels, weakest to strongest, and the anomaly each one stops:

Level	Prevents	Still allows
Read uncommitted	nothing	dirty reads
Read committed	dirty reads	non-repeatable reads
Repeatable read	non-repeatable reads	phantoms (in theory)
Serializable	all of the above	nothing, but retries

7. A **dirty read** is seeing another transaction's uncommitted change. A **non-repeatable read** is reading the same row twice and getting different values. A **phantom** is running the same query twice and getting a new row.
8. PostgreSQL defaults to read committed and never permits dirty reads at all. Its repeatable read also prevents phantoms, because it uses snapshots. MySQL's InnoDB defaults to repeatable read. These are implementation details worth checking for your engine.
9. **Connection pooling.** Opening a database connection is expensive: a TCP handshake, authentication, and on PostgreSQL a new operating system process, costing several megabytes.

10. So the application keeps a small set of open connections and lends them out. A pool of 10 to 30 usually serves far more concurrent users than that, because each request holds a connection for only milliseconds.
11. The classic production failure is one pool per instance times many instances, exhausting the server's connection limit. PgBouncer exists to sit in the middle and fix exactly that.

■ 35.10.5 TECHNICAL – the engineer's version

1. The relational model comes from Edgar F. Codd's 1970 paper at IBM, "A Relational Model of Data for Large Shared Data Banks". SEQUEL was designed by Donald Chamberlin and Raymond Boyce in 1974 and renamed SQL.
2. SQL was standardized by ANSI in 1986 and ISO in 1987; the current edition is SQL:2023. Every engine adds extensions, so portable SQL is a discipline.
3. B-trees come from Rudolf Bayer and Edward McCreight in 1970, published 1972. Databases use B+ trees, where all values sit in the leaves and leaves are linked, making range scans cheap.
4. ACID as a concept is Jim Gray's, 1981; the acronym was coined by Theo Harder and Andreas Reuter in 1983.
5. PostgreSQL descends from POSTGRES, begun by Michael Stonebraker at Berkeley in 1986; Postgres95 added SQL, and the name PostgreSQL dates from 1996. PostgreSQL 18 was released on 25 September 2025.

Type	Example	Shape	Good at
Relational	PostgreSQL 18	tables, SQL	almost everything
Document	MongoDB 8.3	JSON documents	flexible schema
Key-value	Redis 8.10	key to value	cache, counters, queues
Wide-column	Cassandra 5	partitioned rows	huge write volume
Graph	Neo4j	nodes and edges	relationship traversal
Search	Elasticsearch	inverted index	full-text ranking
Time-series	TimescaleDB	time-partitioned	metrics, sensors

6. Origins: MongoDB from 10gen in 2009 (Dwight Merriman, Eliot Horowitz); Redis in 2009 by Salvatore Sanfilippo; Cassandra at Facebook in 2008, Apache in 2010; Neo4j in 2007; Elasticsearch in 2010 by Shay Banon.
7. Redis changed licence in March 2024, which produced the Valkey fork under the Linux Foundation, then adopted AGPLv3 with Redis 8 in May 2025. If you use a managed cache, check which one you are actually running.
8. **CAP theorem**, stated correctly. It was conjectured by Eric Brewer in a July 2000 keynote and proved by Seth Gilbert and Nancy Lynch in 2002.
9. It says: when a network **partition** occurs, a distributed system must choose between **consistency** (every read sees the latest write) and **availability** (every request gets a non-error answer).
10. The common misuse is to describe systems as "CP" or "AP" as a permanent personality, or to say you may "pick two of three". You do not choose partitions; the network chooses them for you. C versus A is a choice made only during a partition, often per operation.
11. Daniel Abadi's PACELC (2012) is the honest extension: else, when there is no partition, you still trade latency against consistency.
12. Five scenarios, and the honest answer for each.

Scenario	Choose	Why
Payments, orders	PostgreSQL	transactions decide correctness
Session cache	Redis	sub-ms reads, expiry built in
Product search	Elasticsearch	ranking, typo tolerance
Device telemetry	Timescale	time partitioning, retention
Social graph paths	Neo4j	multi-hop traversal

13. Note what is missing: no scenario says “MongoDB because we do not know the schema yet”. PostgreSQL has a `jsonb` column type with indexing, so you can keep flexible documents inside a relational database and still have joins and transactions when you need them.
14. Tools: `psql`, `EXPLAIN (ANALYZE, BUFFERS)`, `pg_stat_statements` for the slowest queries, `pgbench` for load, and the slow query log in MySQL.

■ 35.10.6 WORDS – remember these

1. Primary key — the unique name of a row — a column set with a uniqueness and not-null constraint identifying each tuple.
2. Foreign key — a pointer to another table’s row — a referential integrity constraint enforced on insert, update and delete.
3. Normalization — store each fact once — decomposing relations to remove update anomalies, usually to third normal form.
4. Index — the sorted shortcut — an auxiliary B+ tree or hash structure mapping key values to row locations.
5. Transaction — all or nothing — a unit of work bounded by `BEGIN` and `COMMIT` with ACID guarantees.
6. Isolation level — how much of others’ work you may see — the setting choosing which concurrency anomalies are permitted.
7. CAP theorem — under a network split, pick consistency or answers — during a partition a system cannot be both linearizable and available.
8. Connection pool — a reused set of open lines — a bounded cache of database connections shared across requests.

35.11 ORMs and data access

■ 35.11.1 PLAIN – in simple words

1. Your program thinks in objects. The database thinks in tables. Something has to translate.
2. An **ORM**, an Object-Relational Mapper, is a library that does that translation for you.
3. You write `user.orders` in your language and it writes SQL, runs it, and turns the rows back into objects.
4. The gain is real: less repetitive code, fewer typos, and safety against SQL injection because values are sent separately from the query text.
5. The cost is also real: it is easy to write one innocent line that causes hundreds of database round trips.
6. An ORM usually brings **migrations** too: small, ordered, versioned scripts that change the shape of the database and are applied in the same order on every machine.

■ 35.11.2 PLAIN – a picture in your head

1. An ORM is a translator at a meeting between two people who share no language.
2. For ordinary sentences the translation is faithful and saves everyone effort.
3. But if you ask an ambiguous question, the translator will produce something grammatical and wrong, and you will not notice because you cannot read the other language.
4. So a good engineer learns enough of the other language to check the translation, even when using the translator daily.

Where this comparison breaks:

5. A human translator gets tired. An ORM is perfectly consistent, which means a bad pattern is repeated identically ten thousand times a second.

■ 35.11.3 PLAIN – a worked example

1. The **N+1 query problem**, the single most common performance bug in backend work.

```
orders = Order.objects.filter(status="paid")    # 1 query
for o in orders:
    print(o.customer.name)    # 1 query per order
```

2. With 200 orders that is $1 + 200 = 201$ round trips. At 2 ms each, that is about 0.4 seconds of pure waiting, for data one query could fetch.
3. The fix is to tell the ORM to fetch the related rows in advance:

```
orders = (Order.objects
          .filter(status="paid")
          .select_related("customer"))    # 1 query, joined
```

4. Now it is one query with a JOIN. Roughly 2 ms instead of 400 ms.
5. The same fix has different names in different tools: `select_related` and `prefetch_related` in Django, `include` in Prisma, `joinedload` in SQLAlchemy, `includes` in Rails, `with` in Laravel.
6. The way to catch it is to log queries per request in development. If one page view issues 300 queries, you have found it.

■ 35.11.4 PLAIN – what is really happening inside

1. The ORM keeps a map from classes to tables and from attributes to columns, plus rules for relationships.
2. Accessing a relationship attribute that has not been loaded triggers a **lazy load**: a fresh query, issued right there in the loop. That is where N+1 comes from.
3. Many ORMs also keep an identity map and a unit of work, so the same row read twice gives the same object and changes are written in one flush at the end.
4. Migrations work by recording which scripts have run in a table inside the database itself. On deploy, the tool compares that list with the files on disk and applies the missing ones in order.
5. Never edit an applied migration. Write a new one. The applied list is the shared history and rewriting it desynchronizes environments.
6. Raw SQL is better when the query is the point: complex reporting, window functions, recursive queries, bulk updates, and anything where you need to control the plan.
7. Modern practice is to use the ORM for the ordinary 90 percent and drop to SQL for the 10 percent that matters, rather than choosing one forever.

■ 35.11.5 TECHNICAL – the engineer’s version

1. The term object-relational impedance mismatch describes the structural gap: inheritance, identity and associations have no direct relational equivalent.
2. Notable tools and dates: Hibernate for Java (2001, Gavin King), Rails ActiveRecord (2004), Django ORM (2005), SQLAlchemy (2006, Michael Bayer), Entity Framework (2008), Eloquent (2011), Prisma (2019, now version 7), Drizzle (2022).
3. Two design patterns, from Martin Fowler’s 2002 book Patterns of Enterprise Application Architecture: Active Record, where a row and its behaviour are one object, and Data Mapper, where they are separate.
4. Query builders such as Knex, jOOQ and Drizzle sit between raw SQL and a full ORM: typed composition, no object graph, no lazy loading, so no N+1.
5. Parameterized queries prevent SQL injection because the driver sends the query text and the values on separate paths, so a value can never become syntax. String concatenation into SQL is the vulnerability.
6. Migration tools: Alembic, Django migrations, Flyway, Liquibase, Prisma Migrate, Rails Active Record migrations, EF Core migrations.
7. Zero-downtime schema change is a separate discipline: add a nullable column, backfill in batches, start writing both, switch reads, then drop the old column, across several deploys. Never rename in one step.
8. Tools: `django-debug-toolbar`, SQLAlchemy `echo=True`, Prisma’s query event log, and `pg_stat_statements` to find the query issued most often rather than the slowest single one.

■ 35.11.6 WORDS – remember these

1. ORM — a translator between objects and tables — a mapping layer generating SQL and materializing rows as domain objects.
2. N+1 — one query, then one more per result — repeated lazy loads inside an iteration, fixed by eager loading or a join.
3. Migration — a versioned change to the schema — an ordered, recorded script applied idempotently across environments.
4. Lazy loading — fetch it only when touched — deferred query execution on first access to an unloaded association.
5. Parameterized query — values sent apart from the text — placeholders bound by the driver, preventing SQL injection by construction.

35.12 Authentication and sessions

■ 35.12.1 PLAIN – in simple words

1. **Authentication** is proving who you are. **Authorization** is deciding what you are allowed to do once you are known.
2. They are two different questions, they fail in two different ways, and mixing them up is one of the most common security bugs there is.
3. A site must never keep your password the way you typed it. If a site can email your old password back to you, it stored it wrongly.
4. Instead it stores a scrambled fingerprint of the password, produced by a one-way function called a **hash**.
5. One-way means you can go from password to fingerprint easily, and from fingerprint back to password essentially never.

6. It also mixes in a random extra value called a **salt**, different for every single user, so two people with the same password get different fingerprints.
7. And the hash used for passwords is made deliberately slow. That sounds mad, and it is the entire point.
8. Once you have logged in, the site has to remember you on the next click, because HTTP forgets everything between one request and the next.
9. There are two ways to remember. Keep a note on the server and hand you a ticket number: that is a **session**. Or hand you a sealed pass that you carry and the server does not store: that is a **token**.
10. When you click “sign in with Google”, you never give Google’s password to the site you are visiting. A protocol called OAuth 2.0 arranges that carefully.

■ 35.12.2 PLAIN – a picture in your head

1. Think of a cloakroom at a theatre, and a wristband at a festival.
2. A session is the cloakroom ticket. Your coat stays with the cloakroom staff and you carry nothing but a number.
3. The number means nothing on its own. Only the staff’s book turns number 47 into “the long grey coat belonging to that person”.
4. If you lose the ticket, the staff can cross out number 47 in the book instantly, and it stops working everywhere, at once.
5. A token is the festival wristband. Your name is printed on it and it carries a tamper-proof seal that any gate can check without phoning anyone.
6. That makes gates fast and independent. Ten gates can check ten thousand wristbands with no shared book at all.
7. But if you are thrown out of the festival, the wristband is still on your wrist. It keeps working until it expires, unless every gate is handed a list of banned wristbands - and a shared list is a cloakroom book again.

Where this comparison breaks:

8. A photograph of a wristband gets you nowhere at a real gate, because a human looks at your face. A copied token is not a picture of you. To the server it simply is you, completely, until the moment it expires.
9. A real cloakroom holds a few hundred coats in one room. A session store holds millions of tickets across many servers, which is why real systems keep it in Redis or a database rather than in one server’s memory.

■ 35.12.3 PLAIN – a worked example

1. Here is a bcrypt password record, the kind that sits in a users table. It is one string that carries four things at once.

```
$2b$12$KIXqf8vN1sT.u2WcQ7p0yeR3hZ...
```

\$2b\$	the algorithm and version (bcrypt)
12\$	the cost factor: $2^{12} = 4096$ key rounds
KIX...	the first 22 characters are the salt
R3h...	the rest is the hash itself

2. Notice there is no separate salt column. The salt travels inside the record, because it is not a secret. It only has to be unique and unpredictable.

3. Checking a login runs the same function again with the stored salt and cost, then compares the results. It never reverses anything.
4. The cost factor is a power of two, so each step up doubles the work. Rough times on one ordinary 2025 server core:

Cost	Rounds	Time per hash
8	256	about 6 ms
10	1,024	about 25 ms
12	4,096	about 100 ms
14	16,384	about 400 ms

5. Why slow is the goal: an attacker who steals the table wants to try billions of guesses. At cost 12 a single core manages about 10 guesses a second.
6. A fast hash such as SHA-256 runs at billions of guesses a second on a GPU. The same stolen table would fall in hours instead of centuries.
7. The salt kills the other attack. Without salts, an attacker hashes the top ten million common passwords once and looks up every user at the same time.
8. With a unique salt per user, that precomputed table is worthless, and the attacker must redo all the work separately for every single account.
9. Now authorization, with a worked role-based example. Three roles, four permissions, and one rule table.

Role	Read post	Edit any post	Delete user
reader	yes	no	no
editor	yes	yes	no
admin	yes	yes	yes

10. A user is given roles. A role is given permissions. Code never asks “is this user an admin”; it asks “does this user have post.edit”.

```

ROLE_PERMS = {
    "reader": {"post.read"},
    "editor": {"post.read", "post.edit"},
    "admin": {"post.read", "post.edit", "user.delete"},
}

def can(user, perm):
    return any(perm in ROLE_PERMS[r] for r in user.roles)

```

11. The difference matters when the rules change. Adding a “moderator” role who may edit but not delete is one line in the table.
12. If the code had said `if user.is_admin` in forty places, the same change is forty edits and you will miss one.
13. Authentication failed gives HTTP 401. Authorization failed gives HTTP 403. The 401 name in the standard is “Unauthorized”, which is a historical mistake; it really means unauthenticated.

■ 35.12.4 PLAIN - what is really happening inside

1. Session flow, step by step. You POST your email and password over HTTPS.
2. The server looks up the row, runs the hash function with the stored salt, and compares. It uses a constant-time comparison so timing leaks nothing.
3. On success it creates a random session identifier, at least 128 bits from a cryptographically secure generator, and stores it with your user id.
4. It sends that identifier back in a Set-Cookie header. The browser stores it and attaches it to every later request to that site automatically.
5. The cookie carries flags that do the real security work. HttpOnly hides it from JavaScript. Secure sends it only over HTTPS. SameSite controls whether other sites can cause it to be sent.
6. Logging out deletes the server record. The ticket is dead immediately.
7. Token flow, step by step. Same login, but the server returns a **JWT**, a JSON Web Token, and stores nothing.
8. A JWT is three base64url-encoded parts joined by dots: a header saying which algorithm, a payload of claims such as user id and expiry, and a signature.
9. The signature is computed by the server over the first two parts using a secret key. Anyone can read a JWT; only the key holder can forge one.
10. Read that again, because it is the mistake people make: a JWT is signed, not encrypted. Never put anything private in the payload.
11. To check a request, the server verifies the signature and the expiry. No database lookup at all, which is why tokens scale across many servers.
12. The cost is revocation. There is no record to delete, so a stolen token works until it expires. That is why access tokens are given minutes, not days.
13. The standard fix is a pair: a short access token, plus a long-lived **refresh token** kept in an HttpOnly cookie, which can be revoked and rotated.
14. Now the delegated login flow. The point of OAuth 2.0 is that the site you are visiting never sees your Google password, and never can.
15. The four parties: you (the resource owner), the site (the client), Google's login server (the authorization server), and Google's API (the resource server).

You	Site (client)	Auth server	API
--click sign in-->			
	--redirect---->		
<----- login page -----			
--password + consent----->			
	<--code (in URL redirect)--		
	--code+secret+verifier---->		
	<--access token + id token-		
	--token----->		
	<--your profile data-----		
<--logged in--			

16. Step 1: the site sends your browser to the authorization server with its client id, the scopes it wants, a redirect address, a random state value, and a code challenge.
17. Step 2: you log in on Google's own page, on Google's own domain, and approve the list of scopes. The site is not involved and cannot see any of it.
18. Step 3: Google redirects your browser back to the site with a short-lived **authorization code** in the URL.

19. Step 4: the site's server exchanges that code, over a direct back-channel call, for tokens. This step needs the client secret and the code verifier, which never touch the browser.
20. Step 5: the site calls the API with the access token to fetch the data it was granted, and nothing more.
21. Why the extra hop with a code, instead of just handing over a token? Because URLs leak. They land in browser history, server logs and referrer headers. A code is single-use and worthless without the secret.
22. OpenID Connect is a thin layer on top. OAuth 2.0 alone answers "may this site read your calendar". It does not answer "who are you".
23. OpenID Connect adds an **ID token**, a JWT about you specifically, plus a standard userinfo endpoint and the standard scopes openid, profile and email. That is the part that makes it a login system.

■ 35.12.5 TECHNICAL – the engineer's version

1. bcrypt was published by Niels Provos and David Mazieres at USENIX in 1999, built on the Blowfish cipher (Bruce Schneier, 1993). Its cost parameter is log-2, so cost 12 means 2^{12} key expansion rounds.
2. bcrypt's real limits: it truncates input at 72 bytes, and some implementations mishandle the NUL byte. Do not pre-hash into it without base64 encoding first, or you can shorten the effective input.
3. scrypt came from Colin Percival in 2009 and is specified in RFC 7914 (2016). It added memory hardness, to blunt custom hardware.
4. Argon2 won the Password Hashing Competition in July 2015. Authors: Alex Biryukov, Daniel Dinu and Dmitry Khovratovich. It is specified in RFC 9106 (September 2021). Use the Argon2id variant.
5. Current OWASP Password Storage Cheat Sheet settings, as of 2026:

Algorithm	Recommended setting
Argon2id	m=19456 KiB, t=2, p=1
bcrypt	cost factor 10 or more
scrypt	N= 2^{17} , r=8, p=1
PBKDF2-SHA256	600,000 iterations

6. A **pepper** is a site-wide secret mixed in before or after hashing, kept outside the database, ideally in a key management service. It protects against a database-only leak. It is optional; salting is not.
7. NIST SP 800-63B-4, "Digital Identity Guidelines: Authentication and Authenticator Management", published 31 July 2025, is the current reference.
8. It says: require at least 8 characters, accept at least 64, allow all printable ASCII and Unicode, and do not impose composition rules such as "one capital and one symbol".
9. It also says do not force periodic rotation without evidence of compromise, and do check new passwords against known-breached lists. Forced 90-day rotation is now explicitly discouraged, reversing decades of advice.
10. JWT is RFC 7519 (May 2015), sitting on JWS (RFC 7515), JWE (RFC 7516), JWK (RFC 7517) and JWA (RFC 7518). Bearer usage is RFC 6750.
11. The four classic JWT mistakes, all still seen in production:

Mistake	Why it is fatal
Storing in localStorage	any XSS reads it directly
No exp claim	the token never dies
Accepting alg: none	signature check skipped
Not pinning the algorithm	RS256 forced down to HS256

12. The alg: none and algorithm-confusion attacks were publicized by Tim McLean in March 2015. In the confusion attack the server is tricked into verifying an HMAC using the public RSA key as the shared secret, which the attacker also has. Always pin the expected algorithm on the verify side.
13. localStorage versus cookies, stated plainly: localStorage is readable by any script running on the page, so one cross-site scripting hole exports every token. An HttpOnly cookie is not reachable from JavaScript at all.
14. The cost of cookies is cross-site request forgery, which SameSite=Lax or Strict plus a CSRF token handles. Chrome made SameSite=Lax the default in version 80, February 2020.
15. OAuth 2.0 is RFC 6749 (October 2012), replacing OAuth 1.0 (RFC 5849, 2010). PKCE, Proof Key for Code Exchange, is RFC 7636 (September 2015).
16. RFC 9700, “Best Current Practice for OAuth 2.0 Security”, published 30 January 2025 as BCP 240, is now the document to follow. It requires PKCE for all clients including confidential ones, forbids the implicit grant and the resource owner password credentials grant, and requires exact string matching of redirect URIs.
17. OAuth 2.1 is an IETF draft, not yet an RFC as of 2026. It folds RFC 9700’s advice into one document. Say “draft”, not “standard”.
18. OpenID Connect Core 1.0 was published by the OpenID Foundation in February 2014. Discovery lives at /.well-known/openid-configuration; signing keys at a JWKS endpoint.
19. Single sign-on in enterprises is still largely SAML 2.0, an OASIS standard from March 2005, using signed XML assertions over HTTP POST. New systems use OIDC; you will meet SAML because it is entrenched, not because it is better.
20. Multi-factor authentication means two of: something you know, something you have, something you are. Two passwords are not two factors.

Factor	Standard	Real weakness
SMS code	none, carrier route	SIM swap, SS7
TOTP app	RFC 6238, 2011	phishable in real time
Push approve	vendor specific	fatigue, blind taps
Passkey	WebAuthn, FIDO2	device loss, recovery

21. TOTP (RFC 6238, May 2011) builds on HOTP (RFC 4226, December 2005): a 30-second time step, 6 digits, HMAC-SHA1 by default over a shared secret.
22. WebAuthn became a W3C Recommendation on 4 March 2019, with Level 2 in April 2021. Passkeys are discoverable WebAuthn credentials synced by the platform.
23. Passkeys are phishing-resistant by construction: the credential is bound to the origin, so a look-alike domain simply gets no signature. That property is an established fact from the protocol, not a marketing claim.
24. RBAC was formalized by David Ferraiolo and Richard Kuhn at NIST in 1992 and by Ravi Sandhu and colleagues in 1996, and standardized as INCITS 359-2004.

25. ABAC, attribute-based access control (NIST SP 800-162, 2014), decides from attributes of subject, object, action and environment. It is more expressive and much harder to audit. Most teams want RBAC with a few ownership checks.
26. Tools: `bcrypt`, `argon2-cffi`, `libsodium`; `jwt.io` and the `jose` libraries for inspecting tokens; `openssl rand -base64 32` for secrets; Keycloak, Auth0, Okta and Entra ID as identity providers.

■ 35.12.6 WORDS – remember these

1. Authentication — proving who you are — verifying a claimed identity against one or more authenticators.
2. Authorization — deciding what you may do — evaluating a policy over subject, action and resource after identity is established.
3. Salt — a per-user random extra — a unique non-secret value that defeats precomputed rainbow tables.
4. Pepper — a site-wide secret extra — a key held outside the database and mixed into the hash input or output.
5. Work factor — how slow the hash is on purpose — the cost, memory and parallelism parameters tuned to a target verify time.
6. Session — a ticket the server remembers — server-side state keyed by an opaque high-entropy identifier held in a cookie.
7. JWT — a signed pass the user carries — a JSON Web Token, base64url header, payload and signature under RFC 7519.
8. Access token — the short-lived key to the API — a bearer credential with a minutes-long exp claim.
9. Refresh token — the long-lived key to get new keys — a revocable credential exchanged at the token endpoint, usually rotated on each use.
10. OAuth 2.0 — letting a site act for you without your password — a delegated authorization framework, RFC 6749, with PKCE per RFC 9700.
11. OpenID Connect — the login layer on top of OAuth — an identity layer adding an ID token and a userinfo endpoint.
12. RBAC — permissions attached to job titles — role-based access control per INCITS 359-2004, users to roles to permissions.

35.13 Application architecture

■ 35.13.1 PLAIN – in simple words

1. Architecture is the set of decisions that are expensive to change later. It is not about which framework you pick.
2. A **monolith** is one program that does everything. One codebase, one build, one deploy, usually one database.
3. **Microservices** means the same work split into many small programs that talk over the network, each with its own deploy and often its own database.
4. The honest advice, given by most experienced engineers, is this: start with a monolith. Almost every team should.
5. Inside any of them, the code is arranged in **layers**, so that the part drawing the screen is not also the part talking to the database.
6. Some answers are expensive to compute and rarely change, so we keep a copy ready to hand. That copy is a **cache**.

7. Some work does not need to happen while the user waits. We drop a note into a **queue** and a separate worker picks it up later.
8. Sometimes the server needs to tell the browser something without being asked, which normal request-and-reply cannot do.
9. And because anyone can call your server as fast as they like, you have to cap how often each caller may do it. That is **rate limiting**.

■ 35.13.2 PLAIN – a picture in your head

1. Think of a restaurant.
2. A monolith is one kitchen. Everybody cooks in the same room, shares the same fridge, and shouts across the pass when something is ready.
3. Communication is instant and free. Nobody has to book a phone call to ask whether the sauce is done.
4. When the room gets too crowded, you cannot make just the grill section bigger. You have to build a second identical kitchen.
5. Microservices are several small kitchens in separate buildings: one does bread, one does sauces, one does desserts.
6. Each can be scaled, replaced or rebuilt without closing the others. But every request between them is now a phone call that can be busy, slow or dropped.
7. The caching layer is the pass with finished dishes under a heat lamp. Serving from there takes seconds instead of minutes.
8. The queue is the ticket rail. Orders go up, cooks take them down in order, and nothing is lost if one cook steps away.

Where this comparison breaks:

9. A kitchen phone call fails loudly and immediately. A network call can succeed at one end and be lost at the other, so the sender never learns which.
10. Real kitchens do not silently duplicate an order because the first acknowledgement went missing. Distributed systems do exactly that, which is why every message consumer must be safe to run twice.

■ 35.13.3 PLAIN – a worked example

1. A product page that shows a price, a stock count and a review average.
2. Version one asks the database on every single request. At 500 requests a second, that is 1,500 queries a second for data that changes hourly.
3. Version two caches the assembled result in Redis for 60 seconds.

```
def product(pid):
    key = f"product:{pid}"
    hit = redis.get(key)
    if hit:
        return json.loads(hit)          # about 0.5 ms
    data = build_from_db(pid)           # about 40 ms
    redis.setex(key, 60, json.dumps(data))
    return data
```

4. With a 95 percent hit rate the average response time falls from about 40 ms to about 2.5 ms, and the database sees 75 queries a second instead of 1,500.

Layer	Typical hit time	Who it serves
CDN edge	5 to 30 ms	everyone, worldwide
HTTP browser cache	0 ms	one returning user
Redis	0.3 to 1 ms	all app servers
In-process memory	0.0001 ms	one server only
PostgreSQL	1 to 50 ms	the source of truth

- Now the queue. When an order is placed we must charge the card, send an email, generate a PDF invoice and update a search index.
- Doing all four inline makes the user wait about 3 seconds and couples the checkout button to the email provider being awake.
- Doing only the charge inline, and pushing the other three onto a queue, gives a 200 ms response and a checkout that survives the email provider going down.

```
@app.post("/orders")
def create_order(req):
    order = save_order(req)      # inline, must be certain
    charge_card(order)          # inline, user needs the answer
    queue.enqueue("send_receipt", order.id)
    queue.enqueue("build_invoice", order.id)
    queue.enqueue("index_order", order.id)
    return {"id": order.id}      # about 200 ms
```

- The rule that makes this safe: the worker must be **idempotent**, meaning running it twice has the same effect as running it once.
- So `send_receipt` records that it sent receipt number 5512 before it exits, and checks that record on entry. Otherwise a retry mails the customer twice.

■ 35.13.4 PLAIN – what is really happening inside

- Layers, from the outside in. The transport layer receives HTTP and knows nothing about your business. The service or application layer holds the rules. The data access layer talks to the database. The domain holds the concepts.
- The discipline is that dependencies point one way, inward. The database layer must never import the HTTP layer.
- MVC** is the oldest version of this idea. The Model holds data and rules, the View renders, the Controller receives input and coordinates.
- In a modern web app the View often lives in the browser and the Controller is an HTTP route handler, which is why the term drifted.
- Caching happens at four separate places and they are independent: in the browser, at a CDN, in a shared store like Redis, and inside the process.
- HTTP caching is negotiated with headers. `Cache-Control: max-age=3600` says how long a copy stays fresh. `ETag` gives a version fingerprint, so a stale copy can be revalidated with a cheap 304 Not Modified instead of a resend.
- A **CDN** is a network of servers placed near users. It caches your responses physically close to the reader, which cuts the speed-of-light cost that no amount of server tuning can remove.
- The hard part of caching is not storing. It is knowing when a stored answer became a lie. There are two strategies: expire on a timer, or actively delete the entry when the underlying data changes.
- Queues decouple in time. The producer writes a message and returns. The broker stores it durably. A consumer takes it, does the work, and acknowledges.

10. If the consumer crashes before acknowledging, the message reappears. That is at-least-once delivery, and it is why duplicates happen.
11. Failed messages retry with growing gaps, and after a limit are moved to a **dead-letter queue** so one poisoned message cannot block the line forever.
12. Server-to-browser messaging has three shapes. Polling asks repeatedly. Server-sent events keeps one HTTP response open and streams text down it. WebSockets upgrades the connection to a two-way channel.
13. Rate limiting works by counting requests per key, usually per API key or per IP address, and refusing once the count passes a threshold in a time window.

■ 35.13.5 TECHNICAL – the engineer’s version

1. Conway’s Law, from Melvin Conway’s 1967 paper “How Do Committees Invent?”, states that a system’s structure copies the communication structure of the organization that built it. Microservices without matching team boundaries produce a distributed monolith, which has every cost and no benefit.
2. The term microservices was popularized by James Lewis and Martin Fowler in March 2014. Fowler’s own follow-up, “MonolithFirst” (June 2015), argues that teams who started monolithic and split later did better than those who began distributed.
3. The honest trade-off table:

Concern	Monolith	Microservices
Local call cost	nanoseconds	milliseconds, can fail
Deploy	one unit, all at once	independent per service
Transaction	one ACID commit	sagas, eventual consistency
Debugging	one stack trace	distributed tracing needed
Scaling	whole app together	per service, precisely
Team size that fits	1 to 30	many teams, own on-call

4. Amazon’s Prime Video team published in March 2023 that moving an audio-and-video monitoring service from distributed serverless components back into a single process cut infrastructure cost by over 90 percent. It is one case, not a law, but it punctured the idea that splitting always wins.
5. A **modular monolith** is the current mainstream advice: one deployable unit with strictly enforced internal module boundaries, so a later split is mechanical rather than archaeological.
6. MVC comes from Trygve Reenskaug at Xerox PARC in 1979, in Smalltalk-80. Related patterns: MVP, MVVM (Microsoft, 2005), and hexagonal architecture (Alistair Cockburn, 2005).
7. HTTP caching is specified in RFC 9111 (June 2022), which replaced RFC 7234. Key directives: max-age, s-maxage for shared caches only, no-store, private, must-revalidate, and stale-while-revalidate from RFC 5861.
8. Validators are ETag with If-None-Match, and Last-Modified with If-Modified-Since. A match returns 304 with no body.
9. Cache patterns: cache-aside (read through the app, the code above), write-through, write-behind, and refresh-ahead. Cache-aside is the default because it is the only one that is simple to reason about.
10. A **cache stampede** happens when a popular key expires and a thousand requests all miss and all rebuild it at once. Fixes: a mutex or single-flight lock, probabilistic early expiry, or stale-while-revalidate.

11. Akamai, founded in 1998 out of MIT work by Tom Leighton and Daniel Lewin, created the commercial CDN. Cloudflare followed in 2009, Fastly in 2011.
12. Message brokers: RabbitMQ (2007, implementing AMQP 0-9-1), Apache Kafka (built at LinkedIn by Jay Kreps, Neha Narkhede and Jun Rao, open sourced in 2011), Amazon SQS (generally available 2006), NATS, and Redis Streams.
13. Kafka is a distributed append-only log with consumer offsets, so replay is normal. RabbitMQ is a broker that deletes on acknowledgement. They solve different problems and are frequently confused.
14. Job frameworks by language: Celery and RQ for Python, Sidekiq for Ruby, BullMQ for Node, Quartz and Spring Batch for Java, Hangfire for .NET.
15. Retry policy in practice: exponential backoff with full jitter, a maximum attempt count, and a dead-letter queue. Without jitter, retries synchronize and produce a thundering herd against a recovering service.
16. The real-time comparison:

Property	Polling	SSE	WebSocket
Direction	client asks	server to client	both ways
Protocol	plain HTTP	plain HTTP	upgrade, RFC 6455
Reconnect	trivial	automatic	you write it
Binary data	yes	no, UTF-8 text	yes
Proxy trouble	none	rare	occasional
Good for	slow updates	feeds, tokens, logs	chat, games, cursors

17. WebSocket is RFC 6455 (December 2011); it starts as an HTTP request with Upgrade: websocket and then leaves HTTP framing entirely. RFC 8441 (2018) added bootstrapping over HTTP/2.
18. Server-sent events are part of the HTML Living Standard, exposed as the EventSource API, with Content-Type: text/event-stream. The browser reconnects on its own and resumes with Last-Event-ID.
19. SSE is the quiet default for AI token streaming, dashboards and progress bars, because it is ordinary HTTP and needs no special infrastructure.
20. Rate limiting algorithms and what they actually do:

Algorithm	Behaviour
Fixed window	simple, allows 2x burst at edge
Sliding log	exact, memory grows with traffic
Sliding counter	near-exact, cheap, common
Token bucket	allows a controlled burst
Leaky bucket	smooths output to a fixed rate

21. The response is 429 Too Many Requests, defined in RFC 6585 (April 2012), with a Retry-After header from RFC 9110. The RateLimit-Limit and RateLimit-Remaining headers are widely used by convention but are still an IETF Internet-Draft, not an RFC, as of 2026.
22. Tools: redis-cli --stat and INFO stats for hit ratio, curl -I to read cache headers, nginx limit_req and Envoy for edge limiting, OpenTelemetry with Jaeger or Tempo for distributed tracing.

■ 35.13.6 WORDS – remember these

1. Monolith — one program that does it all — a single deployable unit sharing one process, build and usually one datastore.
2. Microservice — a small program with its own deploy — an independently deployable service owning its data behind a network API.
3. Modular monolith — one deploy, firm inner walls — enforced module boundaries inside a single artifact, splittable later.
4. Cache — a ready-made copy of a slow answer — a store trading staleness for latency, governed by a TTL or explicit invalidation.
5. CDN — servers near the user — a geographically distributed edge cache for static and cacheable dynamic responses.
6. ETag — a version fingerprint for a response — a validator enabling conditional requests and 304 Not Modified replies.
7. Queue — a durable to-do list between programs — a broker giving at-least-once delivery with acknowledgement and retry.
8. Idempotent — safe to run twice — an operation whose repeated application gives the same result as one application.
9. Dead-letter queue — the box for messages that keep failing — a sink for messages exceeding the retry limit, kept for inspection.
10. WebSocket — a two-way phone line — a full-duplex framed protocol over one TCP connection, RFC 6455.
11. Server-sent events — a one-way news feed — a long-lived HTTP response streaming text/event-stream with automatic reconnection.
12. Rate limit — a cap on how often you may ask — a per-key request budget over a window, refused with HTTP 429.

35.14 Mobile apps

■ 35.14.1 PLAIN – in simple words

1. A phone app can be built in three broadly different ways, and the choice decides most of what follows.
2. **Native** means writing in the language the phone maker intends, with the phone maker's own toolkit. Two apps, two codebases, two teams' worth of work.
3. **Cross-platform** means writing once in a shared language and letting a framework produce something that runs on both phones.
4. **Web-based** means it is really a website, wrapped so it can be installed and opened from the home screen. That is a progressive web app, a PWA.
5. On iPhone, native means Swift, usually with the SwiftUI toolkit. On Android, native means Kotlin, usually with Jetpack Compose.
6. The main cross-platform choices are React Native, which uses JavaScript and the real platform controls, and Flutter, which uses Dart and paints every pixel itself.
7. There is one more thing that has no equivalent on the web: a stranger at Apple or Google reads your app and decides whether it may exist.
8. On the web you deploy and it is live. On a phone store you submit, you wait, and you can be told no.
9. What you actually upload is a file. On Android it is an APK or an AAB. On iPhone it is an IPA. Both are, underneath, ordinary ZIP archives.

■ 35.14.2 PLAIN – a picture in your head

1. Think about writing a book that must be published in two countries with different languages and different censors.
2. Native is hiring two native writers, each producing a book that reads perfectly naturally in its own country. Best result, double the cost, and the two books drift apart over time.
3. Cross-platform is writing once and translating. React Native is a good human translator using each country's own idioms and typesetting. Flutter is printing your own book, with your own fonts and paper, and shipping it unchanged to both countries.
4. Flutter's version always looks identical everywhere, which is a feature if you want your brand, and a problem if readers expected the local style.
5. A PWA is publishing on a website that both countries can already read. No censor, no printing, instant updates, and no place on the bookshop shelf.

Where this comparison breaks:

6. A translated book cannot suddenly need a printing press the translator does not own. A cross-platform app frequently needs a device feature the framework has not wrapped yet, and then you write native code anyway.
7. And a censor reads a book once. App review happens on every single update, so it is a recurring tax on shipping, not a one-time gate.

■ 35.14.3 PLAIN – a worked example

1. The same screen, a counter with a button, written four ways. Notice how similar the modern declarative shapes have become.

```
// iOS native: Swift with SwiftUI
struct CounterView: View {
    @State private var n = 0
    var body: some View {
        VStack {
            Text("Count: \(n)")
            Button("Add") { n += 1 }
        }
    }
}

// Android native: Kotlin with Jetpack Compose
@Composable
fun Counter() {
    var n by remember { mutableStateOf(0) }
    Column {
        Text("Count: $n")
        Button(onClick = { n++ }) { Text("Add") }
    }
}

// Flutter: Dart
class Counter extends StatefulWidget { ... }
// inside State:
Column(children: [
    Text('Count: $n'),
    ElevatedButton(
        onPressed: () => setState(() => n++),
        child: const Text('Add')),
])
```

```
// React Native: JavaScript
function Counter() {
  const [n, setN] = useState(0);
  return (<View>
    <Text>Count: {n}</Text>
    <Button title="Add" onPress={() => setN(n + 1)} />
  </View>);
}
```

2. Now the honest comparison for a real product decision.

Approach	Performance	Device access
Native Swift or Kotlin	best available	everything, day one
Flutter	very close to native	good, plugin gaps
React Native	good, JS bridge cost	good, plugin gaps
PWA	fine for content	most limited

3. And the effort side, which is what actually decides most projects.

Approach	Effort for both phones	Team you need
Native	roughly 180 percent	iOS plus Android devs
Flutter	roughly 110 percent	Dart devs, some native
React Native	roughly 110 percent	web devs, some native
PWA	roughly 60 percent	your existing web team

4. The “180 percent” is not 200 because design, backend and product work are shared. The extra 10 percent on cross-platform is the native code you end up writing anyway for permissions, push and store plumbing.
5. A concrete rule that holds up: if the app is mostly forms, lists and content, any of these works and you should pick by team. If it is a camera, a game, a map with heavy custom rendering, or something that must feel invisible, go native.

■ 35.14.4 PLAIN – what is really happening inside

1. Native code is compiled ahead of time to machine code for the device’s processor and calls the operating system’s own UI objects directly. There is no layer in between, so there is no layer to be slow.
2. React Native runs your JavaScript in a JavaScript engine on the phone, and that JavaScript creates and updates real platform views. A Text really is a UILabel on iOS and a TextView on Android.
3. So React Native inherits the platform’s look and accessibility for free, and pays a cost whenever data crosses between JavaScript and the native side.
4. Flutter does the opposite. It ships its own rendering engine and draws every button, scrollbar and cursor itself onto a blank canvas.
5. That gives identical output on both platforms and total design control, and means Flutter must re-implement anything the platform changes, including accessibility behaviour and text selection.
6. A PWA is a website with three additions: a manifest file naming the app and its icons, a service worker script that can serve files while offline, and a requirement to be served over HTTPS.
7. The service worker is the key piece. It is a script that sits between the page and the network, can answer requests from a local cache, and keeps running briefly after the page closes.

8. Store review, step by step. You build a signed package, upload it, fill in metadata and screenshots, and it enters a queue.
9. Automated scanning runs first: malware checks, private API usage, permission declarations, size limits. Then, on Apple's store, a human opens the app.
10. The human checks the guidelines: does it do something useful, does it match its description, does it collect data it declared, does it route payments for digital goods through the store's own system.
11. Rejection is normal and is not the end. You get a message, you fix it, you resubmit, and the clock restarts.
12. What this means for shipping is the real point: you cannot fix a live bug in ten minutes. Plan for a review delay on every release, keep a server-side switch to turn broken features off without a new build, and never ship on a Friday hoping to patch on Saturday.
13. Now the files themselves, which ties directly to the file-format material in Chapter 20: a file's identity comes from the bytes at its start, not from its extension.
14. Open an APK in a hex viewer and the first four bytes are 50 4B 03 04, which is PK followed by two control bytes. That is the ZIP local file header signature, named after Phil Katz.
15. An APK is a ZIP. Rename it to .zip, unzip it, and you can read the whole structure. So is an IPA. So is a .jar, a .docx and an .epub.
16. Inside an APK you find compiled code, compiled resources, a compiled manifest and a signature folder. Inside an IPA you find a folder called Payload holding one .app bundle.

app.apk (ZIP)	app.ipa (ZIP)
+ AndroidManifest.xml	+ Payload/
(binary XML, not text)	+ MyApp.app/
+ classes.dex	+ MyApp (Mach-O)
+ resources.arsc	+ Info.plist (binary)
+ res/	+ Assets.car
+ lib/arm64-v8a/*.so	+ _CodeSignature/
+ assets/	+ embedded.mobileprovision
+ META-INF/ (signatures)	+ iTunesMetadata.plist

17. The manifest inside an APK is not readable text. It is Android binary XML, a compact encoding with a string pool, which is why opening it in a text editor shows mush. The tool `aapt2 dump badging` prints it properly.
18. `classes.dex` holds Dalvik Executable bytecode: all your Kotlin and Java classes merged into one register-based bytecode file, not the stack-based `.class` files the Java compiler first produced.
19. `Info.plist` inside an iOS app is a property list, usually stored in Apple's binary plist format starting with the bytes `bplist00`, not XML.

■ 35.14.5 TECHNICAL – the engineer's version

1. Swift was announced at Apple's WWDC in June 2014, designed by a team led by Chris Lattner, and open sourced in December 2015. Swift 6.3 is the current release in 2026, shipping with Xcode 26.
2. SwiftUI was announced at WWDC in June 2019. UIKit, from iPhone OS 2.0 in 2008, remains fully supported and is still required for some controls, so real apps mix both through `UIViewRepresentable`.
3. Kotlin came from JetBrains, announced in 2011, version 1.0 in February 2016. Google announced first-class Android support in May 2017 and a Kotlin-first policy in May 2019.
4. Jetpack Compose reached 1.0 in July 2021. The older approach, XML layouts with `findViewById` or view binding, is still everywhere in existing code.

5. React Native was open sourced by Facebook in March 2015. Its New Architecture, with the Fabric renderer and TurboModules replacing the old asynchronous bridge, became the default in version 0.76 (October 2024). Version 0.87 was released on 10 August 2026.
6. Flutter's first stable release, 1.0, was December 2018, after a 2017 alpha. Flutter 3.47 is current in 2026. It uses Dart, compiled ahead of time to native code for release builds, and the Impeller renderer, which replaced Skia as the default on iOS and on Android API 29 and above.
7. Other real options: Kotlin Multiplatform, stable since November 2023, which shares business logic while leaving the UI native; .NET MAUI (May 2022); Capacitor with Ionic; and Tauri for desktop.
8. Progressive web app is not a standard; it is a name for a set of standards used together: the Web App Manifest (a W3C specification), Service Workers (W3C), and a secure context requirement.
9. The term was coined by Frances Berriman and Alex Russell in 2015. Safari on iOS 16.4, released 27 March 2023, finally added Web Push and badging for home-screen web apps, closing the largest gap.
10. Remaining PWA gaps on iOS as of 2026: no Web Bluetooth, no Web NFC, limited background execution, storage that can be evicted, and no place in the App Store search results, which is a distribution problem, not a technical one.
11. App Store review facts. The App Store opened on 10 July 2008. Apple states that on average 90 percent of submissions are reviewed in less than 24 hours. That is an average, and complex or first-time submissions take longer.
12. Common rejection causes are guideline 2.1 (crashes and incomplete information), 4.2 (minimum functionality, which kills thin website wrappers), 3.1.1 (digital goods must use in-app purchase), and privacy label mismatches.
13. Commission is 30 percent, or 15 percent under the App Store Small Business Program for developers below one million US dollars a year, and 15 percent on subscriptions after a subscriber's first year.
14. The EU Digital Markets Act forced alternative app marketplaces and third-party browser engines on iOS in the EU from iOS 17.4, March 2024. This is jurisdiction-specific and still changing; check the current position rather than trusting any book, including this one.
15. Google Play review facts. Review is mostly automated with human escalation. New personal developer accounts must run a closed test with at least 12 testers for 14 continuous days before applying for production access, a rule introduced in November 2023 and later relaxed from 20 testers to 12.
16. Google Play has required the Android App Bundle for new apps since August 2021. Play requires apps to target API level 36, Android 16, by 31 August 2026.
17. An AAB is a publishing format, not an installable one. Play uses it to generate split APKs per device: one base, plus configuration splits for the device's screen density, CPU architecture and language. Typical download size saving is 15 to 20 percent.
18. APK signing schemes, and which Android version introduced each:

Scheme	Since	What it covers
v1 (JAR)	Android 1.0	each file, per entry
v2	Android 7.0, 2016	whole APK bytes
v3	Android 9, 2018	adds key rotation
v3.1	Android 13, 2022	rotation targeting
v4	Android 11, 2020	streaming install

19. iOS code signing uses a certificate plus a provisioning profile embedded as embed-

ded.mobileprovision. `_CodeSignature/CodeResources` holds a hash of every file in the bundle, so changing one byte after signing breaks launch.

20. App thinning means the store delivers only the slices a device needs. Apple calls the pieces app slicing, bitcode (deprecated with Xcode 14, 2022) and on-demand resources.
21. Inspection commands worth knowing:

```
unzip -l app.apk          # it really is a ZIP
aapt2 dump badging app.apk # readable manifest
apksigner verify -v app.apk # which schemes signed it
unzip -p app.ipa 'Payload/*.app/Info.plist' | plutil -p -
codesign -dvvv MyApp.app  # signing identity
```

22. Where experts disagree: whether Flutter's own-rendering approach is a strength or a liability. One camp values pixel-identical output and one codebase. The other argues that redrawing platform widgets means permanently chasing platform changes, especially in accessibility and text input. Both positions have shipped very large apps.

■ 35.14.6 WORDS - remember these

1. Native app — built with the phone maker's own tools — compiled against the platform SDK, using the platform's UI framework directly.
2. Cross-platform — one codebase, both phones — a shared runtime or compiler targeting iOS and Android from common source.
3. PWA — a website you can install — a manifest plus a service worker over HTTPS, installable to the home screen.
4. Service worker — a script that answers requests offline — a background worker intercepting fetches and managing a cache.
5. APK — the Android app file — a ZIP containing DEX bytecode, binary XML manifest, resources.arsc and signature blocks.
6. AAB — what you upload, not what installs — the Android App Bundle, from which Play generates per-device split APKs.
7. IPA — the iPhone app file — a ZIP containing a Payload folder holding one signed .app bundle with a Mach-O binary.
8. DEX — Android's compiled code format — Dalvik Executable, register-based bytecode ahead-of-time compiled by ART at install.
9. App review — a person deciding if your app may ship — guideline conformance checking applied to every submission and update.
10. Code signing — a seal proving nobody changed the app — certificate-backed hashes over every file, verified at install and launch.

35.15 Testing, performance and accessibility for the web

■ 35.15.1 PLAIN - in simple words

1. Three things separate a demo from a product: it keeps working when you change it, it is fast on a normal phone, and everyone can use it.
2. Tests are code that checks your other code. You run them automatically, every time, so a mistake is caught in seconds rather than by a customer.
3. There are three sizes of test. Small ones check one function. Medium ones check that two real parts fit together. Large ones drive a real browser like a real person.

4. Small tests are fast and cheap, so write many. Large tests are slow and fragile, so write few, and only for the paths that must never break.
5. Speed is not a feeling, it is a measurement, and Google publishes three specific numbers called the **Core Web Vitals**.
6. One measures how long until the main thing appears. One measures how long the page takes to answer a tap. One measures how much the page jumps around while loading.
7. **Lighthouse** is a free tool built into Chrome that scores a page and tells you what to fix.
8. **Accessibility** means the site works for people who cannot see it, cannot use a mouse, or need much larger text.
9. That is not a nice extra. It is a legal requirement in many places, and the basics are genuinely small: correct HTML tags, text for images, working keyboard navigation, readable contrast, a visible focus outline, and labels on inputs.

■ 35.15.2 PLAIN – a picture in your head

1. Think of building a bridge and checking it three ways.
2. You test each bolt in a jig before it goes in. That is a unit test: thousands of them, seconds each, and a failure tells you exactly which bolt.
3. You test one assembled span under load. That is an integration test: fewer, slower, and a failure tells you the span is wrong but not which bolt.
4. You drive a lorry across the finished bridge. That is an end-to-end test: a handful, minutes each, and a failure tells you only that something is wrong.
5. All three are worth doing. Only doing the lorry is how you end up rebuilding the bridge to find one bad bolt.

Where this comparison breaks:

6. A bridge bolt does not change while you are testing it. Software does. Tests also protect against your future self, which a bolt jig never has to do.
7. And a lorry crossing is reliable. Browser tests fail randomly for reasons that have nothing to do with your code, which is called flakiness, and a test suite nobody trusts is worse than no suite at all.

■ 35.15.3 PLAIN – a worked example

1. A single button component, tested at all three levels.

```
// unit: does the function do arithmetic
test("adds tax", () => {
  expect(withTax(100, 0.2)).toBe(120);
});

// integration: do component and state work together
test("shows total after adding", async () => {
  render(<Cart />);
  await user.click(screen.getByRole("button",
    { name: /add/i }));
  expect(screen.getByText("120")).toBeVisible();
});

// end-to-end: a real browser, a real server
test("checkout works", async ({ page }) => {
  await page.goto("/product/42");
  await page.getByRole("button", { name: "Buy" }).click();
});
```

```
await expect(page.getByText("Order placed")).toBeVisible();
});
```

2. Notice the queries use `getByRole` and the accessible name. A test written that way fails if the button stops being reachable by a screen reader, so the test suite quietly enforces accessibility.
3. Now the performance numbers. Here is one real page before and after work.

Metric	Before	After	Good if
LCP	4.8 s	1.9 s	2.5 s or less
INP	420 ms	140 ms	200 ms or less
CLS	0.31	0.02	0.1 or less
Lighthouse	38	94	90 or more

4. What actually produced those changes, in order of effect.
5. LCP: the hero image was 2.4 MB PNG. Converting to a 180 KB WebP, adding width and height, and marking it `fetchpriority="high"` removed most of the delay.
6. CLS: the same missing width and height were letting the text jump when the image finally arrived. Reserving the space fixed it. A web font swapping in late caused the rest, fixed with `font-display: optional`.
7. INP: a 300 ms sorting loop ran inside the click handler. Breaking it up and deferring the non-urgent part let the browser paint first.
8. The general lesson: two of the three fixes were one HTML attribute pair. Most real performance problems are not clever, they are neglected basics.

■ 35.15.4 PLAIN – what is really happening inside

1. A test runner imports your code, runs functions, compares results, and reports the ones that differ. There is no magic in it.
2. An end-to-end runner starts a real browser process and drives it through an automation protocol, then reads back the rendered page.
3. Test doubles replace slow or unreliable parts. A **mock** stands in for a thing and records how it was called. A **fake** is a working cheap version, such as an in-memory database.
4. The trap is mocking so much that the test only proves your mocks agree with each other. Prefer a real database in a container over a mocked one.
5. **LCP**, Largest Contentful Paint, is the time from navigation start until the biggest image or text block in the visible area finishes rendering. The browser keeps updating its guess and stops at the first user interaction.
6. **INP**, Interaction to Next Paint, watches every click, tap and key press for the whole visit, measures from the input to the next frame painted, and reports roughly the worst one.
7. **CLS**, Cumulative Layout Shift, adds up how much visible content moved without the user causing it. Each shift scores the fraction of the screen affected times the distance moved. It has no unit.
8. There are two kinds of measurement and they answer different questions. Lab data comes from one run on one machine with simulated conditions. Field data comes from millions of real visits on real phones.
9. Lighthouse gives lab data. It is repeatable, so it is good for finding causes and for stopping regressions in a build pipeline.
10. Core Web Vitals as Google actually uses them are field data, collected from real Chrome users who opted in, and judged at the 75th percentile.

11. That percentile choice matters: it means three quarters of your visits must be good. Your own fast laptop on office fibre tells you nothing.
12. Accessibility works through the **accessibility tree**, a second structure the browser builds beside the DOM, where each element has a role, a name, a state and a value.
13. A screen reader reads that tree, not your pixels. A `<button>` arrives with role “button” and is focusable and clickable by keyboard, all for free.
14. A `<div onclick=...>` styled to look like a button arrives with no role, no name, no keyboard focus and no Enter key handling. That is why the tag you choose is the single biggest accessibility decision you make.

■ 35.15.5 TECHNICAL – the engineer’s version

1. The testing pyramid was described by Mike Cohn in his 2009 book “Succeeding with Agile”. A common rule of thumb is roughly 70 percent unit, 20 percent integration, 10 percent end-to-end, though the exact split is convention, not a finding.
2. Kent C. Dodds’ “testing trophy” (2018) is the main counter-argument: it puts the most weight on integration tests, on the grounds that they catch the most real bugs per minute spent. Experts genuinely disagree here; both shapes ship good software.

Level	Tools	Typical time
Unit	Vitest, Jest, pytest, JUnit 5	1 to 50 ms
Integration	Testing Library, supertest	50 to 500 ms
Container-backed	Testcontainers, MSW	0.5 to 5 s
End-to-end	Playwright, Cypress, Selenium	5 to 60 s

3. Dates: Selenium began in 2004 with Jason Huggins at ThoughtWorks; WebDriver became a W3C Recommendation in June 2018. Cypress launched in 2017. Playwright came from Microsoft in 2020, written by former Puppeteer authors. Jest came from Facebook in 2014; Vitest arrived in 2021 alongside Vite.
4. Coverage percentages measure lines executed, not behaviour verified. A suite at 100 percent coverage with no assertions proves nothing. Use coverage to find untested files, never as a target to hit.
5. Core Web Vitals thresholds, current as of 2026. A page passes only if all three are in the good band at the 75th percentile, split by mobile and desktop.

Metric	Good	Needs work	Poor
LCP	2.5 s or less	2.5 to 4.0 s	over 4.0 s
INP	200 ms or less	200 to 500 ms	over 500 ms
CLS	0.1 or less	0.1 to 0.25	over 0.25

6. History: LCP and CLS became Core Web Vitals in May 2020. INP was announced as a replacement for First Input Delay in May 2023 and became a Core Web Vital on 12 March 2024. FID was retired because it measured only the delay before the first handler ran, not how long the page took to respond.
7. Field data comes from the Chrome User Experience Report, CrUX, which publishes monthly aggregates. PageSpeed Insights shows CrUX and Lighthouse side by side, and they routinely disagree, which confuses people. They are measuring different populations.

8. Lighthouse performance score weights, current version:

Metric	Weight
Total Blocking Time	30 percent
Largest Contentful Paint	25 percent
Cumulative Layout Shift	25 percent
First Contentful Paint	10 percent
Speed Index	10 percent

- Score bands are 0 to 49 poor, 50 to 89 needs improvement, 90 to 100 good. Lighthouse applies simulated mobile throttling by default, which is why a local score is lower than what your machine feels like.
- Note that INP is not in the Lighthouse score. Lighthouse uses Total Blocking Time as its lab proxy, because a single automated load performs no interactions. Fixing TBT usually fixes INP, but not always.
- Other measurement tools: WebPageTest for filmstrips and multi-location runs, the Chrome DevTools Performance panel for flame charts, and the web-vitals JavaScript library to report real user metrics to your own backend.
- Accessibility is governed by WCAG, the Web Content Accessibility Guidelines. WCAG 2.0 became a W3C Recommendation in December 2008, 2.1 in June 2018, and WCAG 2.2 on 5 October 2023. WCAG 3.0 remains a working draft.
- Conformance levels are A, AA and AAA. AA is the level almost all law and procurement references. AAA is not expected for whole sites.
- The genuinely non-negotiable basics, each tied to its success criterion:

Basic	Criterion	Concrete rule
Semantic HTML	1.3.1, 4.1.2	real button, nav, h1 to h6
Alt text	1.1.1	describe it, or empty alt
Keyboard	2.1.1, 2.1.2	everything reachable, no traps
Contrast	1.4.3, 1.4.11	4.5:1 text, 3:1 large and UI
Focus visible	2.4.7, 2.4.11	a clear ring, never obscured
Labels	3.3.2, 1.3.5	every input has a <label for>

- Contrast detail: 4.5 to 1 for normal text, 3 to 1 for large text, meaning at least 18 point or 14 point bold, which is about 24 CSS pixels or 18.66 pixels bold. AAA raises those to 7 to 1 and 4.5 to 1.
- New in WCAG 2.2 and often missed: 2.4.11 Focus Not Obscured, 2.5.8 Target Size Minimum at 24 by 24 CSS pixels, and 3.3.8 Accessible Authentication, which forbids requiring a cognitive test such as transcribing a code by hand with no paste allowed.
- The first rule of ARIA, from the W3C ARIA Authoring Practices Guide: do not use ARIA if a native HTML element with the needed semantics exists. Incorrect ARIA is measurably worse than none, because it overrides what the browser already knew.
- Never remove a focus outline without replacing it. outline: none with no substitute is the single most common accessibility regression in modern CSS.
- Automated tools find only part of the problem. Deque, the maker of axe, reports that automated testing catches roughly a third to a half of issues depending on the ruleset; the figure varies by study, so treat it as approximate. Nothing replaces a keyboard-only pass and a screen reader pass.

20. Law, briefly and with dates, because it drives budgets: Section 508 in the United States was refreshed in January 2017 to reference WCAG 2.0 AA. The European Accessibility Act's requirements applied from 28 June 2025. The United States Department of Justice ADA Title II rule of April 2024 requires WCAG 2.1 AA for state and local government sites, phasing in from 2026.
21. Tools: axe DevTools and @axe-core/playwright for automated checks, Lighthouse's accessibility category, WAVE, and the real screen readers: NVDA and JAWS on Windows, VoiceOver on macOS and iOS, TalkBack on Android.

■ 35.15.6 WORDS - remember these

1. Unit test — checks one small piece alone — an isolated assertion over a single function or module with no external dependencies.
2. Integration test — checks two real parts together — a test exercising collaborating components, often with a real database in a container.
3. End-to-end test — drives the app like a person — a full-stack test through a real browser via an automation protocol.
4. Flaky test — passes and fails without changes — a non-deterministic test, usually a timing or ordering race, that destroys trust in the suite.
5. LCP — how long until the main thing appears — Largest Contentful Paint, good at 2.5 seconds or less at the 75th percentile.
6. INP — how long the page takes to answer a tap — Interaction to Next Paint, good at 200 milliseconds or less.
7. CLS — how much the page jumps about — Cumulative Layout Shift, unitless, good at 0.1 or less.
8. Lab data — one measured run — a synthetic test under fixed throttled conditions, repeatable and good for regression gates.
9. Field data — what real users got — real user monitoring aggregated at a percentile, such as the Chrome User Experience Report.
10. Accessibility tree — what a screen reader reads — a parallel tree of roles, names, states and values derived from the DOM.
11. WCAG AA — the level the law usually means — the middle conformance level of the Web Content Accessibility Guidelines, currently 2.2.
12. Semantic HTML — using the tag that means the thing — elements chosen for meaning, giving roles and keyboard behaviour without extra code.

35.16 A realistic learning path

■ 35.16.1 PLAIN - in simple words

1. You have almost certainly seen a picture called a 180-day web development roadmap. It usually says something close to this.

Days	Topic
1 to 25	HTML and CSS
26 to 55	JavaScript
55 to 60	responsive design
61 to 75	React
76 to 105	Node and MongoDB
106 to 125	GitHub and APIs
126 to 180	cloud and projects

2. We are going to take that roadmap seriously, because it is not stupid. Parts of it are genuinely right, and it has helped a lot of people start.
3. What it gets right: the big order. Structure first, then styling, then the language, then a framework, then a server. You cannot begin at React.
4. What it also gets right: giving JavaScript more time than HTML and CSS. That ratio is correct, and most beginners get it backwards.
5. And it ends with projects and putting things online, which is exactly where a learning path should end.
6. Now the three things it gets wrong, said kindly, because each one costs beginners real months.
7. First, it teaches MongoDB before SQL, and in fact never teaches SQL at all. That is the wrong way round.
8. Second, it puts GitHub on day 106. Version control belongs on day one, before your first file, not after your first framework.
9. Third, and biggest: days are not the unit. “25 days of HTML” measures how long you sat there. It does not measure whether you can build a page.
10. The honest replacement is a list of stages, each defined by a thing you have built and can show, and each finished when the thing works.
11. Some people reach stage four in two months. Some take eight. Both are normal and neither number tells you anything about how good they will be.

■ 35.16.2 PLAIN – a picture in your head

1. Think about learning to cook rather than learning to watch cooking.
2. A day-count roadmap is a timetable that says “week three: sauces”. You attend week three. At the end of it you have watched sauces being made.
3. A stage-based path says “you are finished with sauces when you have served four different sauces to someone who ate them and told you the truth”.
4. The second one cannot be faked. There is no way to have served the sauce without having made the sauce.
5. It also self-corrects. If your sauce splits, you go back and learn about emulsions because you need to, and that lesson sticks permanently, in a way no scheduled lecture on emulsions ever does.
6. And notice what the timetable never mentions: cleaning as you go, knife safety, tasting. Those are not week seven. They are every single day.
7. Version control, testing and accessibility are the cleaning as you go.

Where this comparison breaks:

8. Cooking gives you an answer in twenty minutes. Software gives you a bug that takes two days, and during those two days it feels like you are learning nothing. You are learning the most.
9. Also, a chef can taste the dish. You cannot tell whether your code is good by looking at it. That is what other people’s review and real users are for.

■ 35.16.3 PLAIN – a worked example

1. Here is the replacement. Eight stages. Each one names what to learn, what to build, and the specific signal that you are finished with it.

Stage	Learn	Build	Done when
0 Tools	git, shell, editor	a repo of notes	you can undo a commit
1 Pages	HTML, CSS, layout	3 real pages	it works at 320px
2 Language	JavaScript, DOM	a to-do, no library	you explain the loop
3 Data	SQL, schema design	a 4-table schema	you write a join
4 Server	HTTP, one backend	a JSON API + login	passwords are hashed
5 Client	React or Svelte	rebuild stage 2	you know what it saved
6 Ship	deploy, CI, logs	put it online	a stranger used it
7 Depth	tests, ally, speed	improve stage 6	Lighthouse over 90

2. Stage 0, tools, roughly one week. Learn `git init`, `add`, `commit`, `push`, `branch`, `merge`, and how to read `git log`. Learn `cd`, `ls`, `grep`, and how to kill a stuck process.
3. Build: a repository where you keep your notes from this book, committed daily. The point is not the notes. The point is that committing becomes automatic before you have anything valuable to lose.
4. Done when: you can deliberately make a mess, and get out of it with `git restore`, `git reset` or `git revert`, without asking anyone.
5. Stage 1, pages, roughly three to five weeks. HTML semantics, the box model, flexbox, grid, and media queries. Responsive is not a separate topic; it is how you write CSS from the first line.
6. Build three pages that are not tutorials: a page about something you actually care about, a copy of a real site's layout, and a form-heavy page.
7. Done when: you resize the browser to 320 pixels wide and nothing overlaps, nothing overflows, and you did not need to look up flexbox.
8. Stage 2, the language, roughly six to eight weeks, and this is the stage people rush. Variables, types, functions, arrays, objects, closures, `this`, promises, `async/await`, modules, and the DOM API.
9. Build a to-do application with no framework at all. Then a small game. Then something that calls a public API and handles the error case properly.
10. Done when: you can predict the output order of a piece of asynchronous code on paper before running it, and you are right.
11. Stage 3, data, roughly three weeks, and this is the stage the popular roadmap skips entirely. Learn SQL before you learn any other database.
12. Build a schema for something real with four or five related tables: users, posts, comments, tags, and a join table. Load it with a few thousand rows.
13. Done when: you can write a query with two joins, a `GROUP BY` and a `HAVING` clause, and explain why you added an index.
14. Stage 4, the server, roughly four to six weeks. Pick one backend and stay there. Node with Express or Fastify, or Python with FastAPI or Django.
15. Build a JSON API with real registration and login, sessions or tokens done properly, input validation, and correct status codes.
16. Done when: passwords are stored with `bcrypt` or `Argon2id`, a wrong password gives 401, a valid user hitting somebody else's record gives 403, and you can say why those two are different.
17. Stage 5, the client framework, roughly four weeks. Now, and only now, learn React, Svelte or Vue.
18. Build: rebuild your stage 2 to-do app in the framework, against your stage 4 API. Same app, second time.

19. Done when: you can name three specific things the framework did for you that you had written by hand, and one thing it made harder.
20. Stage 6, shipping, roughly two weeks. A domain, HTTPS, environment variables, a build pipeline, a real deploy, and logs you can read at 2 a.m.
21. Done when: someone you have never met has used the thing and told you something about it. That single event teaches more than the previous month.
22. Stage 7, depth, ongoing forever. Tests, accessibility, performance, security and reading other people's code.
23. Done when: nothing. This stage does not end, and that is the job.

■ 35.16.4 PLAIN – what is really happening inside

1. Why SQL before MongoDB, concretely. A relational schema forces you to decide, on day one, what your data actually is and how the pieces relate.
2. That decision is the hard part of software, and doing it early with a database that argues back is how you learn it.
3. A document database lets you postpone every one of those decisions. That feels faster and it genuinely is faster, for about six weeks.
4. Then you need to know which users commented on which posts, and you discover you now have to write the join yourself, in application code, without transactions, at three in the morning.
5. There is nothing wrong with MongoDB. It is a good database for documents that really are documents. It is a bad first database because it does not teach modelling, and modelling is the transferable skill.
6. Also, in almost every job you will apply for, the database is PostgreSQL, MySQL or SQL Server. SQL is asked about in interviews. Mongo query syntax almost never is.
7. Why git on day one. Every hour you spend without version control is an hour where the worst outcome is losing everything and the second worst is being afraid to try things.
8. Version control is not an advanced topic. It is the thing that makes learning safe, because you can break anything and get back.
9. It is also, bluntly, the one skill every single job requires. Learning it on day 106 means 105 days of work you cannot show anyone.
10. And be precise about the words: **git** is the version control program, made by Linus Torvalds in 2005. **GitHub** is one website that hosts git repositories, founded in 2008 and bought by Microsoft in 2018. Learn git. GitHub takes an afternoon once you know git.
11. Why days are the wrong unit. Two people both “do 25 days of HTML”. One watches videos for two hours a day. One builds and breaks pages for four. They are not in the same place, and the calendar cannot tell them apart.
12. Worse, a day count creates a false finish line. Day 26 arrives, you move to JavaScript because the plan says so, and you carry a shaky understanding of the box model into everything that follows.
13. A build target has no such problem. The page either survives 320 pixels or it does not. You cannot argue with it and you cannot schedule past it.
14. What the popular roadmap leaves out entirely, and should not: testing, accessibility, security, debugging, and reading code you did not write.
15. Those are not advanced. Debugging in particular is most of the job, and no roadmap ever allocates a single day to it.
16. One more honest thing. You will feel, somewhere around stage 2, that everyone else understands this and you do not. That feeling is not evidence. It is what learning something genuinely hard feels like from the inside, and it arrives for everyone, including the people you are comparing

yourself to.

■ 35.16.5 TECHNICAL – the engineer’s version

1. Realistic hour costs, from teaching experience rather than from a course listing. Treat these as approximate and personal.

Stage	Focused hours	At 15 h/week
0 Tools	10 to 20	about 1 week
1 Pages	60 to 100	4 to 7 weeks
2 Language	120 to 200	8 to 13 weeks
3 Data	40 to 70	3 to 5 weeks
4 Server	80 to 120	5 to 8 weeks
5 Client	50 to 80	3 to 5 weeks
6 Ship	20 to 40	1 to 3 weeks

2. That totals roughly 380 to 630 focused hours to be employable at a junior level, which at a genuine 15 hours a week is about 26 to 42 weeks.
3. “180 days” at 3 hours a day is 540 hours, so the popular roadmap’s total is not unreasonable. Its problem is the distribution and the ordering, not the size.
4. The distribution the roadmap gets wrong, in numbers: it gives 25 days to HTML and CSS and 15 days to React, while giving zero to SQL, zero to testing and zero to debugging.
5. A portfolio that actually gets interviews contains three items, not fifteen:

Project	What it must prove
A full-stack app	auth, a real schema, deploy
A focused tool	you can finish something small
A contribution	you can read others’ code

6. Concrete quality bar for the full-stack app: a live URL, a README with setup steps that work on a clean machine, at least a handful of tests, migrations rather than a hand-edited database, and no secrets in the repository.
7. Reading the git history of a candidate’s project tells an interviewer more than the project does. Many small commits with real messages beat one commit called “initial commit” containing 40,000 lines.
8. Order of technologies, with the reasoning stated as a rule: learn the thing that is underneath before the thing that abstracts it, unless the underneath thing is genuinely obsolete.
9. So: HTML and CSS before Tailwind. JavaScript before React. SQL before an ORM. HTTP before a framework’s router. Git before any hosting website.
10. The exception is real. Nobody should learn assembly before Python, or Objective-C before Swift. The rule applies one layer down, not all the way.
11. TypeScript timing: learn it after stage 2, not during. Types over a language you cannot yet read are noise. Once you can read JavaScript, TypeScript takes about a week and pays for itself immediately.
12. On tutorials, stated plainly: following a tutorial produces the feeling of competence without the substance, because every decision was already made for you. The cure is to build the same thing again, from an empty folder, with the tutorial closed.
13. On AI assistants, separating fact from claim. Established fact: they generate working code for common tasks quickly and are excellent at explaining unfamiliar code. Active research: how much

they change long-term skill acquisition in beginners; the studies are early and mixed. Marketing claim: that they remove the need to understand the fundamentals. If you cannot read the answer, you cannot tell when it is wrong, and it is confidently wrong often enough to matter.

14. A defensible use during learning: write it yourself first, then ask for a critique. That order keeps the skill and gains the speed.
15. What does not decide your career: which framework you chose. React, Vue, Svelte and Angular all teach the same underlying ideas of state, components and reconciliation, and moving between them takes a fortnight.
16. What does decide it: whether you can take a vague problem, model the data, build something that works, and explain the trade-offs you made.

■ 35.16.6 WORDS – remember these

1. Roadmap — a picture of what to learn next — an ordered curriculum, useful as a sequence and misleading as a schedule.
2. Stage gate — a built thing that proves readiness — an objective completion criterion replacing elapsed time as the unit of progress.
3. Tutorial trap — following along and learning nothing — passive completion without independent recall or decision-making.
4. Portfolio — the few things you can show — a small set of deployed, documented, version-controlled projects with real commit history.
5. git — the version control program — a distributed content-addressable version control system, Linus Torvalds, 2005.
6. GitHub — one website that stores git projects — a hosting and collaboration platform for git repositories, founded 2008.
7. Full stack — you can do both halves — competence across the browser client, the server application and its datastore.
8. Junior-ready — good enough to be paid to learn — able to ship a reviewed feature end to end with guidance, not to work unsupervised.

35.98 Common wrong ideas

1. Wrong: HTML is a programming language and knowing it means you can program. Right: HTML is a markup language. It describes structure and has no variables, no conditions and no loops. JavaScript is the only one of the three web languages that can compute anything.
2. Wrong: `!important` fixes a specificity problem. Right: it hides one. It wins by leaving the normal cascade entirely, so the next person needs a second `!important` to override it, and after four rounds nobody can predict which rule applies. The real fix is a lower-specificity selector.
3. Wrong: `async` and `await` make JavaScript run on several threads. Right: your JavaScript still runs on exactly one thread. `await` yields the thread while waiting, so other queued work can run. Parallelism needs Web Workers.
4. Wrong: the virtual DOM is faster than touching the DOM. Right: a hand-written direct DOM update is always faster than a diff plus the same update. The virtual DOM buys predictable performance from declarative code, not raw speed, and its own authors have said so.
5. Wrong: you need a framework to build a website. Right: a framework earns its place when many pieces of shared state must stay in sync. A content site, a form, a landing page and a blog are all better plain, and ship faster.
6. Wrong: MongoDB should be your first database because it scales. Right: PostgreSQL handles the workload of almost every application ever written, has transactions and joins, and has a `jsonb`

column type when you genuinely need flexible documents. Learn relational modelling first; it transfers everywhere.

7. Wrong: hashing a password with SHA-256 is secure because SHA-256 is a strong hash. Right: SHA-256 is strong and fast, and fast is exactly the wrong property. A GPU tries billions of SHA-256 guesses a second. Use bcrypt, scrypt or Argon2id with a per-user salt, tuned to take about 100 milliseconds.
8. Wrong: JSON Web Tokens are more secure than sessions. Right: they are a different trade, not a better one. They remove the server lookup and, with it, the ability to revoke. Sessions in an `HttpOnly` cookie are the safer default, and a JWT in `localStorage` is stolen by the first cross-site scripting bug.
9. Wrong: microservices scale better, so start with them. Right: they let teams deploy independently, which is an organizational benefit with a technical cost. Every in-process call becomes a network call that can fail, and one ACID transaction becomes a saga. Start with a modular monolith and split when a specific team or scaling boundary forces it.
10. Wrong: accessibility is a polish step near launch, and mostly means adding ARIA attributes. Right: it is decided by the tags you choose on the first day. A real `<button>` arrives with a role, keyboard focus and Enter handling; a styled `<div>` arrives with none. The W3C's own first rule of ARIA is not to use ARIA when a native element already does the job.

35.99 Chapter summary in 20 lines

1. A web application runs in two places: a frontend on the user's device that you do not control, and a backend on a server that you do. Never trust the first.
2. HTML gives structure, CSS gives presentation, JavaScript gives behaviour, and only JavaScript is a programming language.
3. Semantic HTML tags are not decoration. They set the role, keyboard behaviour and screen-reader name of everything on the page, for free.
4. CSS resolves conflicts by origin, then specificity counted as inline, id, class and element, then source order. The box model, flexbox and grid replace guessing with rules.
5. JavaScript has one thread, a call stack, a task queue and a microtask queue. Microtasks drain completely before the next task, which is why promise callbacks beat `setTimeout(0)` every time.
6. The browser parses HTML into the DOM and CSS into the CSSOM, combines them into a render tree, then runs layout, paint and composite. Changing geometry forces a reflow; changing transform and opacity does not.
7. The browser is a platform in its own right: storage, workers, fetch, canvas, WebGL and WebAssembly, all inside a same-origin sandbox that CORS relaxes deliberately and explicitly.
8. A frontend framework's real product is keeping the screen in agreement with the state. Virtual DOM diffing, fine-grained signals and compilation are three answers to that one problem.
9. Build tooling exists because browsers want a few small optimized files while humans want many readable ones: bundling, transpiling, minifying, tree shaking and content-hashed names for caching.
10. A backend program listens on a port, parses HTTP, routes the request through middleware to a handler, talks to a database, and serializes a reply. Every framework in every language is that same loop.
11. Relational databases store facts once, in typed tables, joined on keys, with B+ tree indexes turning scans into lookups and ACID transactions making partial failure impossible.
12. Choose a datastore by workload, not by fashion. PostgreSQL is the correct default; Redis, Elasticsearch, Cassandra and Neo4j each answer a specific question that PostgreSQL answers less well.

13. An ORM removes repetitive code and adds the N+1 query trap. Use it for the ordinary ninety percent, read the SQL it generates, and write SQL yourself when the query is the point.
14. Store passwords with a deliberately slow, salted hash: Argon2id or bcrypt, never a bare fast hash, and never the password itself.
15. Sessions keep state on the server and can be revoked instantly. Tokens keep state with the client and scale without a lookup. Pick the trade knowingly, and keep credentials in HttpOnly cookies.
16. OAuth 2.0 delegates authorization through an authorization code exchanged on a back channel with PKCE; OpenID Connect adds the ID token that turns it into a login system. Authentication is who you are; authorization is what you may do, and RBAC keeps the second one auditable.
17. Architecture is caching at the right layer, moving slow work to a queue, choosing polling, server-sent events or WebSockets on merit, rate limiting every public endpoint, and starting with a monolith you could split later.
18. Mobile means choosing between native Swift or Kotlin, cross-platform React Native or Flutter, and an installable web app, then living with a store review on every release. An APK and an IPA are both ZIP archives, which is Chapter 20's lesson that the bytes, not the extension, define a file.
19. Quality is measurable: a pyramid of fast unit tests under fewer browser tests, LCP at 2.5 seconds, INP at 200 milliseconds, CLS at 0.1 measured at the 75th percentile of real users, and WCAG 2.2 level AA as the floor rather than the ambition.
20. Learn in stages defined by what you have built, not in days spent: tools, then pages, then the language, then SQL, then a server, then a framework, then shipping, then depth forever.

APIs — What They Are and Where They Came From

36.0 What this chapter gives you

1. You will be able to say what an API actually is in one sentence, and to defend that sentence: it is a published agreement, not a technology.
2. You will be able to point at an API at five different scales, from one function inside one file up to a service on the other side of the planet.
3. You will be able to tell the real history, with names and years: subroutine libraries in 1951, the term itself in 1968, remote procedure calls in 1984, CORBA in 1991, SOAP and REST around 2000, GraphQL and gRPC after 2015.
4. You will be able to explain the difference between an API and an ABI, and say why a program can compile fine and still crash at run time.
5. You will be able to read semantic versioning and say exactly what a change in each of the three numbers promises you.
6. You will be able to list the eight fallacies of distributed computing and say what each one costs you when you believe it.
7. You will be able to write out a raw HTTP request and response by hand, name every part, and pick the right status code for a situation.
8. You will be able to state what Roy Fielding actually specified in 2000, and why almost nothing sold as REST today meets that specification.
9. You will be able to compare REST, GraphQL, gRPC, SOAP, JSON-RPC, WebSockets and server-sent events on payload size, typing, streaming and tooling.
10. You will be able to walk through the OAuth 2.0 authorization code flow with PKCE step by step, decode a JWT field by field, and say what a scope does.
11. You will be able to make a client that does not fall over: retries with backoff and jitter, idempotency keys, timeouts and circuit breakers.
12. You will be able to design an API other people can use, document it with OpenAPI, and explain why some companies live or die by that decision.

36.1 What an API actually is

■ 36.1.1 PLAIN – in simple words

1. Software is made of parts. Parts need to ask each other to do things.
2. An **API** is the published agreement about how one part may ask another part to do something.
3. The letters stand for **application programming interface**. That name is not very helpful, so ignore the letters and remember the idea: an agreement.
4. The agreement says three things. What you may ask for. How you must ask. What you will get back.
5. It also says, quietly, one more thing: everything not written down is not promised, and may change without warning.
6. An API is not a program. It is the shape of the doorway into a program.
7. The most important word in this chapter is **contract**. An API is a contract between the person who wrote a piece of software and the person who wants to use it.
8. Because it is a contract and not a technology, an API can be made of almost anything. Text over a wire. A name in a file. A number in a register.
9. So the sentence “we built an API” tells you nothing about the technology. It only tells you that somebody wrote down a promise.

■ 36.1.2 PLAIN – a picture in your head

1. Think of a restaurant kitchen. You are hungry. The kitchen can cook.
2. You do not walk into the kitchen. You are handed a menu.
3. The menu is the API. It lists exactly what you may ask for. It gives the name of each dish and what comes with it.
4. You do not need to know whether the soup is made from a family recipe or from a packet. You do not need to know who the cook is today.
5. The kitchen can throw out its old stove and buy a new one. Your order form does not change. That is the whole benefit in one line.
6. And if the menu says “soup” but the kitchen sends a sandwich, the kitchen has broken the contract, even if the sandwich is delicious.

Where this comparison breaks:

7. A menu is a suggestion. An API is checked by a machine, and a machine has no good manners. A single wrong letter in the order and nothing is served.
8. A restaurant serves one customer at a time at one table. A busy API may be serving fifty thousand callers in the same second, and they can interfere with each other in ways that diners cannot.
9. A menu never fails halfway. An API call over a network can be taken, cooked, and then lost on the way back, which is a case with no restaurant equivalent. Section 36.5 deals with exactly that case.

■ 36.1.3 PLAIN – a worked example

1. Here is the same idea, an API, written at five completely different scales. All five are APIs. None of them is more of an API than the others.

Scale	The published agreement
One function	<code>int strlen(const char *s)</code>
One library	<code>curl_easy_setopt(h, opt, v)</code>
The kernel	<code>write(fd, buf, count)</code>
A GPU driver	<code>vkQueueSubmit(queue, n, ...)</code>
The internet	<code>GET /repos/{owner}/{repo}</code>

Scale	The published agreement

2. Scale one is inside one program. The function `strlen` promises: give me the address of a run of text ending in a zero byte, and I return its length.
3. Scale two is a library on your machine. The C library `libcurl` promises: set this option to this value on this handle, and I return zero if I accepted it.
4. Scale three crosses the boundary into the operating system. The `write` call promises: I will try to write that many bytes, and tell you how many I wrote.
5. Scale four crosses into a device driver and then into hardware. Vulkan promises: I will hand this batch of work to the graphics card's queue.
6. Scale five crosses the planet. The GitHub HTTP API promises: ask me for that path with a valid token and I will return a JSON description of that repository.
7. Notice what changed across the five, and what did not. The technology changed completely. The idea did not change at all.
8. What did change is how much can go wrong. At scale one, essentially nothing. At scale five, everything. That is section 36.5.

■ 36.1.4 PLAIN – what is really happening inside

1. An API has three parts, and all three matter.
2. The **surface** is the list of things you can call, and their names.
3. The **shape** of each call is the data you must supply and the data you get back: how many items, of what kind, in what order, in what units.
4. The **behaviour** is what the call actually does, including how it fails, how long it may take, and whether calling it twice is different from once.
5. Beginners think an API is only the surface. Most real bugs live in the behaviour.
6. The surface can be checked by a machine. A compiler will refuse to build code that calls `strlen` with two arguments.
7. The shape can be partly checked by a machine. Types catch some mistakes.
8. The behaviour usually cannot be checked at all. Nothing stops a function documented as “returns quickly” from taking nine seconds.
9. So a good API is a small surface, a strict shape, and behaviour written down in plain sentences that a person actually reads.

The honest version: the contract also includes things nobody wrote down.

10. If your code depends on a list coming back in alphabetical order, and the documentation never promised an order, you have invented a clause.
11. When the other side changes and your code breaks, both of you will believe you were right. This is the single most common source of API arguments.
12. The rule that resolves it: if it is not in the document, it is not promised, no matter how long it has been true in practice.

■ 36.1.5 TECHNICAL – the engineer's version

1. Formally, an API is an **interface specification**: a set of named operations, each with a signature, a set of pre-conditions and post-conditions, and an error model.
2. Two systems that share an interface specification are said to be **interoperable** at that interface. Neither needs the other's source.

3. The specification may be enforced statically (the compiler rejects bad calls), dynamically (the runtime throws), or not at all (documentation only).
4. The formal notion behind this is **design by contract**, named by Bertrand Meyer for the Eiffel language in the 1980s: pre-conditions, post-conditions and invariants stated as part of the interface.
5. An API is distinct from a **protocol**. A protocol specifies bytes on a wire and the ordering rules between peers. An API specifies operations available to a caller. HTTP is a protocol; the GitHub REST API is an API that uses it.
6. An API is distinct from an **ABI** (application binary interface), which fixes register use, calling convention, structure layout and symbol naming. Section 36.3 measures the difference.
7. Real interface surfaces, for scale. These are order-of-magnitude figures from published documentation and headers, not exact counts, because every one of them depends on version and configuration.

Interface	Rough size of surface
C standard library	about 200 functions
Linux system calls	about 350 on x86-64
Win32 core DLLs	thousands of functions
POSIX.1-2024	over 1,200 interfaces
GitHub REST API	over 1,000 endpoints

8. Tools that let you observe an interface directly: `nm -D libc.so.6` lists exported symbols, `objdump -T` shows the dynamic symbol table, `man 2 syscalls` lists Linux system calls, `strace` records them as they happen, and `curl -v` prints an HTTP exchange in full.

■ 36.1.6 WORDS – remember these

1. API — a published agreement about how to ask software to do something — an interface specification of operations, signatures and error behaviour.
2. Contract — a promise both sides rely on — the pre-conditions, post-conditions and invariants attached to an interface.
3. Interface — the doorway into a component — the named boundary across which two components interact without sharing internals.
4. Protocol — the rules for talking on a wire — the byte format and message ordering between peers, independent of any one caller's API.
5. Implementation detail — how it happens to work today — behaviour observable but not specified, therefore free to change without notice.
6. Interoperable — two things can work together — able to interact correctly through a shared interface specification without shared source code.

36.2 The history, properly

■ 36.2.1 PLAIN – in simple words

1. The reader asked how APIs were invented. The honest answer is that nobody invented them in one moment. They were arrived at, in stages, over 75 years.
2. Stage one, around 1951. People noticed they were writing the same little piece of code again and again, so they saved it and reused it.
3. That reusable piece was called a **subroutine**. To reuse it you needed to know what to give it and what it gave back. That written-down knowledge was the first API, before anybody had a name for it.

4. Stage two, the 1960s. Computers grew operating systems. A program could no longer touch the disk directly. It had to ask the operating system.
5. That asking needed rules, so operating systems published their own lists of calls. The name **application program interface** appears in print in 1968.
6. Stage three, the 1980s. Machines were joined by networks. Somebody asked: why can we not call a piece of code on another machine the same way we call one on this machine?
7. That idea is the **remote procedure call**. It was tried properly in 1984, and it worked, and it also created every hard problem in the rest of this chapter.
8. Stage four, the 1990s. Big vendors built big frameworks to make remote calls easy across languages. They were called CORBA and DCOM. They mostly failed.
9. Stage five, from 1998 to 2002. People gave up on frameworks and used the web instead: plain text over HTTP, through the same port a browser uses.
10. Stage six, the 2000s. Companies started publishing these web interfaces to the outside world, so that strangers could build on top of their business.
11. Stage seven, since 2015. New shapes appeared for particular problems: GraphQL for clients that want to choose their own data, gRPC for fast machine-to-machine traffic inside a company.

■ 36.2.2 PLAIN – a picture in your head

1. Think of a workshop that makes wooden chairs.
2. At first one person makes the whole chair. Everything is in their head.
3. Then somebody makes a jig, a wooden guide for cutting a leg to the right length. Now anyone can cut a leg. That is a subroutine.
4. Then a drawer of jigs appears, with labels. That is a library.
5. Then the workshop gets so big that the wood store is locked, and only a storekeeper may enter. You fill in a slip to get wood. That is a system call.
6. Then a second workshop opens across town. You want to send slips there too. Someone invents a courier. That is a remote procedure call.
7. And the moment the courier exists, new problems exist: the van breaks down, the slip arrives twice, the other workshop is closed.

Where this comparison breaks:

8. Jigs do not have versions, and a jig never returns a different result on Tuesday. Software interfaces change under you, which is section 36.3.
9. A courier who fails tells you he failed. A network can swallow a message and tell nobody, which is the hardest single fact in this chapter.

■ 36.2.3 PLAIN – a worked example

1. Take one task and watch it move across the eras: “give me the photographs belonging to user 42”.
2. In 1951, the task does not exist, but the shape does. You would look up a subroutine in a paper catalogue, copy its tape into your program, and know from the catalogue that it wanted a number in one register.
3. In 1975, on a Unix machine, you would open a file, read it, and close it. Three system calls, all local, all fast, all likely to succeed.
4. In 1988, on a network, you would call a function named `getPhotos(42)` that secretly sent a message to another machine. It looked local. It was not.
5. In 1995, with CORBA, you would write an interface file, run a compiler on it, and get a stub object in your language that hid the network again.

6. In 1999, with SOAP, you would send a page of XML describing a method call over HTTP, and get back a page of XML describing the answer.
7. In 2005, with a web API, you would ask for a URL and get a small block of JSON. Here is the whole thing:

```
GET /users/42/photos HTTP/1.1
Host: api.example.com

HTTP/1.1 200 OK
Content-Type: application/json

{"photos":[{"id":9,"title":"Pune, monsoon"}]}
```

8. In 2016, with GraphQL, you would send one query naming exactly the fields you wanted, and get back exactly those fields and nothing else.
9. Look at step 6 against step 7. Same task. The XML version was often two to five kilobytes; the JSON version above is under 60 bytes of payload. That size difference is a large part of why the industry moved.

■ 36.2.4 PLAIN – what is really happening inside

1. The history is not random. Each step is a response to a specific pain.
2. Subroutines answered “I am typing the same thing repeatedly”.
3. Subroutine libraries answered “I cannot find the piece I wrote last year”.
4. Operating system calls answered “two programs both wrote to the disk and destroyed each other’s data”.
5. Shared libraries answered “forty programs each carry their own copy of the same code and the machine has 4 MB of memory”.
6. Remote procedure calls answered “the data I need is on another machine”.
7. CORBA and DCOM answered “my two programs are in different languages”.
8. XML over HTTP answered “corporate firewalls block everything except port 80, and I cannot get the network team to open anything”.
9. REST-style web APIs answered “SOAP is enormous and my phone has a slow connection and a small battery”.
10. GraphQL answered “my mobile screen needs six fields and the endpoint sends ninety”.
11. gRPC answered “we make two million internal calls per second and JSON parsing is now a measurable part of our electricity bill”.
12. Read that list backwards and you can predict the next step: whatever hurts most at scale today is what gets replaced next.

■ 36.2.5 TECHNICAL – the engineer’s version

1. The timeline, with checkable dates.

Year	Event	Who
1951	Subroutine library book	Wilkes, Wheeler, Gill
1968	Term “application program interface”	Cotton, Grestorex
1974	API applied to databases	C. J. Date
1981	RPC named in a PhD thesis	Bruce Jay Nelson
1984	Working RPC published	Birrell and Nelson
1988	POSIX 1003.1 published	IEEE
1991	CORBA 1.0, October	OMG

Year	Event	Who
1993	Win32 with Windows NT 3.1	Microsoft
1996	DCOM ships	Microsoft
1998	XML-RPC published	Dave Winer, UserLand
2000	SOAP 1.1 W3C Note, 8 May	W3C submission
2000	REST defined in a thesis	Roy Fielding
2000	First commercial web API, 7 Feb	Salesforce
2000	Web API, 20 November	eBay
2002	Amazon Web Services, 16 July	Amazon
2004	Photo API	Flickr
2006	Platform API, August	Facebook
2006	Public API, September	Twitter
2012	OAuth 2.0, RFC 6749, October	IETF
2015	GraphQL specification opened	Facebook
2016	gRPC 1.0, 23 August	Google

2. **1951.** Maurice Wilkes, David Wheeler and Stanley Gill published *The Preparation of Programs for an Electronic Digital Computer* at Cambridge, generally called the first book on programming. It grew out of a September 1950 internal report on the EDSAC subroutine library. The subroutines lived on punched paper tape in a filing cabinet, and the catalogue describing how to call each one is, in every meaningful sense, the first API document.
3. **1968.** The term “application program interface”, without the “-ing”, appears in the paper *Data structures and techniques for remote computer graphics* by Ira W. Cotton and Frank S. Grestorex, presented at the AFIPS Fall Joint Computer Conference held 9 to 11 December 1968. It described the boundary that gave a graphics program independence from the hardware. This is the commonly cited first published use. It is a first citation, not a moment of invention: the practice was already twenty years old.
4. **1974.** C. J. Date’s paper *The Relational and Network Approaches: Comparison of the Application Programming Interface* carried the idea into databases, and the “-ing” form became standard.
5. **1981 and 1984.** Bruce Jay Nelson’s doctoral work at Carnegie Mellon named the remote procedure call. Andrew Birrell and Nelson then built one at Xerox PARC, reported in Xerox technical report CSL-83-7 in December 1983 and published as *Implementing Remote Procedure Calls* in ACM Transactions on Computer Systems, volume 2, number 1, February 1984. It ran on the Cedar system over Ethernet, and reported a null local call at a few microseconds against a null remote call at about 1.1 milliseconds. That ratio is the whole story of distributed systems in one measurement.
6. **1988.** IEEE Std 1003.1-1988 standardized the Unix system call interface. Richard Stallman suggested the name POSIX in place of IEEE-IX.
7. **1991 to 1997.** The Object Management Group, founded 1989, published CORBA 1.0 in October 1991 and CORBA 2.0 with the IIOP wire protocol in 1997. Microsoft shipped DCOM in 1996. Both aimed at language-independent remote objects.
8. Why they lost, in the analysis of Michi Henning’s 2006 ACM Queue article *The Rise and Fall of CORBA*: specifications too large and partly never implemented; a committee process driven by vendor politics; no answer to firewalls; and, for DCOM, Windows only. Experts still disagree on the weighting. Henning stresses process failure; Douglas Schmidt’s published reply stresses that the technology worked and the market moved.
9. There is a second reason, and it is the deeper one. Both tried to make a remote call look exactly like a local call. Section 36.5 explains why that is a lie that cannot be sustained.

10. **1998 to 2000.** Dave Winer of UserLand shipped XML-RPC in 1998, subset from an unshipped 1998 SOAP draft. SOAP 1.1 was published as a W3C Note on 8 May 2000, and SOAP 1.2 became a W3C Recommendation on 24 June 2003.
11. **2000.** Roy Thomas Fielding's doctoral dissertation *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, 2000, supervised by Professor Richard N. Taylor, defined REST in Chapter 5. Fielding was a co-author of the HTTP specifications, so this was a description of the web's actual architecture, derived, not invented. Section 36.7 covers what it says.
12. **The web API era.** Salesforce demonstrated an XML API at the IDG Demo conference on 7 February 2000, widely called the first commercial web API. eBay launched its API and developer programme on 20 November 2000. Amazon launched Amazon Web Services on 16 July 2002. Flickr's API followed its February 2004 launch. Facebook's platform arrived in August 2006 and Twitter's API in September 2006.
13. **2015 to 2016.** GraphQL was built inside Facebook in 2012 for its mobile apps, and the specification and reference implementation were opened during 2015, announced publicly on 14 September 2015. Google built gRPC from its internal Stubby system, announced it in 2015, and released version 1.0 on 23 August 2016.

■ 36.2.6 WORDS – remember these

1. Subroutine — a saved, reusable piece of code — a callable block with a defined entry point, parameters and return convention.
2. Subroutine library — a labelled drawer of saved pieces — an indexed collection of reusable routines with published calling documentation.
3. RPC — calling code on another machine like a local function — remote procedure call, with client stub, marshalling, transport and server skeleton.
4. Marshalling — packing arguments for travel — serializing typed values into a byte stream and reconstructing them at the far end.
5. CORBA — a 1990s cross-language remote object standard — Common Object Request Broker Architecture, OMG, CORBA 1.0 in October 1991.
6. IDL — a file describing an interface once, for many languages — interface definition language, compiled into stubs and skeletons.
7. SOAP — a heavy XML message format for method calls — W3C Note in May 2000, Recommendation 1.2 in June 2003, usually carried over HTTP.

36.3 Local APIs: signatures, libraries and versions

■ 36.3.1 PLAIN – in simple words

1. The simplest API is a function inside a program you are writing.
2. Its **signature** is its name, the things it takes, and the thing it gives back. That single line is the whole contract for a lot of code.
3. When you package many functions together for other people to use, you have a **library**.
4. To use a library in C or C++ you need a **header file**: a file that lists the signatures without the code, so the compiler knows what is allowed.
5. In other languages the same job is done by import statements, package manifests or type definition files. The idea is identical.
6. Once other people depend on your library, you are stuck with your own promises. Changing them breaks their programs.

7. So library authors need a way to say “this change is safe” or “this change will break you”. That way is **versioning**.
8. A change that breaks existing callers is called a **breaking change**. The whole discipline of versioning exists to warn about exactly that.

■ 36.3.2 PLAIN – a picture in your head

1. Think of a plug and a socket in a wall.
2. The API is the shape of the plug: three pins, in that arrangement, that size.
3. Any lamp with that plug shape works. The wall does not care what the lamp is.
4. Now imagine the electricity company changes the voltage from 230 to 400 while keeping the socket shape identical.
5. Your plug still fits. Your lamp explodes.
6. That is the difference between the shape of the interface and its behaviour, and it is exactly how most painful breakages feel.

Where this comparison breaks:

7. Sockets are regulated by law and change once a century. Libraries change weekly, and any one program may depend on two hundred of them.
8. A plug either fits or does not. A software interface can half-fit: it compiles, links, runs for six weeks, and then fails on one unusual input.

■ 36.3.3 PLAIN – a worked example

1. Here is a C function signature and every promise inside it.

```
| size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```
2. `fread` is the name. That is the only part most people remember.
3. `void *ptr` is where to put the data. `void *` means “any kind of thing”, so the compiler will not check what you point at. That is a hole.
4. `size_t size` is how big one item is, in bytes. `size_t n` is how many.
5. `FILE *stream` is the open file to read from.
6. `size_t` on the left is the return: the number of complete items actually read, which may be fewer than `n`. Not bytes. Items.
7. That last point is a real bug factory. The type says nothing about it. Only the documentation does.
8. Now the versioning question. Suppose a library goes from 2.4.1 to 3.0.0. Here is exactly what each position promises, under semantic versioning.

Change	Example	What it promises
Patch	2.4.1 to 2.4.2	Bug fix only, safe
Minor	2.4.1 to 2.5.0	New features, still safe
Major	2.4.1 to 3.0.0	Something broke, read notes

9. Real cases. Python 2 to Python 3, released December 2008, changed `print` from a statement to a function and forced a decade-long migration. That is a major change and the number said so.
10. Adding a new optional argument with a default value is normally a minor change: old callers keep working.
11. Removing a function, renaming it, changing an argument’s type, or changing what a return value means are all major changes, even when the code still compiles.

■ 36.3.4 PLAIN – what is really happening inside

1. There are two different boundaries and people confuse them constantly.
2. The **API** is what the compiler checks: names, argument counts, types.
3. The **ABI** is what the machine checks at run time: which register holds which argument, how big a structure is, where each field sits inside it, what the function symbol is called in the compiled file.
4. You can keep the API identical and break the ABI. Add one field to the middle of a structure and every already-compiled program that uses that structure now reads the wrong bytes.
5. Nothing warns you. The program starts. Then it produces nonsense, or dies.
6. That is why shared libraries carry a separate number in the file name, like `libssl.so.3`. That number is the ABI version, not the product version.
7. Rebuilding from source fixes an ABI break. It cannot fix an API break, because the source itself no longer matches.
8. So the two rules are: change the API and callers must edit their code; change the ABI and callers must at least rebuild.

■ 36.3.5 TECHNICAL – the engineer’s version

1. Semantic Versioning, written by Tom Preston-Werner, co-founder of GitHub, published as version 1.0.0-beta in 2010 and version 2.0.0 in 2013, states the rule as MAJOR.MINOR.PATCH.
2. It is a **convention**, not a standard body’s specification. Package managers such as npm, Cargo and Go modules enforce it mechanically, which makes it feel like a law. Many projects publish semver numbers and break them anyway.
3. The exact promises from the specification: MAJOR when you make incompatible API changes, MINOR when you add functionality in a backward-compatible way, PATCH when you make backward-compatible bug fixes.
4. Version 0.y.z means anything may change at any time. That clause is heavily used and heavily abused.
5. Go modules make major versions part of the import path from v2 onward, so `example.com/pkg/v2` and `example.com/pkg` can coexist in one build. That is a deliberate answer to the diamond dependency problem.
6. ABI stability in practice, with real numbers:

Interface	ABI policy
Linux kernel to userspace	Stable, never break it
Linux internal module ABI	No stability promise
glibc	Stable, symbol versioning
C++ standard library	Compiler and flag dependent
Windows Win32	Stable since 1993

7. glibc uses **symbol versioning**: one shared object can export `memcpy@GLIBC_2.2.5` and a newer variant at once, so old binaries keep the old behaviour. Inspect it with `objdump -T /lib/x86_64-linux-gnu/libc.so.6`.
8. Linus Torvalds’s rule for the kernel, stated repeatedly on the kernel mailing list since the 1990s, is that breaking userspace is never acceptable, whatever the technical merit. That policy is why binaries from 1998 still run.

9. Tools: `abidiff` from `libabigail` compares two builds and reports ABI changes; `nm -D --defined-only` lists exported symbols; `readelf -d` shows the SONAME that encodes the ABI version; `pkg-config --modversion` reports the declared version.
10. Deprecation in code is itself part of the contract: `__attribute__((deprecated))` in GCC and Clang, `@Deprecated` in Java, `#[deprecated]` in Rust, `DeprecationWarning` in Python. A warning is a promise of a future break.

■ 36.3.6 WORDS – remember these

1. Signature — the name, inputs and output of a function — the typed interface of a callable, checked by the compiler at the call site.
2. Header — a file listing signatures without code — a declaration-only translation unit included to satisfy the compiler.
3. ABI — the machine-level agreement — application binary interface: calling convention, register use, structure layout and symbol naming.
4. Breaking change — a change that makes working code stop working — any change violating the published compatibility promise for that version position.
5. Semantic versioning — three numbers that say how risky an upgrade is — MAJOR.MINOR.PATCH per the SemVer 2.0.0 convention of 2013.
6. Symbol versioning — one library holding old and new behaviour at once — versioned symbol names in ELF so old binaries bind to old semantics.
7. Deprecation — a warning that something will be removed — a documented period during which an interface still works but is scheduled for removal.

36.4 System call APIs

■ 36.4.1 PLAIN – in simple words

1. Chapter 18 covered operating systems. Here is the one-line recap needed now.
2. A running program is not allowed to touch the disk, the network card or another program's memory. The operating system owns those.
3. So the program asks. That request is a **system call**, and the list of possible requests is the operating system's API.
4. This is the most important API on any machine, because everything else is built on top of it.
5. Different operating systems chose different lists. That is why a program written for one does not simply run on another.
6. To fix that, people wrote down a common list that many systems agreed to provide. That agreement is called **POSIX**.
7. Windows did not follow POSIX. It has its own list, called **Win32**.
8. So writing a file looks different on each, and that difference is the reason for a huge amount of the software world's duplicated effort.

■ 36.4.2 PLAIN – a picture in your head

1. Think of a bank counter with a thick glass screen.
2. You cannot reach the money. You write on a slip, push it under the glass, and the clerk does the work.
3. The list of slips the bank accepts is the system call API. There are maybe 350 kinds of slip on Linux.
4. POSIX is like an agreement between many banks in many countries to accept the same slips, so a traveller can use any of them.

5. Win32 is a bank that never signed that agreement and prints its own slips, which are longer and ask for more detail.

Where this comparison breaks:

6. A bank clerk can ask you what you meant. A kernel cannot. A malformed slip returns a number and nothing else.
7. A bank does one thing at a time per counter. A kernel handles millions of these per second and must keep every program's slips separate and safe.

■ 36.4.3 PLAIN – a worked example

1. Writing the five bytes “hello” to a new file, on both systems, cut to the bone.

```
/* POSIX */
int fd = open("out.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
write(fd, "hello", 5);
close(fd);

/* Win32 */
HANDLE h = CreateFileA("out.txt", GENERIC_WRITE, 0, NULL,
                      CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
DWORD written;
WriteFile(h, "hello", 5, &written, NULL);
CloseHandle(h);
```

2. Same task. Same result on disk. Completely different shape.
3. POSIX gives back a small integer, the **file descriptor**, usually 3 for your first opened file, since 0, 1 and 2 are already taken by input, output and errors.
4. Win32 gives back a **HANDLE**, an opaque value that is not an index into anything you can reason about.
5. POSIX signals failure by returning -1 and setting a separate global number called `errno`. Win32 returns a special invalid handle value and you call `GetLastError()`.
6. POSIX packs the options into one integer of flags. Win32 spreads them across seven separate arguments, several of which are usually zero or `NULL`.
7. Notice `write` returns how many bytes it wrote. It may write fewer than you asked. Ignoring that return value is one of the classic C bugs.

■ 36.4.4 PLAIN – what is really happening inside

1. A system call is not an ordinary function call. It crosses a privilege boundary, and that is deliberate and expensive.
2. Your program puts a number identifying the call into a register, puts the arguments into other registers, and executes one special instruction.
3. On x86-64 Linux that instruction is `syscall`, and the call number goes in register `rax`. Write is number 1.
4. The processor switches from user mode to kernel mode, jumps to a fixed handler, and the kernel checks everything you passed.
5. The kernel does the work, puts a result back in a register, and returns to user mode.
6. The C library function `write()` is a thin wrapper around this. It is not the system call itself; it is a normal function that performs one.
7. POSIX does not define the instruction, the register or the call number. It defines the C function and its behaviour. That is the portability layer.

8. So POSIX is an API standard, not an ABI standard: source portability, not binary portability. Code recompiles; compiled binaries do not travel.

■ 36.4.5 TECHNICAL – the engineer’s version

1. POSIX is IEEE Std 1003.1, first published 1988. The current joint text is maintained by the Austin Group as the Single UNIX Specification and IEEE Std 1003.1; the 2024 edition is the current revision at the time of writing in 2026.
2. Real conformance, honestly stated: Linux is broadly POSIX-compatible but has never been certified. macOS is certified UNIX. AIX and z/OS are certified. Windows is not POSIX and dropped its old POSIX subsystem long ago.
3. Windows Subsystem for Linux version 2, shipped from 2019, runs a real Linux kernel in a lightweight virtual machine rather than translating calls. That is why its system call fidelity is high and its file access across the Windows boundary is slow.
4. Comparison of the two families at the same job:

Task	POSIX	Win32
Open a file	open	CreateFileW
Write bytes	write	WriteFile
Handle type	int descriptor	HANDLE
Error report	errno	GetLastError
Start a process	fork + exec	CreateProcess
Load a library	dlopen	LoadLibrary
Memory map	mmap	MapViewOfFile

5. Cost, measured. A system call on modern x86-64 Linux costs roughly 100 to 300 nanoseconds when nothing goes wrong, against roughly 1 to 2 nanoseconds for a plain function call. Kernel page-table isolation, added in January 2018 against the Meltdown vulnerability, increased that cost measurably on affected processors.
6. This is why high-throughput code avoids system calls: buffered I/O to batch write calls, `io_uring` on Linux since kernel 5.1 in 2019 to submit many operations with one crossing, and kernel bypass networking for the extreme cases.
7. Observation tools: `strace -c ./prog` counts and times every system call; `lttrace` does the same for library calls; `dttruss` is the macOS equivalent; Windows uses Process Monitor and Event Tracing for Windows.
8. Linux keeps roughly 350 system calls on x86-64, and the list is architecture-specific. `ausyscall --dump` or the table in `arch/x86/entry/syscalls/syscall_64.tbl` in the kernel source gives the numbers.

■ 36.4.6 WORDS – remember these

1. System call — a request from a program to the operating system — a controlled entry into kernel mode with checked arguments.
2. File descriptor — the small number that stands for an open file — a per process index into the kernel’s open file table, an `int` on POSIX.
3. POSIX — the agreed common list of Unix-style calls — IEEE Std 1003.1, first published 1988, source-level portability across conforming systems.
4. Win32 — the Windows system interface — the C API exported by `kernel32`, `user32`, `gdi32` and friends, ABI-stable since Windows NT 3.1 in 1993.

5. User mode and kernel mode — the two privilege levels — CPU rings restricting which instructions and memory a running thread may touch.
6. Wrapper — the ordinary function that performs the special instruction — the libc stub that loads the call number and executes `syscall`.

36.5 Remote APIs: what the network changes

■ 36.5.1 PLAIN – in simple words

1. This is the single biggest conceptual jump in a developer's life, and most people make it without noticing, which is why it hurts.
2. A local function call either happens or the program crashes. There is no third case.
3. A remote call has a third case, and a fourth, and a fifth.
4. It can fail before arriving. It can arrive and fail. It can arrive, succeed, and the answer can be lost on the way home.
5. In that last case, the work was done and you do not know it. You cannot know it. There is no way to find out from the failure itself.
6. This is called **partial failure**, and it does not exist inside one machine.
7. Also, remote calls take real time. A local call costs about a nanosecond. A call across a city costs about ten million nanoseconds.
8. That is a factor of ten million. No amount of clever coding hides a factor of ten million.
9. So the rule is: never pretend a remote call is a local call. Every framework that tried to hide the difference eventually failed at exactly this point.

■ 36.5.2 PLAIN – a picture in your head

1. Asking a colleague sitting beside you for a number is a local call. You ask, they answer, you carry on.
2. Now the colleague moves to another building and you must send letters.
3. You post a letter asking for the number. Silence for a week.
4. Did the letter get lost? Did they get it and not reply yet? Did they reply and that letter got lost? Did they leave the company?
5. You cannot tell. All four look identical from your desk.
6. Now suppose the letter asked them to transfer money. Do you send a second letter? If the first one arrived, you have transferred twice.
7. That is the entire problem, and section 36.10 gives the fix, which is to put a unique reference number on the letter.

Where this comparison breaks:

8. Letters arrive in days. Network messages arrive in milliseconds, which tempts you into thinking they are instant. They are not; they are just fast.
9. The post has no equivalent of a message being delivered twice because your own side automatically resent it. Networks do that constantly.

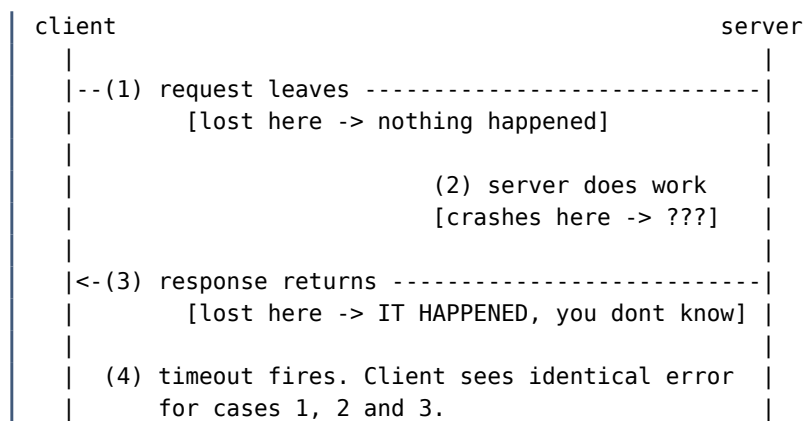
■ 36.5.3 PLAIN – a worked example

1. The reader's own session shows this exactly. From a flat in India, `curl -v https://github.com` printed `Trying 20.207.73.82:443...` and then timed out after 15 seconds with no response of any kind.
2. Nothing came back. No refusal, no error message from the network, nothing. Silence.

3. On the same phone, over mobile data, the same site loaded instantly.
4. Now ask the question that matters. Was GitHub down? No: the mobile test proves it was serving.
5. Was the request received? Unknown, and unknowable from the client side.
6. For a GET that does not matter, because reading a page twice is harmless.
7. For a POST that creates something, it matters enormously.
8. In the same session, two `git push` operations failed with `send-pack: unexpected disconnect while reading sideband packet`. Both times the push had not been applied, but that could not be known from the error.
9. The disconnect happened while reading the *response*. A lost response tells you nothing about whether the *write* landed. The only way to find out was to ask the server again afterwards.
10. That is partial failure, observed in a real session, on a real evening.

■ 36.5.4 PLAIN – what is really happening inside

1. Here is the timeline of one remote call, with the failure windows marked.



2. Case 1 and case 3 produce the same visible result at the client: a timeout.
3. In case 1 retrying is correct and free. In case 3 retrying may double the effect. The client cannot tell them apart.
4. This is not a bug in any particular system. It is a proved limit. The general result is that no protocol can guarantee both sides agree, in the presence of arbitrary message loss.
5. So the industry does not solve it. It works around it, in three ways.
6. Make operations **idempotent**, so that doing them twice equals doing them once. Then retrying is always safe.
7. Or attach a unique key to each attempt, so the server can recognize a repeat and return the first answer instead of acting again.
8. Or accept the risk and reconcile later, which is what accounting systems did for centuries before computers.

■ 36.5.5 TECHNICAL – the engineer's version

1. The eight fallacies of distributed computing. The list originated at Sun Microsystems: L. Peter Deutsch drafted seven in 1994, drawing on earlier work by Bill Joy and Dave Lyon, and James Gosling added the eighth around 1997. Attribution of individual items is disputed in the sources; the list is not.

#	Fallacy	What it costs you
1	The network is reliable	No retry path

#	Fallacy	What it costs you
2	Latency is zero	Chatty designs
3	Bandwidth is infinite	Huge payloads
4	The network is secure	Plaintext, no auth
5	Topology does not change	Hardcoded addresses
6	There is one administrator	No cross-team plan
7	Transport cost is zero	Ignored serialization
8	The network is homogeneous	Protocol assumptions

2. Fallacy 1. Packets are dropped, links flap, and TCP connections are reset by middleboxes. Every remote call needs an explicit failure path, not an exception handler bolted on afterwards.
3. Fallacy 2. Latency has a floor set by physics. Light in fibre travels about 200,000 km per second, so Mumbai to London and back, about 14,400 km of fibre, cannot beat roughly 72 ms of pure propagation, and measures 110 to 130 ms in practice. A design that makes 50 sequential calls has bought itself six seconds before writing a line of logic.
4. Fallacy 3. Bandwidth is finite and often asymmetric, and mobile uplinks are far slower than downlinks. Payload size is a design decision, not an afterthought.
5. Fallacy 4. Everything crossing a network is hostile territory. This is why Chapter 32 covered TLS and why section 36.9 covers tokens.
6. Fallacy 5. Addresses change. Containers get new addresses on every restart, which is why service discovery and DNS-based routing exist.
7. Fallacy 6. Your call crosses networks owned by people you will never meet. In the reader's traceroute, the path crossed the home router, the ISP's private core at addresses like 172.31.0.17, the ISP's public edge at 137.97.29.249, and then Microsoft's backbone through Delhi, Mumbai and Pune. That is at least three administrative domains, none of which the reader could telephone.
8. Fallacy 7. Serializing and deserializing costs CPU and memory. At Google scale this is why Protocol Buffers exist; JSON parsing at millions of requests per second is a real budget line.
9. Fallacy 8. Different operating systems, TLS versions, HTTP versions, proxies and MTUs. A path that works over one network may silently fail over another, which is precisely what the reader observed between broadband and mobile.
10. The standard model: **at-most-once** delivery risks losing work, **at-least-once** risks duplicating it, and **exactly-once** delivery is not achievable at the transport level. What is achievable is exactly-once *effect*, built from at-least-once delivery plus idempotency. Say it that way and you will never be wrong.

■ 36.5.6 WORDS - remember these

1. Partial failure — the work happened but you never found out — a call whose outcome is unknown to the caller because the response was lost.
2. Idempotent — doing it twice is the same as doing it once — an operation whose repeated application produces the same state as a single application.
3. Latency — the waiting time before an answer starts — round-trip time, floored by propagation delay and inflated by queueing and processing.
4. Timeout — giving up after waiting — a client-side deadline that converts an unknown outcome into a definite local error.
5. At-least-once — it will arrive, maybe more than once — a delivery guarantee requiring the receiver to tolerate duplicates.

6. Exactly-once effect — the result happens once even if the message does not — at-least-once delivery combined with server-side deduplication.

36.6 HTTP APIs in practice

■ 36.6.1 PLAIN – in simple words

1. Most APIs you meet today are carried over HTTP, the same protocol your browser uses for web pages.
2. That was not a technical decision at first. It was a political one. Port 443 was the one port that was open everywhere.
3. An HTTP exchange has exactly two messages: a **request** you send and a **response** you get back.
4. A request has four parts: a **method** saying what kind of action, a **URL** saying what to act on, **headers** giving extra facts, and an optional **body** carrying data.
5. A response has three parts: a **status code**, a three-digit number saying how it went, then headers, then an optional body.
6. The methods you need are GET to read, POST to create, PUT to replace, PATCH to change part, and DELETE to remove.
7. The body is almost always **JSON** these days: text that describes data using braces, square brackets, names and values.
8. That is the whole thing. Everything else in this section is detail.

■ 36.6.2 PLAIN – a picture in your head

1. An HTTP request is a filled-in form pushed through a letterbox.
2. The method is the big word at the top: COLLECT, DEPOSIT, REPLACE, DESTROY.
3. The URL is the address line: which office, which department, which file.
4. The headers are the small boxes down the side: who you are, what format you can read, how long you will wait.
5. The body is the attached sheet, and only some kinds of form have one.
6. The response comes back with a three-digit stamp at the top. 200 means done. 404 means no such file. 500 means the office caught fire.

Where this comparison breaks:

7. Paper forms are read by a person who can guess your intent. HTTP is parsed by a machine, and an unexpected header can be silently ignored or fatal.
8. A form goes to one office. An HTTP request may pass through a browser cache, a company proxy, a content delivery network and a load balancer, any of which may answer it without the real server ever hearing about it.

■ 36.6.3 PLAIN – a worked example

1. Here is a complete raw exchange, exactly as it goes on the wire, with every part labelled. This is what `curl -v` shows you.

```
POST /v1/orders?notify=true HTTP/1.1
Host: api.example.com
Authorization: Bearer sk_live_9f2a...
Content-Type: application/json
Accept: application/json
Content-Length: 52
Idempotency-Key: 8f14e45f-ea4f-4c1b-9b4a-0c1d2e3f4a5b
```

```
{"item_id":"sku_771","qty":2,"currency":"INR"}
```

2. Line 1 has three things: the method POST, the path /v1/orders, and the query string ?notify=true. Then the protocol version.
3. Host says which site, because one server address may serve thousands.
4. Authorization carries the credential. Bearer means “whoever holds this may act”, which is exactly as dangerous as it sounds.
5. Content-Type says what the body is. Accept says what you want back. These are two different things and mixing them up is a common mistake.
6. Content-Length is the body size in bytes. Idempotency-Key is a unique reference for this attempt, explained in section 36.10.
7. A blank line ends the headers. Everything after it is the body.
8. The response:

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /v1/orders/ord_5512
RateLimit-Remaining: 98
Cache-Control: no-store

{"id":"ord_5512","status":"pending","total":1798}
```

9. 201 Created says a new thing exists. Location says where it now lives.
10. RateLimit-Remaining says how many more calls you may make in this window.
11. Cache-Control: no-store forbids anyone in the middle from keeping a copy.

■ 36.6.4 PLAIN – what is really happening inside

1. Two properties decide how a method may be treated, and they are not the same property.
2. **Safe** means the call is not supposed to change anything. It is a read.
3. **Idempotent** means doing it many times leaves the same state as doing it once. Safe methods are automatically idempotent.
4. GET, HEAD and OPTIONS are safe. PUT and DELETE are idempotent but not safe. POST and PATCH are neither.
5. Why this matters in practice: browsers, proxies and client libraries are allowed to retry safe and idempotent requests automatically.
6. So DELETE /orders/5512 sent twice must leave the order deleted, not produce an error the second time. Many APIs get this wrong.
7. And POST /orders sent twice creates two orders. That is correct behaviour and it is exactly why idempotency keys exist.

Method	Safe	Idempotent	Body
GET	Yes	Yes	No
HEAD	Yes	Yes	No
OPTIONS	Yes	Yes	No
POST	No	No	Yes
PUT	No	Yes	Yes
PATCH	No	No	Yes
DELETE	No	Yes	Rare

■ 36.6.5 TECHNICAL – the engineer’s version

1. HTTP semantics are now specified in RFC 9110, published June 2022, which replaced the RFC 7230 to 7235 series from 2014, which in turn replaced RFC 2616 from June 1999. Quote RFC 9110 for semantics and RFC 9112 for HTTP/1.1 message syntax.
2. JSON is specified in RFC 8259, December 2017, with media type `application/json`. It has no comment syntax, no trailing commas, and its numbers are IEEE 754 doubles in practice, so integers above 2^{53} lose precision. Send large identifiers as strings.
3. The status codes worth knowing, and when to use each:

Code	Meaning	Use when
200	OK	Read or update succeeded
201	Created	New resource made
202	Accepted	Queued, not done yet
204	No Content	Success, nothing to send
301	Moved Permanently	URL changed forever
304	Not Modified	Client’s cache is valid
400	Bad Request	Malformed or invalid input
401	Unauthorized	No or bad credential
403	Forbidden	Valid identity, not allowed
404	Not Found	No such resource
405	Method Not Allowed	Wrong verb for that URL
409	Conflict	State clash, version clash
410	Gone	Deleted on purpose
415	Unsupported Media Type	Wrong Content-Type sent
422	Unprocessable Content	Syntax fine, semantics bad
429	Too Many Requests	Rate limit hit
500	Internal Server Error	Server bug
502	Bad Gateway	Upstream returned garbage
503	Service Unavailable	Overloaded or in maintenance
504	Gateway Timeout	Upstream did not answer

4. The 401 versus 403 distinction is worth memorizing, because it is wrong in most codebases. 401 means “I do not know who you are, authenticate”; the header name `WWW-Authenticate` is required with it. 403 means “I know exactly who you are and you still may not”.
5. 429 comes from RFC 6585, April 2012, which also defined 428, 431 and 511. Pair it with `Retry-After`, which takes either seconds or an HTTP date.
6. 4xx means the client should change something before retrying. 5xx means the client may retry the identical request later. That single rule drives most retry logic.
7. Content negotiation uses `Accept`, `Accept-Encoding` and `Accept-Language` in the request against `Content-Type`, `Content-Encoding` and `Content-Language` in the response, with `Vary` telling caches which of them affected the answer. Omitting `Vary` is a classic cache poisoning bug.
8. Conditional requests: the server sends `ETag`, the client sends `If-None-Match` on the next read and gets 304 with no body, or sends `If-Match` on a write and gets 412 if someone else changed it first. That second pattern is optimistic concurrency control and costs almost nothing.
9. HTTP/2, RFC 9113, multiplexes many requests over one TCP connection and compresses headers with HPACK. HTTP/3, RFC 9114, runs over QUIC on UDP, removing head-of-line blocking at the transport layer. Both change performance dramatically and change the semantics above not at all.

■ 36.6.6 WORDS - remember these

1. Method — the verb saying what kind of action — the HTTP request method defining semantics, safety and idempotency per RFC 9110.
2. Status code — the three-digit result — 1xx informational, 2xx success, 3xx redirection, 4xx client error, 5xx server error.
3. Header — a small labelled fact attached to a message — a field name and value carrying metadata separate from the body.
4. Safe method — a read that changes nothing — a method whose semantics are defined as read-only, therefore automatically retryable.
5. ETag — a short label for a version of a thing — an opaque validator used with If-None-Match and If-Match for caching and concurrency.
6. Content negotiation — asking for the format you can read — matching Accept family request fields against server representations, signalled by Vary.

36.7 REST, properly

■ 36.7.1 PLAIN - in simple words

1. REST is the most misused word in this entire subject.
2. It was defined in 2000 by Roy Fielding in his doctoral thesis. He was describing why the web itself works so well at enormous scale.
3. He did not describe a technology. He described a set of six rules that an architecture may follow, and what each rule buys you.
4. The rules are: separate client from server; keep no client state on the server between calls; say clearly what may be cached; allow layers in between; use one uniform way of addressing and acting on things; and, optionally, allow the server to send code the client runs.
5. Almost nothing sold as REST today follows all six.
6. Most of what people call REST is really “JSON sent over HTTP with nouns in the URL”. That is a useful style. It is not what the thesis specified.
7. This matters less than purists claim and more than beginners think. It matters because two of the six rules, statelessness and cacheability, are the reason the web scales, and people abandon them without realizing.

■ 36.7.2 PLAIN - a picture in your head

1. Think of a public library with open shelves.
2. Every book has a fixed shelf mark. You can hand that mark to anyone and they find the same book. That is addressability.
3. The librarian does not remember you between visits. You bring your card each time. That is statelessness, and it means any librarian can serve you.
4. Some books are marked “reference, always current” and some “loan copy, fine for a month”. That is cacheability.
5. And each book ends with a list of related shelf marks, so you can keep going without asking anyone. That last one is the rule everybody skips.

Where this comparison breaks:

6. A library has one building. A REST system may have caches and proxies that answer you without the library ever knowing you asked.

■ 36.7.3 PLAIN – a worked example

1. Resource-oriented design in one page. The nouns get URLs; the verbs come from HTTP.

GET	/v1/orders	list orders
POST	/v1/orders	create one
GET	/v1/orders/ord_5512	read one
PATCH	/v1/orders/ord_5512	change some fields
DELETE	/v1/orders/ord_5512	remove it
GET	/v1/orders/ord_5512/items	sub-collection

2. Plural nouns, no verbs in the path, lower case, hyphens not underscores. All of that is **convention**, not standard. Nothing enforces it.
3. Filtering, sorting and paging go in the query string, never in the path: GET /v1/orders?status=paid&sort=-created_at&limit=50.
4. Now paging, which is where designs break at scale. Two options.
5. Offset paging: ?limit=50&offset=1000. Simple, and lets you jump to page 21.
6. Cursor paging: ?limit=50&after=eyJpZCI6Nzcx... The cursor is an opaque marker for “the last row you saw”.
7. Cursor wins at scale for two reasons. First, a database must count and skip 1,000 rows to serve offset 1,000, so page 400 is far slower than page 1.
8. Second, if a new row is inserted while you page, offset paging shows you one row twice and skips another. Cursor paging does not.
9. Partial responses let the client ask for less: ?fields=id,status,total. That is a convention borrowed from GraphQL’s core idea.

■ 36.7.4 PLAIN – what is really happening inside

1. The rule that carries the most weight is statelessness.
2. Stateless means every request carries everything needed to understand it. The server keeps nothing about you between calls.
3. The payoff is that any server can answer any request. So you can put ten servers behind one address, and add an eleventh at lunchtime.
4. The moment you store a session in one server’s memory, that request must come back to that server, and your ten servers become ten fragile pets.
5. The cost of statelessness is that every request repeats its credentials and context, so requests are bigger. That trade was made deliberately.
6. The rule everyone skips is called **HATEOAS**: hypermedia as the engine of application state. It means the response tells you what you can do next, by including links, so the client never hard-codes URLs.
7. The web browser does this perfectly. You do not type URLs; you follow links a server gave you. Almost no JSON API does it.

The honest version: an API without HATEOAS is not REST as Fielding defined it.

8. Fielding said so directly in a 2008 post on his own site titled “REST APIs must be hypertext-driven”. The industry read it, agreed, and carried on.
9. Practical position: use the six constraints as a checklist of trade-offs, not a purity test. Know which ones you are breaking and why.

■ 36.7.5 TECHNICAL – the engineer’s version

1. The six constraints from Chapter 5 of Fielding’s 2000 dissertation, with what each one actually buys:

Constraint	What it buys
Client-server	Independent evolution
Stateless	Horizontal scaling
Cacheable	Fewer round trips
Layered system	Proxies, CDNs, gateways
Uniform interface	Generic tooling
Code on demand	Optional, rarely used

2. Code on demand is the only optional constraint. JavaScript delivered to a browser is the canonical example, and it is why the web can extend clients without redeploying them.
3. The uniform interface itself decomposes into four sub-constraints: identification of resources, manipulation through representations, self-descriptive messages, and hypermedia as the engine of application state.
4. A **resource** in Fielding’s terms is a concept, not a row and not a file. A **representation** is one rendering of it at one moment, in one media type. The URL identifies the resource; the response body is only a representation.
5. Richardson’s Maturity Model, presented by Leonard Richardson in 2008, classifies real APIs: level 0 is one URL and one verb, level 1 adds resources, level 2 adds HTTP verbs and status codes, level 3 adds hypermedia. Most commercial APIs sit at level 2. This is a **convention** for describing maturity, not a standard.
6. Hypermedia formats that do exist: HAL, JSON:API version 1.1, Siren, and application/problem+json from RFC 9457, July 2023, which replaced RFC 7807 from March 2016 for error bodies.
7. Cursor paging in production: GitHub uses Link headers with rel="next", an application of RFC 8288 web linking. Stripe uses starting_after with an object identifier. Both are cursor schemes; neither exposes an offset.
8. Cost of offset paging, measured: a LIMIT 50 OFFSET 1000000 on PostgreSQL must scan and discard a million rows, taking seconds where a keyset query using WHERE id > ? ORDER BY id LIMIT 50 on an indexed column takes sub-millisecond time regardless of depth.
9. Statelessness has a cost you can measure: repeating a 900-byte JWT on every request costs 900 bytes uplink per call. HTTP/2 HPACK header compression removes most of that repetition on a persistent connection.

■ 36.7.6 WORDS – remember these

1. REST — a style of network architecture with six rules — Representational State Transfer, defined in Chapter 5 of Fielding’s 2000 dissertation.
2. Stateless — the server remembers nothing about you between calls — each request carries all context needed for the server to understand it.
3. Resource — the thing a URL names — a concept with an identifier, distinct from any single representation of it.
4. Representation — one rendering of a thing at one moment — the bytes plus media type returned for a resource.

5. HATEOAS — the answer tells you what you can do next — hypermedia as the engine of application state, the constraint almost nobody implements.
6. Cursor paging — “carry on from this row” — keyset pagination using an opaque marker, stable under insertion and constant-time at any depth.

36.8 The alternatives, compared honestly

■ 36.8.1 PLAIN – in simple words

1. REST-style JSON over HTTP is the default, not the only choice. Four alternatives matter.
2. **GraphQL** gives one endpoint and lets the client write a query saying exactly which fields it wants. The server returns exactly that shape.
3. **gRPC** sends compact binary messages defined in advance by a schema file. It is fast and strictly typed, and it can stream in both directions.
4. **SOAP** is the older XML approach. It is verbose and heavy, and it is still running in banking, insurance, telecoms and government.
5. **JSON-RPC** is the minimal option: one endpoint, a method name, parameters, an identifier. About two pages of specification in total.
6. And for pushing data to a client, **WebSockets** give a two-way permanent connection, while **server-sent events** give a simpler one-way stream from server to client.
7. None of these replaces the others. Each solves a specific pain.

■ 36.8.2 PLAIN – a picture in your head

1. Ordering food. REST is a menu with fixed dishes; you order dish 12 and get what dish 12 contains, including the garnish you did not want.
2. GraphQL is telling the kitchen your exact ingredients, and getting a plate with only those. Wonderful for you, much harder for the kitchen to plan.
3. gRPC is a factory canteen with pre-printed cards and numbered slots. Fast, rigid, and useless to a visitor who does not have the card printer.
4. WebSockets are leaving the intercom switched on both ways all evening.

Where this comparison breaks:

5. A kitchen can refuse a silly order. A GraphQL server can be handed a query that legally asks for a million rows through four levels of nesting, and will try, unless somebody wrote a cost limit.

■ 36.8.3 PLAIN – a worked example

1. The same request, three ways. First GraphQL:

```
POST /graphql
{"query":"{ user(id:42){ name photos(first:2){ title } } }"}

{"data":{"user":{"name":"Asha","photos":[
  {"title":"Pune, monsoon"}, {"title":"Bhor ghat"}]}}}
```

2. One call, one round trip, exactly the fields asked for. With REST this often needs two calls: one for the user, one for the photos.
3. That is the **over-fetching** and **under-fetching** problem GraphQL solves, and on a mobile connection it is worth real seconds.
4. Now the same thing as a gRPC schema, written in Protocol Buffers:

```
message PhotoRequest { int64 user_id = 1; int32 first = 2; }
message Photo { string title = 1; int64 taken_at = 2; }
service Photos {
  rpc List(PhotoRequest) returns (stream Photo);
}
```

- The numbers 1 and 2 are field tags. They go on the wire instead of the field names, which is most of why the encoding is small.
- `stream Photo` means the server can send photos one at a time as it finds them, rather than making the client wait for all of them.
- And JSON-RPC, for contrast, is this and nothing more:

```
{ "jsonrpc": "2.0", "method": "listPhotos",
  "params": { "user": 42, "id": 7 }
```

■ 36.8.4 PLAIN – what is really happening inside

- Every one of these has a cost, and the honest version of each is the part the marketing leaves out.
- GraphQL solves over-fetching and creates two new problems.
- Problem one is caching. HTTP caching works on URLs and methods. GraphQL sends everything as a `POST` to one URL, so every proxy, CDN and browser cache in the world becomes useless to it.
- Problem two is the **N+1 query problem**. Asking for 100 users and each user's photos naively becomes 1 query for users plus 100 for photos. The standard fix is a batching loader that gathers the 100 into one query, but you must build it. The API shape hides the database cost from the caller.
- gRPC is fast and typed, and it is rare in browsers, for a concrete reason: it needs control over HTTP/2 frames and trailers that browser JavaScript cannot access. The workaround, gRPC-Web, needs a proxy to translate, so it is extra machinery.
- SOAP is heavy but it brought things REST had to reinvent: a machine-readable contract in WSDL, and standards for signing and encrypting individual message parts rather than the whole channel.
- WebSockets give you a persistent two-way channel, and in exchange you now own reconnection, heartbeats, backpressure and message ordering yourself.
- Server-sent events do far less and cost far less: it is plain HTTP, it reconnects automatically, and the browser gives you an event identifier so you can resume. It only goes one way, which is what most apps need.

■ 36.8.5 TECHNICAL – the engineer's version

- Comparison on the axes that decide real choices:

Aspect	REST/JSON	GraphQL	gRPC
Payload	Text, medium	Text, tuned	Binary, small
Browser	Native	Native	Needs proxy
Typing	Optional	Schema, strong	Schema, strong
Streaming	SSE or WS	Subscriptions	Native, 2-way
HTTP caching	Excellent	Poor	Not applicable
Tooling	Universal	Good, newer	Good, generated

- Encoding sizes, illustrative, for one small record of five fields: SOAP envelope roughly 1,200 bytes, JSON roughly 150 bytes, Protocol Buffers roughly 40 to 60 bytes. Exact figures depend entirely on field names and values, so measure your own payloads rather than quoting these.

3. Protocol Buffers achieve that by sending field tag numbers instead of names, using variable-length integers, and omitting fields at default values. The cost is that the message is unreadable without the schema.
4. gRPC supports four call shapes: unary, server streaming, client streaming and bidirectional streaming. It runs over HTTP/2, uses trailers for status, and carries a status code set of its own, where OK is 0 and DEADLINE_EXCEEDED is 4.
5. GraphQL's specification is maintained by the GraphQL Foundation, hosted by the Linux Foundation since 2018. Practical defences you must add yourself: query depth limits, query cost analysis, persisted queries, and a batching loader such as DataLoader.
6. SOAP lives on where contracts and message-level security matter: SWIFT and ISO 20022 messaging in banking, many telecom provisioning systems, and government tax and customs filing. It is not a legacy curiosity; it is running production traffic in 2026.
7. JSON-RPC 2.0, specified in 2010, is used by Ethereum nodes, the Language Server Protocol behind editor tooling, and the Model Context Protocol used for tool calling by AI systems. Small specifications survive.
8. Push mechanisms: WebSockets are RFC 6455, December 2011, upgrading an HTTP connection to a two-way frame protocol. Server-sent events are part of the HTML living standard, media type `text/event-stream`, one way, with automatic reconnection and a `Last-Event-ID` header for resumption. Long polling still exists and still works where nothing else gets through.
9. Selection rule, stated plainly. Public API for unknown clients: REST. Many different client shapes over one graph of data: GraphQL. Internal service-to-service at high volume: gRPC. Live updates to a browser, one way: server-sent events. Live two-way, such as a game or a chat: WebSockets.

■ 36.8.6 WORDS - remember these

1. Over-fetching — getting more fields than you needed — a fixed representation returning data the client discards, wasting bandwidth and battery.
2. N+1 problem — one query becomes a hundred — resolving a list then issuing one dependent query per element instead of one batched query.
3. Protocol Buffers — a compact binary format with a schema file — Google's IDL and wire format using numbered field tags and varint encoding.
4. Streaming — sending results as they appear — a call that yields many messages over one open call instead of one response.
5. WebSocket — a permanent two-way channel — RFC 6455 framing over an upgraded HTTP connection, full duplex, no request-response structure.
6. Server-sent events — a one-way live feed over plain HTTP — the `text/event-stream` media type with automatic client reconnection.

36.9 Authentication and authorization for APIs

■ 36.9.1 PLAIN - in simple words

1. Two different questions, and confusing them causes real breaches.
2. **Authentication** asks: who are you? **Authorization** asks: are you allowed to do this particular thing?
3. An **API key** is a long random string you send with each request. It answers neither question well. It identifies the caller's account and nothing else.
4. **Basic auth** sends a username and password on every request, lightly encoded, not encrypted. Only safe inside TLS, and even then it is blunt.

5. A **bearer token** is a string that means “whoever holds this may act”. It is the standard today. It usually expires, which is its main improvement.
6. **OAuth 2.0** is the system that lets you grant one app limited access to your account on another service, without giving it your password.
7. A **scope** is a label on a token limiting what it may do. Not who you are. What this token may do.
8. A **JWT** is a token with readable information inside it and a signature proving nobody edited it.
9. **mTLS** is where both sides present certificates, so the connection itself proves identity before any request is sent.

■ 36.9.2 PLAIN – a picture in your head

1. Think of a hotel.
2. Your passport at the desk is authentication. It proves who you are, once.
3. The room card you get is a bearer token. It expires on checkout day, and anyone who finds it can open your room. It does not know your name.
4. The card opens room 402 and the gym, but not the wine cellar. Those are scopes.
5. Handing your passport to a tour company so they can collect your parcel is the bad old way. OAuth is instead telling the desk “issue a card that opens only the parcel locker, valid for one hour”.
6. And mTLS is the hotel checking your face against a photo while you check the hotel’s licence on the wall. Both sides prove identity.

Where this comparison breaks:

7. A hotel card is physical, so only one person can hold it. A leaked token can be copied a million times in a second, and the copies are indistinguishable.

■ 36.9.3 PLAIN – a worked example

1. The OAuth 2.0 authorization code flow with PKCE, step by step. This is the flow you should use for web apps, mobile apps and command-line tools alike.

user	app	auth server	API
	-1 make verifier+chal->		
	<-2 open browser to /authorize-->		
	-3 log in and approve ----->		
	<-4 redirect with code-		
	-5 POST code+verifier->		
	<-6 access + refresh -->		
	-7 Bearer access token ----->		
	<-8 data or 403 if scope missing ----->		

2. Step 1. The app invents a random secret called the **code verifier**, then hashes it to make a **code challenge**. Real values, computed with SHA-256:

```
verifier : AAECaWQFBgcICQoLDA00DxAREhMUFYXGBkaGxwdHh8
challenge: 6oZqdX5M0Lq_qBJ8vppAnT4fk6AP8UiP9zX8-Rev_9A
```

3. Step 2. The app opens the user’s browser at the authorization server, sending its client identifier, the redirect address, the scopes it wants, a random state value, and the challenge. Never the verifier.
4. Step 3. The user logs in at the authorization server, not at the app. The app never sees the password. That is the entire point of OAuth.

■ 36.9.5 TECHNICAL – the engineer’s version

1. Specifications, with dates: OAuth 2.0 is RFC 6749, October 2012. Bearer token usage is RFC 6750, October 2012. PKCE is RFC 7636, September 2015. JWT is RFC 7519, May 2015. JWS signatures are RFC 7515. Token introspection is RFC 7662. Token revocation is RFC 7009.
2. Current guidance is RFC 9700, Best Current Practice for OAuth 2.0 Security, published January 2025. It requires PKCE for all client types, requires exact redirect URI matching, and rules out the implicit grant and the resource owner password credentials grant.
3. That is a change with a date, so state it as one: advice that was correct in 2015, such as “use the implicit flow for single-page apps”, is now explicitly wrong.
4. Credential mechanisms compared:

Mechanism	Expires	Scoped	Main weakness
API key	Rarely	Sometimes	Leaks live forever
Basic auth	No	No	Password on every call
Bearer token	Yes	Yes	Theft equals full use
JWT	Yes	Yes	Hard to revoke early
mTLS	Cert lifetime	Via cert fields	Certificate management

5. Why API keys are weak, precisely: they are long-lived, usually coarse-grained, frequently committed to source control, and often sent in query strings where they land in server access logs and browser history. GitHub’s secret scanning finds millions of leaked credentials a year.
6. JWT signature algorithms: HS256 uses one shared secret, so every verifier can also mint tokens. RS256 and ES256 use a private key to sign and a public key to verify, so verifiers cannot mint. Prefer asymmetric for anything crossing a trust boundary.
7. The classic JWT vulnerability is accepting `alg: none`, or accepting an attacker-chosen algorithm. Always pin the expected algorithm in the verifier rather than reading it from the token you are verifying.
8. Access tokens should be short-lived, commonly 5 to 60 minutes. Refresh tokens are long-lived and must be stored securely; RFC 9700 requires that refresh tokens for public clients be sender-constrained or rotated on each use, so that a stolen one is detectable when both copies are presented.
9. mTLS, mutual TLS, means the client presents an X.509 certificate during the handshake. It is standard inside service meshes such as Istio and Linkerd, where certificates are issued automatically and rotated hourly. RFC 8705 defines binding OAuth tokens to a client certificate, which stops a stolen token being used from another machine.
10. Observation: `curl -v` shows the Authorization header being sent, jwt command line tools or any base64 decoder reveal claims, and the token introspection endpoint of RFC 7662 tells you server-side whether a token is still live.

■ 36.9.6 WORDS – remember these

1. Authentication — proving who you are — verifying an identity claim against a credential the verifier trusts.
2. Authorization — deciding what you may do — evaluating a policy against an authenticated identity and a requested operation.
3. Bearer token — whoever holds it may act — a credential requiring no proof of possession beyond presentation, per RFC 6750.

4. Scope — a limit written into the token — a named restriction on the operations a specific token may perform, independent of the user's rights.
5. JWT — a signed, readable token — JSON Web Token, RFC 7519: base64url header, claims and signature, integrity-protected but not encrypted.
6. PKCE — proof that the app finishing the flow is the app that started it — RFC 7636 code verifier and S256 challenge, now required for all clients.
7. Refresh token — the long-lived key used to get new short-lived ones — a credential exchanged at the token endpoint, rotated or sender-constrained.
8. mTLS — both sides show certificates — mutual TLS authentication binding identity to the transport connection itself.

36.10 Making an API reliable in the real world

■ 36.10.1 PLAIN – in simple words

1. An API that works when everything is calm is easy. An API that works when things go wrong is the whole job.
2. Six habits do almost all of the work. Learn these six and you are ahead of most working code.
3. **Rate limiting** is the server saying “you may only ask me this many times in this much time”. It protects the server from being flattened.
4. **Retrying** is the client saying “that failed, let me try again”. Done badly, it turns a small problem into an outage.
5. **Backoff** means waiting longer before each retry. **Jitter** means adding a random amount to that wait so that everyone does not retry together.
6. An **idempotency key** is a unique reference you attach to a request, so the server can recognize a repeat and refuse to do the work twice.
7. A **timeout** is a deadline. Without one, a slow answer becomes a stuck program, and stuck programs pile up until the machine dies.
8. A **circuit breaker** stops calling a service that is clearly broken, so you fail fast instead of queueing up thousands of doomed calls.
9. **Graceful degradation** means giving a worse but useful answer instead of no answer: a slightly old price, a shorter list, a page without the map.
10. Now the one rule to carry out of this section, stated plainly. A client that retries a request which is not idempotent, without an idempotency key, can charge a real person twice for the same purchase.
11. That is not a rare edge case. It is the normal outcome of a lost response plus an automatic retry, and it happens on ordinary evenings.

■ 36.10.2 PLAIN – a picture in your head

1. Picture two buckets, side by side. They are the two standard ways to limit a rate, and they behave differently in a way that matters.
2. The first bucket holds tokens. A tap drips a new token in at a steady speed. The bucket has a maximum size and overflows if it fills.
3. To make a request you must take a token out. No token, no request.
4. If you have been quiet for a while, the bucket is full, and you may fire off a whole burst at once. That is the **token bucket**.
5. The second bucket has a small hole in the bottom. You pour requests in at the top. They drip out of the hole at one fixed speed, no matter how fast you pour.

6. If you pour too fast the bucket fills up and the rest spills on the floor. That is the **leaky bucket**.
7. So the token bucket forgives a burst after a quiet period. The leaky bucket never forgives a burst; it smooths everything into one steady stream.
8. Now the circuit breaker, which is a different picture. It is the fuse in the fuse box of a house.
9. When a wire keeps short-circuiting, the fuse blows and cuts the power. It does not keep pushing current into a fault until the house catches fire.
10. Later, somebody carefully switches it back on to see whether the fault has been fixed. If it fails again, it trips again straight away.

Where this comparison breaks:

11. Real buckets and taps are continuous. Software counts in discrete steps and checks the clock, so a token bucket is arithmetic, not plumbing.
12. A house fuse is reset by a person. A circuit breaker in software resets itself on a timer, which means it can flip back and forth if you choose the timer badly.
13. Water in a bucket is anonymous. Rate limits are almost always counted per caller, so there are millions of tiny buckets, not one big one.

■ 36.10.3 PLAIN – a worked example

1. First, the maths of a herd, because this is the part people do not believe until they see the numbers.
2. Set the scene. Ten thousand clients are talking to one service. The service restarts and is unavailable for 30 seconds.
3. Every client fails at roughly the same instant, because they all failed for the same reason: the service went away.
4. Each client uses plain exponential backoff with a base of one second: wait 1, then 2, then 4, then 8, then 16, then 32 seconds.
5. Because they all started together, they all wake together. Here is when each retry wave lands, measured from the moment of failure.

Attempt	No jitter, all arrive at	Full jitter, spread over
1	$t = 1 \text{ s}$	0 to 1 s
2	$t = 3 \text{ s}$	0 to 2 s
3	$t = 7 \text{ s}$	0 to 4 s
4	$t = 15 \text{ s}$	0 to 8 s
5	$t = 31 \text{ s}$	0 to 16 s
6	$t = 63 \text{ s}$	0 to 32 s

6. Without jitter, all ten thousand requests arrive inside the same few milliseconds. Suppose the service can handle 500 requests per second.
7. Ten thousand requests in, say, 10 milliseconds is an instantaneous rate of one million requests per second. That is 2,000 times capacity.
8. So the wave fails. And because it fails together, the next wave is still synchronized. The herd never breaks up. This is the **thundering herd**.
9. Worse: the service comes back at $t = 30 \text{ s}$, gets hit by the whole herd at $t = 31 \text{ s}$, falls over again, and the outage extends itself.
10. Now add jitter. The rule called full jitter is: wait a random amount between zero and the current backoff window.

11. At attempt 6 the window is 32 seconds wide. Ten thousand clients spread evenly over 32 seconds is 10,000 divided by 32, which is about 312 requests per second.
12. 312 is comfortably under the 500 the service can take. The herd has become a queue, and the service recovers.
13. That is the entire argument for jitter, in one division. Randomness turns a spike into a flow, and it costs one call to a random number generator.
14. Second worked example: the double charge, step by step, with real money.

```
t=0.0s  client POSTs /charges amount 4999 (49.99 in minor units)
t=0.4s  server creates charge ch_A, moves 4999, starts reply
t=0.4s  the reply is lost somewhere on the path back
t=5.0s  client timeout fires. Client sees: "no response"
t=5.0s  client retry logic: "safe to retry, it probably failed"
t=5.4s  server creates charge ch_B, moves 4999 AGAIN
        customer has now paid 99.98 for one order
```

15. Nothing here is a bug in the usual sense. Every component did what it was told. The client could not tell a lost request from a lost response.
16. Now the same story with an idempotency key. The client generates a random identifier once, before the first attempt, and reuses it on every retry.

```
t=0.0s  POST /charges Idempotency-Key: 8f14e45f-ea4f-4c1b-9b4a-0c1d
t=0.4s  server records key 8f14... -> charge ch_A, result 201
t=0.4s  reply lost
t=5.0s  client retries with the SAME key 8f14...
t=5.4s  server finds the key already recorded
t=5.4s  server replays the stored result. No second charge.
t=5.4s  client receives 201 and the SAME charge id ch_A
```

17. Read the last line again. The client gets the original answer, not a new one, and not an error. From the client's view the call simply succeeded.
18. The key must be generated once per logical operation, not once per attempt. A new key on each retry recreates the original bug exactly.

■ 36.10.4 PLAIN – what is really happening inside

1. A token bucket in code is three numbers per caller: how many tokens are left, the refill rate, and the time it was last touched.
2. Nothing runs on a timer. When a request arrives, the server works out how much time has passed and adds that many tokens, capped at the bucket size.
3. Worked with numbers. Bucket size 100 tokens, refill 10 tokens per second. The caller has been quiet for a minute, so the bucket is full at 100.

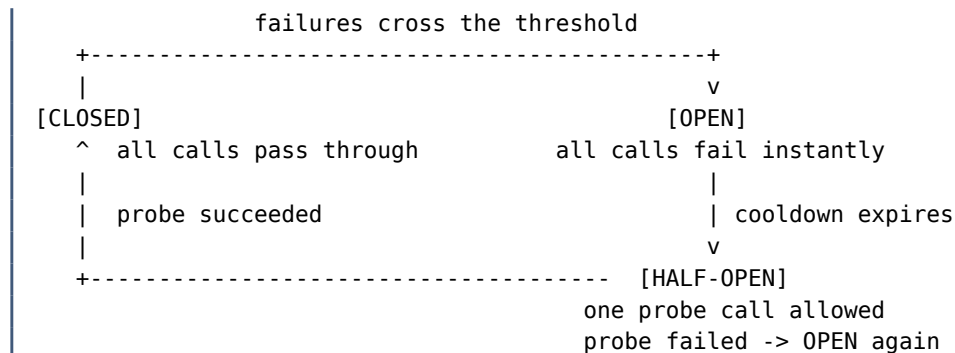
Event	Tokens before	Tokens after
Burst of 100 requests	100	0
Request 101, same instant	0	refused, 429
100 ms later, 1 request	1	0
Steady state	tops up 10/s	allows 10/s

4. So the shape is: a burst of up to 100, then a sustained 10 per second forever. That is usually what you want from a public API.
5. A leaky bucket used as a queue is different. Requests go into a queue of depth 100 and are released at exactly 10 per second.

6. Send 100 at once and none are refused, but the hundredth is not served for ten seconds. The burst is paid for in waiting instead of in refusals.
7. Compare the two directly:

Property	Token bucket	Leaky bucket
Bursts	Allowed up to depth	Smoothed away
Output shape	Bursty, then steady	Constant rate
Excess traffic	Refused immediately	Queued, then dropped
Adds delay	No	Yes, up to depth/rate

8. Which to choose: token bucket when callers are humans or apps that come and go, leaky bucket when you must protect something downstream that cannot take a spike at all, such as a payment processor or a legacy database.
9. When the limit is hit, the server answers 429 Too Many Requests, and should include a Retry-After header saying how long to wait.
10. Retry-After may be a number of seconds, such as Retry-After: 30, or an HTTP date. A client that ignores it and hammers on is the reason services start banning integrations.
11. Now the circuit breaker. It is a small state machine with three states, kept per downstream service.



12. Closed is normal. Calls go through and failures are counted.
13. Open means the breaker has tripped. Calls are not attempted at all. They fail immediately, locally, in microseconds, with no network involved.
14. That instant local failure is the point. It frees the threads, sockets and memory that would otherwise be stuck waiting on a dead service.
15. Half-open is the careful test. After a cooldown, one call is let through. If it works, close the breaker. If not, open it again and wait longer.
16. Timeouts sit under all of this. There is not one timeout; there is a timeout at every layer, and if any layer has none, the whole chain can hang.
17. And graceful degradation is the plan for when the breaker is open. You always need an answer for the question: what do we show the user now?

■ 36.10.5 TECHNICAL – the engineer’s version

1. Status codes and headers, with sources. 429 Too Many Requests comes from RFC 6585, April 2012, and is carried forward in RFC 9110, June 2022. Retry-After is specified in RFC 9110 section 10.2.3 and takes either delta-seconds or an HTTP-date.
2. Use 429 for “you exceeded your quota” and 503 Service Unavailable for “I am overloaded or in maintenance”. Both may carry Retry-After. This distinction is a convention that most large APIs follow, not a rule the specification enforces.

- Standard rate-limit headers are, as of August 2026, still not an RFC. The IETF HTTPAPI working group document `draft-ietf-httpapi-ratelimit-headers` reached revision 11 in May 2026 and defines two structured fields, `RateLimit` and `RateLimit-Policy`, replacing the earlier triple `RateLimit-Limit`, `RateLimit-Remaining` and `RateLimit-Reset`.
- In practice, today, most services send their own `X-RateLimit-*` headers. Treat header names as per-service configuration, not as a standard, and read the documentation for each API you call.
- GitHub’s published REST limits, which are a good calibration point:

Caller	Requests per hour
Unauthenticated	60
Personal access token	5,000
GitHub App installation	5,000 to 12,500
Enterprise Cloud app	15,000

- GitHub returns `x-ratelimit-limit`, `x-ratelimit-remaining`, `x-ratelimit-used`, `x-ratelimit-reset` (UTC epoch seconds) and `x-ratelimit-resource`, and answers with 403 or 429 when you exceed a limit. GitHub also enforces undocumented secondary rate limits on bursts and on expensive operations, which is why blind parallelism gets integrations blocked.
- Backoff formulas. The reference treatment is Marc Brooker’s AWS Architecture Blog post “Exponential Backoff And Jitter”, 4 March 2015. Its three variants, exactly as published:

```

no jitter      : sleep = min(cap, base * 2 ** attempt)
full jitter    : sleep = random(0, min(cap, base * 2 ** attempt))
equal jitter   : temp  = min(cap, base * 2 ** attempt)
                  sleep = temp/2 + random(0, temp/2)
decorrelated  : sleep = min(cap, random(base, sleep * 3))

```

- The post’s simulations found full jitter best overall on total work done and competitive on completion time. Use full jitter unless you have measured a reason not to. Typical parameters: base 100 ms, cap 20 s, 5 attempts.
- Only retry what is retryable. Retry on connection errors, timeouts, 429, 502, 503 and 504. Do not retry 400, 401, 403, 404, 409 or 422; the answer will not change and you are just generating load.
- Idempotency keys. `Idempotency-Key` is described in the IETF draft `draft-ietf-httpapi-idempotency-key-header`, revision 07, October 2025. It is still a draft, not an RFC, but the header name is a de facto industry convention.
- Stripe’s implementation is the reference behaviour, and its documented rules are worth copying exactly:

Rule	Stripe’s behaviour
Key form	V4 UUID suggested, 255 chars max
Storage	Status code and body saved
Retention	Pruned after at least 24 hours
Different params, same key	Request errors

- Note the fourth row. Reusing a key with different parameters is an error, not a silent overwrite. That catches the bug where a client reuses a key for a different purchase.
- Stripe accepts idempotency keys on all POST requests, and states plainly that they have no effect on GET and DELETE because those are already idempotent by definition per RFC 9110.

14. Server-side, the usual implementation is a table keyed on (account, idempotency key) with a unique index, holding a state of in-progress or complete plus the serialized response, written in the same database transaction as the effect.
15. Timeouts, layer by layer. Every one of these has a default, and several common defaults are effectively infinite.

Layer	Setting	Sensible value
DNS lookup	resolver timeout	2 to 5 s
TCP connect	connect timeout	2 to 5 s
TLS handshake	handshake timeout	5 s
Response read	read timeout	10 to 30 s
Whole call	overall deadline	30 s

16. With curl these are `--connect-timeout` and `--max-time`, plus `--retry`, `--retry-delay`, `--retry-max-time` and `--retry-all-errors`. In Go they are fields on `http.Transport` plus a context deadline. In Python's `requests` a bare `timeout=` covers connect and read, and omitting it means wait forever, which is the single most common production mistake in that library.
17. Deadlines should propagate. gRPC does this properly: a deadline set by the caller travels in the `grpc-timeout` header and is reduced at each hop, so a service that is already out of time does not start new work. HTTP has no standard equivalent, which is why hand-rolled chains leak time.
18. Hard platform limits exist and will surprise you. Amazon API Gateway had a fixed 29-second integration timeout for years; in June 2024 AWS allowed that limit to be raised beyond 29 seconds for regional REST APIs, subject to an account-level quota increase. Check the number for your platform rather than assuming.
19. Circuit breakers. The pattern was named and popularized by Michael Nygard in "Release It!", 2007. Netflix's Hystrix was the best-known Java implementation; its last release, 1.5.18, was in November 2018, after which Netflix put it into maintenance mode and pointed new work at `resilience4j`. Envoy, Istio and Linkerd implement the same idea in the proxy layer, outside your code.
20. Practical breaker settings: trip on a rolling error rate rather than a raw count, for example 50 percent failures over a 10-second window with a minimum of 20 requests; cooldown 5 to 30 seconds; one probe in half-open.
21. Related and often confused: a **bulkhead** limits how many concurrent calls one dependency may consume, so a slow dependency cannot eat the whole thread pool. Breakers stop bad calls; bulkheads contain them.
22. Load shedding is the server-side twin of the breaker. Under overload, reject early and cheaply with 429 or 503 rather than accepting work you cannot finish. Accepting everything and timing out is the worst option because you pay the cost and deliver nothing.
23. Graceful degradation in concrete terms: serve a stale cached value and mark it stale, drop optional page sections, fall back to a smaller model or a simpler ranking, queue writes for later, or return partial results with a flag saying what is missing.
24. HTTP even has a header for the stale case: `Cache-Control: stale-if-error` from RFC 5861, May 2010, lets a cache serve a stale response when the origin errors.
25. Observation and testing: `curl -w` prints timing breakdowns such as `time_connect` and `time_starttransfer`; `ab`, `wrk`, `hey` and `k6` generate load; `tc netem` and `Toxiproxy` inject latency and loss; and chaos engineering, as practised by Netflix's Chaos Monkey from 2011, tests these paths on purpose rather than by accident.

■ 36.10.6 WORDS – remember these

1. Rate limit — a cap on how often you may ask — a server-enforced quota per caller per window, signalled with 429 and `Retry-After`.
2. Token bucket — a jar of tickets that refills steadily — a limiter holding capacity *C* refilled at rate *R*, permitting bursts up to *C*.
3. Leaky bucket — a bucket with a hole, output always steady — a limiter that queues arrivals and releases them at a fixed rate, smoothing bursts.
4. Exponential backoff — wait longer after each failure — retry delay growing as base times two to the power of the attempt number, capped.
5. Jitter — randomness added to the wait — a randomized delay that de-synchronizes retrying clients and prevents a thundering herd.
6. Thundering herd — everybody retries at the same instant — synchronized client retries producing a load spike far above steady-state capacity.
7. Idempotency key — a reference number on the request — a client-generated unique value letting the server deduplicate retries and replay the result.
8. Circuit breaker — a fuse for a failing dependency — a three-state machine, closed, open and half-open, that fails fast while a service is unhealthy.
9. Bulkhead — a limit on how much one dependency may consume — a bounded concurrency pool per downstream, containing the blast radius of slowness.
10. Graceful degradation — a worse answer beats no answer — deliberately reduced functionality preserving the core path during partial failure.

36.11 Webhooks: the inversion where the server calls you

■ 36.11.1 PLAIN – in simple words

1. Everything so far had the same shape. You call the server. The server answers. You are the one who starts the conversation.
2. A **webhook** turns that around. You give the other service a web address of yours, and when something happens, it calls you.
3. So your program stops being only a client. For webhooks it must also be a server, sitting there with a door open, waiting.
4. The name comes from “web” plus “hook”, where a hook is an old programming word for a place you can attach your own code so it runs when an event occurs.
5. The reason webhooks exist is simple: without them you have to keep asking “has it happened yet? has it happened yet?”, which is called **polling**.
6. Polling wastes almost all of its calls, and it is always a little late, because you only find out at the next check.
7. There is a price for the inversion. Anyone on the internet can send a POST to your address, so you must prove that a message really came from the sender you expect.
8. You do that with a **signature**: the sender computes a code from the message and a shared secret, and you compute the same code and compare.
9. And because the network can lose things, senders resend. So the same event can arrive twice, or three times, or out of order. Your handler must cope.
10. Finally, the practical wall every beginner hits: a laptop at home has no address the internet can reach, so nothing can call you at all. Section 36.11.4 explains why and section 36.11.5 gives the fixes.

■ 36.11.2 PLAIN – a picture in your head

1. Polling is standing at the window every minute, all day, watching for the postman.
2. You look 480 times during the working day. The post arrives once. You looked 479 times for nothing, and you found out up to a minute late.
3. A webhook is a doorbell. You install it once, you give the wire an address, and you get on with your life. It rings when there is something to know.
4. The signature is the courier saying a password you agreed in advance. Anyone can press a doorbell. Only the real courier knows the word.
5. The password is not simply spoken, though. It is mixed with the parcel itself, so the word only matches if the parcel is unchanged.
6. And the courier is stubborn. If nobody answers, they come back in a minute, then in five, then in an hour. So the same parcel can be delivered twice if you took it and then failed to sign for it.

Where this comparison breaks:

7. A doorbell rings in your house, where you already are. A webhook has to reach a machine that in most cases has no public address at all. That is the difference the whole of section 36.11.4 is about.
8. A courier delivers parcels in the order they were posted, near enough. A webhook sender makes no such promise, and several large ones explicitly say so in their documentation.
9. A doorbell that rings twice is obvious to a human. Two identical webhook deliveries look identical to a program, which is why you need event identifiers.

■ 36.11.3 PLAIN – a worked example

1. Compare the two designs with numbers. A shop gets 5 orders a day and wants to react when a payment succeeds.
2. Polling every 60 seconds: 60 times 24, which is 1,440 requests a day. Five of them find something. 1,435 are wasted, and news is up to 60 seconds old.
3. Webhooks: 5 deliveries a day, each arriving about a second after the event. That is 288 times fewer requests and 60 times fresher.
4. Polling every second to fix the delay makes it 86,400 requests a day for the same 5 pieces of news, and burns your rate limit for nothing.
5. Now the delivery itself. Here is a real signed webhook, in the style Stripe uses. The values below were computed for this book and are genuine.

```
POST /hooks/payments HTTP/1.1
Host: shop.example
Content-Type: application/json
Stripe-Signature: t=1767225600,
  v1=9b76fe28839a1536d4ba22b9641546bd14ad4a938f8fa3cfa3f
  e34d35633eb8
Content-Length: 57

{"id":"evt_9f2","type":"payment.succeeded","amount":4999}
```

6. The signature line is one line in reality; it is wrapped here to fit the page. `t` is the time the sender signed it. `v1` names the scheme.
7. To verify, you build the signed string by joining the timestamp, a full stop, and the exact body bytes:

```
1767225600.{ "id": "evt_9f2", "type": "payment.succeeded",
"amount": 4999 }
```

8. Then you compute HMAC-SHA256 over that string using the endpoint secret, here `wh-sec_kedbyte_demo`, and compare with the `v1` value. They match.
9. Now watch what happens if your web framework parses the JSON and your code re-serializes it before checking. Same data, spaces after the colons:

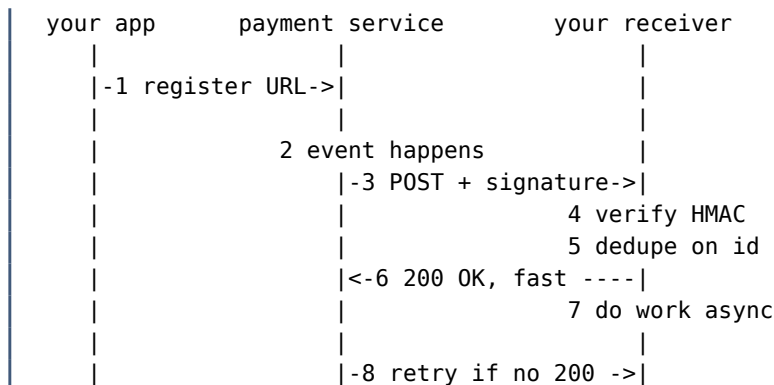
```
body as sent : {"id":"evt_9f2","type":"payment.succe...
signature    : 9b76fe28839a1536d4ba22b9641546bd14ad4d...

body re-made : {"id": "evt_9f2", "type": "payment.suc...
signature     : bd01c7e20696abf97e590d6b0e86cae26157bf...
```

10. Every character of the signature is different. Nothing about the meaning changed. Four space characters changed the bytes, and HMAC signs bytes.
11. This is the single most common webhook bug in the world, and now you will never write it: verify against the **raw body**, before any parsing.
12. Last piece: the timestamp is inside the signed string. So an attacker who captures a valid delivery and replays it tomorrow cannot change `t` without breaking the signature, and your tolerance check rejects the old one.

■ 36.11.4 PLAIN - what is really happening inside

1. A webhook delivery is nothing exotic. It is an ordinary HTTP POST, made by the sender's servers, to a URL you registered with them earlier.
2. The sequence, end to end:



3. Step 4 must come before anything else. An unverified webhook is an instruction from a stranger, and people have been robbed by acting on one.
4. Step 5 exists because delivery is at-least-once. Every serious sender attaches a unique event identifier, and your job is to remember the ones you have already handled and ignore repeats.
5. Step 6 must be fast. Senders time out. If your handler does ten seconds of database work before replying, you will be marked as failed and resent the same event, which multiplies your load exactly when you are slow.
6. So the pattern is: verify, deduplicate, write the event to a queue or table, return 200, and do the real work afterwards in the background.
7. Ordering. Events are generated in one order and delivered in whatever order the network and the retry queue produce. Stripe states in its documentation that it does not guarantee delivery order.
8. The fix for ordering is not to sort them. It is to stop depending on order: treat each event as a hint that something changed, and read the current state back from the API before acting on anything important.
9. Now the wall. Why can a webhook not reach your laptop?

10. Your machine at home has a private address, something like 192.168.0.14, handed out by your router. Chapter 24 covered these: the ranges 10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16 from RFC 1918 are not routable on the public internet.
11. Your router performs NAT, network address translation. When you make an outbound connection it writes down a mapping and rewrites the addresses, so replies find their way back to you.
12. The mapping is created by your outbound packet. Nothing creates a mapping for an inbound connection from a stranger. So a POST from a payment company arrives at your router with no idea where to send it, and is dropped.
13. The reader's own network makes this worse in a way worth seeing. Their traceroute showed the ISP's own core using private addresses, 172.31.0.17 and 172.26.22.235 among them, before reaching a public address at 137.97.29.249.
14. That pattern is carrier-grade NAT: the reader's home router is itself behind another layer of translation run by the ISP. Even opening a port on the home router would not help, because the public address is not theirs to control.
15. So the answer is not to accept an inbound connection at all. The answer is to make an outbound connection from inside, which NAT allows, and let the inbound requests travel back down it. That is a tunnel, and Chapter 34 covered exactly this mechanism for VPNs and proxies.

■ 36.11.5 TECHNICAL – the engineer's version

1. History and standardization. Jeff Lindsay coined the term “web hook” in a blog post on 3 May 2007. There was no specification for twenty years. On 13 December 2023 the Standard Webhooks project published an open specification, with a steering committee drawn from Zapier, Twilio, Lob, Mux, ngrok, Supabase and Kong. It is a convention with momentum, not an RFC.
2. Header conventions in the wild differ, and there is no winner:

Sender	Signature header	Algorithm
GitHub	X-Hub-Signature-256	HMAC-SHA256, hex
Stripe	Stripe-Signature	HMAC-SHA256, hex
Standard Webhooks	webhook-signature	HMAC-SHA256, base64

3. GitHub signs the raw body alone and prefixes the digest with sha256=. Stripe signs timestamp + "." + raw body. Standard Webhooks signs id + "." + timestamp + "." + raw body. Read the sender's documentation; guessing the signed string is the second most common webhook bug.
4. GitHub also sends X-GitHub-Event naming the event type, X-GitHub-Delivery carrying a GUID that identifies the delivery, and X-GitHub-Hook-ID identifying the webhook itself.
5. Verification checklist, in order:
 1. Read the raw request body as bytes. Disable any middleware that parses, re-encodes, decompresses or pretty-prints it first.
 2. Recompute the HMAC with the endpoint secret.
 3. Compare with a constant-time comparison, such as `hmac.compare_digest` in Python or `crypto.timingSafeEqual` in Node, never with `==`.
 4. Check the timestamp is recent. Stripe's libraries default to a 5-minute tolerance and warn against setting it to zero, which disables the check.
 5. Only then parse the JSON.
6. Constant-time comparison matters because a naive byte-by-byte comparison returns faster on an early mismatch, and that timing difference is enough to recover a signature one byte at a time over many attempts.

7. Secret rotation: senders usually support two active secrets during a change window. Stripe allows a rolled secret's predecessor to stay valid for up to 24 hours and sends one signature per active secret in the same header.
8. Retry behaviour from the sender's side, with real published policies. Stripe retries deliveries for up to three days in live mode with exponential backoff, and allows manual resends for up to 15 days from the Dashboard and 30 days from the CLI. Other senders differ enormously; some do not retry automatically at all and only offer manual redelivery. Never assume.
9. Because retries exist, deduplicate on the sender's event identifier. Stripe recommends recording processed event IDs, and warns that in some cases two distinct Event objects are generated for the same underlying change, so deduplicating on `data.object.identity` plus `event.type` is sometimes needed as well.
10. Idempotent handlers are the real defence. Design the handler so that applying the same event twice leaves the same state. Then duplicates are free rather than dangerous, and you are back to the exactly-once effect described in section 36.5.
11. Return 2xx immediately, then process asynchronously. Stripe treats any 3xx as a failure, so a webhook URL that redirects, for example from a bare domain to `www`, silently fails every delivery.
12. Additional hardening: allowlist the sender's published IP ranges as a second layer; require HTTPS with TLS 1.2 or 1.3; exempt the webhook route from CSRF protection, which otherwise rejects every legitimate delivery; and never trust amounts or ownership from the payload alone for anything financial, since the correct move is to re-read the object from the API.
13. Receiving on a machine behind NAT. The options, honestly compared:

Approach	Works behind CGNAT	Cost of setup
Public server or function	Yes	Deploy something
Router port forward + DDNS	No	Router config
Reverse tunnel service	Yes	One command
Cloud queue relay	Yes	Extra service

14. Port forwarding fails under carrier-grade NAT for the reason Chapter 24 gave: the public address belongs to the ISP and is shared, so there is no port on it that is yours to map.
15. Reverse tunnels work because the connection is established outbound from inside the network, which NAT permits, and inbound requests are then multiplexed back down that already-open connection.
16. The common tools: ngrok, first released by Alan Shreve in 2013 and incorporated in 2015; Cloudflare Tunnel, launched as Argo Tunnel in 2018 and made free for everyone on 15 April 2021 with the `cloudflared` daemon; Tailscale Funnel; and plain `ssh -R` to any host you already own, which needs GatewayPorts enabled on that host.
17. Vendor CLIs do the same thing without a public URL at all. `stripe listen --forward-to localhost:4242/webhook` opens an outbound connection to Stripe and replays events to your local port, printing a signing secret for the session. `gh webhook forward` does the equivalent for GitHub.
18. Testing tools: `stripe trigger payment_intent.succeeded` fires a real test event; request-bin style services capture and display raw deliveries; and the sender's own delivery log, which shows the exact request and your exact response, is almost always the fastest way to find the fault.
19. Webhooks versus the alternatives, in one table:

Mechanism	Direction	Best for
Polling	Client asks	Simple, low event rate
Webhook	Server calls you	Rare events, any client
WebSocket	Both, held open	Live two-way, chat
Server-sent events	Server pushes	One-way live feed
Message queue	Broker in middle	High volume, ordering

20. At scale, webhooks become a queue problem, and teams move to a broker such as Amazon EventBridge, Google Pub/Sub or Kafka. Stripe supports sending events directly to EventBridge or Azure Event Grid instead of to an HTTP endpoint, which removes your receiver from the critical path entirely.

■ 36.11.6 WORDS – remember these

1. Webhook — the other service calls you when something happens — a user-registered HTTP call-back delivering event notifications by POST.
2. Polling — asking repeatedly whether anything changed — client-initiated periodic requests, costly and bounded in freshness by the interval.
3. HMAC — a code proving a message came from someone with the secret — keyed-hash message authentication code, RFC 2104, usually with SHA-256.
4. Raw body — the exact bytes that arrived — the unparsed request payload, the only correct input to signature verification.
5. Replay attack — resending a captured valid message — defended by signing a timestamp and rejecting deliveries outside a tolerance window.
6. Delivery identifier — the sender's unique name for this event — an event or delivery ID used by the receiver to detect and drop duplicates.
7. Reverse tunnel — an outbound connection used to carry inbound requests — a NAT-traversal technique multiplexing public traffic to a private host.
8. Carrier-grade NAT — the ISP translates too, not just your router — RFC 6598 shared address space, making inbound port mapping impossible for the user.

36.12 Designing an API people can actually use

■ 36.12.1 PLAIN – in simple words

1. A usable API is mostly boring. It is boring in the same way, everywhere, and that sameness is the feature.
2. **Naming.** Use nouns for things and let the method say the verb. A path called `/orders/8812` with a `DELETE` is better than `/deleteOrder?id=8812`.
3. **Consistency.** If one part of your API uses `created_at` and another uses `creationTime`, every user of your API will get it wrong forever.
4. **Errors with codes.** Send back a short machine-readable code such as `insufficient_funds`, alongside the human sentence.
5. That code matters far more than the sentence, and here is why: sentences get reworded, translated and improved, and every rewording silently breaks any program that was matching on the text.
6. A code is part of the contract. A message is not. If you promise nothing about the message, you may fix its typos without breaking anyone.

7. **Versioning.** There are four honest answers to “how do we change things without breaking people”, and each costs something. Section 36.12.5 compares them without pretending one is obviously right.
8. **Deprecation.** Say a thing is going away, say when, say what to use instead, and then actually keep to the date.
9. **Documentation.** Reference pages tell you what exists. Guides tell you how to do the common job. You need both, and you need examples that run.
10. **A machine-readable description.** If your API is described in a file a computer can read, then documentation, client libraries, test doubles and input checking can all be generated from that one file.
11. **A sandbox.** Somewhere users can try everything with fake money and fake data, and get predictable answers, without fear.

■ 36.12.2 PLAIN – a picture in your head

1. Think of a large hardware shop with many aisles.
2. A good shop puts screws in one place, and every box is labelled the same way: size, then length, then material, then quantity.
3. Once you have read one label you can read all of them. You never have to learn the shop twice.
4. A bad shop labels some boxes by length first and some by material first, and keeps a few screws in the paint aisle for historical reasons.
5. The stock is identical in both shops. One is pleasant, one is exhausting. That difference is entirely consistency, and it is the same for APIs.
6. Now the error codes. Imagine the shop’s till printing “Card refused” on one receipt and “Your card has been declined” on another for the same fault.
7. A person shrugs. A machine that was told to look for the exact words “Card refused” now silently does the wrong thing.
8. Put a number on the receipt as well, always the same number for the same fault, and the machine can be right regardless of the wording.

Where this comparison breaks:

9. A shop can rearrange the aisles overnight and customers adapt in a week. An API cannot, because its customers are programs that do not adapt at all.
10. A shop has one physical layout. An API often has to serve the old layout and the new layout at the same time, for years. That is versioning, and shops have no equivalent of it.

■ 36.12.3 PLAIN – a worked example

1. Here is one endpoint designed badly, then the same endpoint designed well. Nothing about the underlying system changes.

```
BAD
GET /getOrderData?orderid=8812&fmt=json&full=1
200 OK
{"ok":0,"err":"Order not found or you dont have access"}

GOOD
GET /v1/orders/ord_8812
404 Not Found
Content-Type: application/problem+json
{"type":"/errors/not-found","code":"order_not_found", ...}
```

2. Five faults in the bad version, one at a time.
3. The path contains a verb, `getOrderData`, which duplicates what GET already says, and it will sit awkwardly beside `updateOrderData` later.
4. The identifier is in a query string, so `/getOrderData` with no arguments is a different thing again, and caches and logs treat them inconsistently.
5. The status is 200, meaning success, for a failure. Every client library, proxy and monitoring dashboard in the world will record this as fine.
6. The error is a sentence with a spelling mistake. Fix the spelling and you break anybody who matched on the string. So the mistake becomes permanent.
7. The message merges two different situations, “no such order” and “not yours”, so the caller cannot tell them apart even if it wanted to.
8. Now the good version in full. This is what the body should look like, using the standard problem format:

```
HTTP/1.1 422 Unprocessable Content
Content-Type: application/problem+json

{
  "type": "/errors/insufficient-funds",
  "title": "Insufficient funds",
  "status": 422,
  "detail": "Balance 1200 is below the amount 4999.",
  "instance": "/orders/ord_8812",
  "code": "insufficient_funds",
  "balance_minor": 1200,
  "required_minor": 4999
}
```

9. `code` is the stable machine-readable name. `title` and `detail` are for humans and may be reworded at any time. `instance` says which thing failed.
10. The extra fields `balance_minor` and `required_minor` let a client build a useful message itself, in its own language, without parsing English.
11. Notice the money. It is an integer in minor units, 4999 meaning 49.99, not a decimal number. Section 36.12.5 explains why floats and money do not mix.
12. Now the same discipline applied to field naming across a whole API:

Decision	Pick one	Never mix
Field case	snake_case	with camelCase
Timestamps	RFC 3339 UTC	with epoch seconds
Identifiers	prefixed strings	with bare integers
Money	integer minor units	with decimals
Lists	always paginated	with sometimes-all

13. Prefixed identifiers such as `ord_8812` and `cus_41a` are a small idea with a large payoff: a support engineer can tell what a value is by looking at it, and a client cannot accidentally pass a customer where an order goes.

■ 36.12.4 PLAIN – what is really happening inside

1. Versioning exists because of one asymmetry. You can deploy a change in a minute. Your callers may take three years to update, and some never will.

2. So the real question is never “what is the tidiest version scheme”. It is “how long am I willing to run the old behaviour, and who pays for it”.
3. There are four families of answer, and a fifth position that says do not version at all.
4. Put the version in the path, as `/v1/orders`. Simple, visible, easy to route in a proxy, easy to explain. The cost is that the version is attached to every URL, so identifiers are not stable across versions and a client that upgrades must rewrite every path it holds.
5. Put it in a header, as `API-Version: 2` or `Stripe-Version: 2026-07-29`. URLs stay stable and the change is invisible in logs and links, which is both the benefit and the drawback: it is easy to forget you set it.
6. Put it in the media type, as `Accept: application/vnd.kedbyte.v2+json`. This is what content negotiation in HTTP was actually designed for, and it is the most correct answer. It is also the one nobody finds friendly, and it breaks casual testing in a browser.
7. Use dates instead of numbers, so each account is pinned to the behaviour of the day it joined. Very kind to callers, very expensive to run, because the server keeps every old shape alive behind translation layers.
8. Or do not version at all. Only make additive changes: new fields may appear, nothing is ever removed or given a new meaning. This is what GraphQL encourages and what many internal APIs do. It works until the day you genuinely need to remove something, and then it does not.
9. Whichever you choose, the deprecation mechanics are the same. Announce with a date. Send a header on every response to the old thing. Publish what to use instead. Then remove it on the day you said.
10. A useful middle step is the **brownout**: on announced days before the removal, turn the old endpoint off for an hour, so that anybody still calling it notices while there is still time.
11. Now the machine-readable description. A file such as an OpenAPI document lists every path, every parameter, every response shape and every error.
12. Once that file exists, tools can read it and produce: reference documentation, client libraries in a dozen languages, a mock server that returns example data, request validation at the gateway, and a diff between two versions that tells you whether a change is breaking.
13. The important consequence is subtle. The specification stops being a document that describes the code and starts being a thing the code is checked against. Drift becomes a test failure instead of a support ticket.

■ 36.12.5 TECHNICAL – the engineer’s version

1. Error format. RFC 9457, Problem Details for HTTP APIs, published July 2023, obsoletes RFC 7807 from March 2016. Media type `application/problem+json`. Members: `type` (a URI reference identifying the problem type), `title`, `status`, `detail`, `instance`, plus any extension members you add.
2. RFC 9457 is explicit that `detail` is for humans and that clients should key off `type`. Add your own short code extension if you want a compact stable token; that is a convention, not part of the RFC.
3. Do not put secrets, stack traces or internal hostnames in `detail`. Error bodies are logged, screenshotted and pasted into public issue trackers.
4. Versioning strategies, compared honestly:

Strategy	Example	Main cost
URL path	<code>/v1/orders</code>	URLs churn on upgrade
Header	<code>API-Version: 2</code>	Invisible, easy to forget
Media type	<code>vnd.x.v2+json</code>	Correct but unfriendly

Strategy	Example	Main cost
Date-pinned	2026-07-29	Server keeps every shape
None, additive	no marker	Cannot ever remove

5. Real practice, with dates. Stripe pins each account to a dated version and names major releases; as of this writing the current version string is 2026-07-29.dahlia, sent in the Stripe-Version header, with monthly releases guaranteed backward compatible and named releases such as Aca-cia and Basil marking the breaking ones. GitHub uses a dated X-GitHub-API-Version header. Kubernetes uses group-and-version in the path, such as apps/v1, with an explicit alpha, beta, stable ladder.
6. Experts genuinely disagree here. Roy Fielding has argued publicly that versioning a REST API in the URL is an admission that hypermedia was not used and that clients were coupled to URL structure. The pragmatic camp replies that /v1/ is legible, greppable and routable, and that ship-ping beats purity. Both are right about different things.
7. What counts as a breaking change is the part teams fail to write down. A workable rule: adding an optional field, adding a new endpoint, and adding a new enum value to an output are non-breaking. Removing or renaming a field, adding a required input, narrowing a type, changing a status code, and changing the meaning of an existing value are breaking.
8. That third one bites. Adding a new enum value is only safe if you told clients in advance to tolerate unknown values. Say it in the documentation on day one, because you cannot add the rule retroactively.
9. Deprecation headers. RFC 9745, The Deprecation HTTP Response Header Field, published 17 March 2025, defines Deprecation as a structured field date. RFC 8594, May 2019, defines Sunset as the date the resource stops responding. RFC 9745 requires that Sunset not be earlier than Deprecation.


```
HTTP/1.1 200 OK
Deprecation: @1780272000
Sunset: Wed, 30 Sep 2026 23:59:59 GMT
Link: </docs/migrate-v2>; rel="deprecation"; type="text/html"
```
10. The @ form is the structured-field syntax for a Unix timestamp; here it is 1 June 2026. Link with rel="deprecation" points at the migration guide, using web linking from RFC 8288, October 2017.
11. A humane deprecation policy in five parts: a minimum notice period, stated publicly, of 6 to 12 months for a paid public API; response headers from day one of the notice; direct email to accounts still calling it; a migration guide with before-and-after examples; and scheduled brownouts.
12. OpenAPI. History: Swagger began in 2010, written by Tony Tam at Wordnik; SmartBear donated the specification to the OpenAPI Initiative under the Linux Foundation in November 2015; it was renamed the OpenAPI Specification on 1 January 2016. Version 3.0.0 was released 26 July 2017, 3.1.0 on 18 February 2021, and 3.2.0 on 23 September 2025. Version 3.1 matters most, because it made the schema language a proper superset of JSON Schema 2020-12.
13. A minimal but real OpenAPI document:

```
openapi: 3.1.0
info:
  title: KedByte Orders API
  version: 1.4.0
paths:
  /orders/{orderId}:
```

```

get:
  operationId: getOrder
  summary: Fetch one order
  parameters:
    - name: orderId
      in: path
      required: true
      schema: { type: string, pattern: "^ord_[a-z0-9]+$" }
  responses:
    "200":
      description: The order
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/Order"
    "404":
      description: No such order
      content:
        application/problem+json:
          schema:
            $ref: "#/components/schemas/Problem"
  components:
    schemas:
      Order:
        type: object
        required: [id, status, amount_minor, currency]
        properties:
          id: { type: string }
          status:
            type: string
            enum: [pending, paid, shipped, cancelled]
          amount_minor: { type: integer, minimum: 0 }
          currency: { type: string, minLength: 3, maxLength: 3 }
          created_at: { type: string, format: date-time }

```

14. What that file buys you, concretely: `swagger-ui` and `redoc` render documentation; `openapi-generator` and `oapi-codegen` emit clients and server stubs; `prism` runs a mock server; `schemathesis` and `dredd` test the live API against the document; `oasdiff` reports breaking changes between two revisions; and gateways including Kong, Envoy and Amazon API Gateway can validate requests before they reach your code.
15. Equivalents in other ecosystems: `.proto` files for gRPC, the GraphQL SDL for GraphQL, WSDL for SOAP, and AsyncAPI for event-driven and webhook interfaces. OpenAPI 3.1 also gained a top-level webhooks section, so outbound callbacks can be described in the same document.
16. Generated SDKs, honestly. They give type safety, retries, pagination helpers, authentication and release-note churn for free, and they are how Stripe and Twilio made their APIs pleasant. The costs are real: generated code is often unidiomatic, one bad generator template produces the same bug in twelve languages at once, and every extra language is an extra thing to support forever. Many teams generate a thin core and hand-write the ergonomic layer on top.
17. Sandboxes. The requirements that matter: separate credentials that cannot touch production, deterministic magic values that force specific outcomes, no real money or messages, and the same version and error behaviour as production. Stripe's test card numbers are the canonical example, where a specific card number always produces a specific decline code.
18. Money and time, since they cause more API bugs than anything else. Money: integer minor units plus an ISO 4217 currency code, never a binary floating-point number, because 0.1 plus 0.2 is not

0.3 in IEEE 754. Time: RFC 3339 timestamps in UTC with an explicit offset, and a separate IANA time zone identifier such as Asia/Kolkata when local meaning matters. Never a bare local time.

19. Documentation that works has four layers: a five-minute quickstart that produces a real result, task guides for the common jobs, generated reference for every field, and a changelog with dates. The quickstart is the one that decides whether a developer stays.

■ 36.12.6 WORDS – remember these

1. Error code — a short stable name for a failure — a machine-readable token that is part of the contract, unlike the human-readable message.
2. Problem Details — the standard shape for an error body — RFC 9457, media type `application/problem+json`, with type, title and status.
3. Breaking change — existing callers stop working — any change that invalidates a promise the previous version made, whatever the version number.
4. Deprecation — announced, still working — RFC 9745 Deprecation header marking a resource as scheduled for removal, with a migration path.
5. Sunset — the date it stops answering — RFC 8594 Sunset header giving the time after which a resource becomes unresponsive.
6. Brownout — a rehearsal for the switch-off — a deliberate short outage of a deprecated endpoint to surface remaining callers before removal.
7. OpenAPI — a description of the API a machine can read — the OpenAPI Specification, 3.1.0 from 2021 and 3.2.0 from 2025, formerly Swagger.
8. SDK — a ready-made client library — a generated or hand-written package wrapping transport, authentication, retries and pagination for one language.
9. Sandbox — a safe copy to experiment in — an isolated environment with test credentials, deterministic fixtures and no real-world side effects.

36.13 Consuming an API well

■ 36.13.1 PLAIN – in simple words

1. Most of your working life you will be the caller, not the builder. Being a good caller is a skill, and it is mostly discipline.
2. **Read the documentation first.** Not the tutorial. Find four things: the base address, how to authenticate, the rate limits, and how lists are paged.
3. **Try it by hand before writing code.** One `curl` command tells you more than an hour of guessing, because it shows you the real bytes.
4. **Check the status code.** The most common bug in code that calls APIs is assuming it worked and reading a field out of an error body.
5. **Respect the rate limit.** The service is telling you, in a header, how much you have left. Read it. Slow down before you are refused.
6. **Do not fetch what has not changed.** If the server gave you a label for the version you already have, send it back, and it will tell you “no change” with an empty body.
7. **Page properly.** Lists are not lists. They are the first page of a list, and there is a documented way to ask for the next one. Use it, in a loop, with a stopping condition.
8. **Never put a secret in anything the user can see.** A key in a web page, a phone app, or a browser extension is a public key, no matter what you named the variable.
9. That last one is not a style preference. It is the reason a large number of real breaches happened, and the fix is always the same: keep the secret on a server you control and let the user’s device

talk to that.

■ 36.13.2 PLAIN – a picture in your head

1. Calling somebody else's API is like being a guest researcher in a large library that belongs to another institution.
2. You get a reader's card. It is yours, it has your name on it, and lending it to a friend gets you both banned. That is your token.
3. The library allows twenty book requests an hour. A sign at the desk tells you how many you have left today. That sign is the rate-limit header.
4. Books come from the stacks a trolley at a time, twenty per trolley, with a slip saying "more available". That slip is the pagination link.
5. If you already copied a chapter, the librarian will happily tell you "that chapter has not been revised since you last saw it" instead of fetching the whole volume again. That is the conditional request.
6. And when the librarian says no, they say why: no such book, not allowed, closed for lunch, or you have asked too often. Four different answers that need four different reactions from you.

Where this comparison breaks:

7. A librarian notices a rude guest and speaks to them. An API notices nothing and simply blocks your account, often without warning and often at 3 a.m.
8. A human researcher naturally slows down when tired. A for loop does not, so the politeness must be written into the code deliberately.

■ 36.13.3 PLAIN – a worked example

1. A full session against a real public API, using the GitHub REST API. The header names and limits below are exactly as GitHub documents them.
2. Step one, the plain call, with `-i` so we see the response headers as well as the body:

```
curl -i \
  -H "Accept: application/vnd.github+json" \
  -H "X-GitHub-API-Version: 2022-11-28" \
  https://api.github.com/repos/kedbyte/book
```

3. Step two, read the top of the answer before reading any of the body:

```
HTTP/2 200
content-type: application/json; charset=utf-8
etag: W/"7f2c1a9b6e0d4c8f1a2b3c4d5e6f7a8b"
x-ratelimit-limit: 60
x-ratelimit-remaining: 57
x-ratelimit-used: 3
x-ratelimit-reset: 1767229200
x-ratelimit-resource: core
```

4. Four facts, immediately. The call worked. There is an etag label for this version. We have 57 of 60 calls left this hour. The window resets at the epoch second 1767229200, which is 1 January 2026 at 01:00 UTC.
5. The limit is 60 because we sent no token. With a personal access token the same headers would say 5,000. That single header is why anonymous scripts fail at scale and authenticated ones do not.
6. Step three, ask again politely, sending the label back:

```
curl -i -H "If-None-Match: \
W/"7f2c1a9b6e0d4c8f1a2b3c4d5e6f7a8b\""" \
https://api.github.com/repos/kedbyte/book
```

7. If nothing changed, the answer is:

```
HTTP/2 304
etag: W/"7f2c1a9b6e0d4c8f1a2b3c4d5e6f7a8b"
x-ratelimit-remaining: 57
```

8. Read the last line carefully. `x-ratelimit-remaining` did not go down. GitHub documents that a conditional request returning 304 does not count against your primary rate limit when the request was correctly authorized.
9. So conditional requests are not merely a bandwidth saving. On this API they are free calls. A poller that ignores ETags burns its quota on data it already has.
10. Step four, handling the failures rather than assuming success. Here is the shape every calling script should have:

```
code=$(curl -s -o body.json -w '%{http_code}' \
--connect-timeout 5 --max-time 30 \
-H "Authorization: Bearer $GH_TOKEN" \
https://api.github.com/repos/kedbyte/book)

case "$code" in
  200) jq -r '.full_name, .stargazers_count' body.json ;;
  304) echo "unchanged, using cached copy" ;;
  401) echo "token missing or invalid"; exit 1 ;;
  403|429) echo "rate limited; see reset header"; exit 2 ;;
  404) echo "no such repo, or token cannot see it"; exit 3 ;;
  5*) echo "server side; retry with backoff" ; exit 4 ;;
  *) echo "unexpected status $code"; exit 5 ;;
esac
```

11. Note 404. On GitHub a private repository you cannot see returns 404, not 403, on purpose, so that the API does not leak the existence of private things. Your handler must not treat 404 as “definitely does not exist”.
12. Step five, pagination in a loop. GitHub returns a Link header holding the next page’s address, and you follow it until there is no next.

```
url="https://api.github.com/repos/kedbyte/book/issues?per_page=100"
while [ -n "$url" ]; do
  curl -sS -D head.txt -o page.json \
  -H "Authorization: Bearer $GH_TOKEN" "$url"
  jq -r '.[ ] | "\(.number) \(.title)"' page.json
  url=$(tr -d '\r' < head.txt | grep -i '^link:' \
    | tr ',' '\n' | grep 'rel="next"' \
    | sed -e 's/.*<(.*)>.*\/1/' )
  sleep 1
done
```

13. Three things make that loop safe. It follows the server’s own link instead of building page numbers itself. It stops when there is no next link. And it pauses a second between pages, which is politeness in one word.
14. The same call in `httpie`, which is friendlier to type and colours the output, though the plumbing is identical:

```
https://api.github.com/repos/kedbyte/book \
Accept: application/vnd.github+json \
Authorization: "Bearer $GH_TOKEN"
```

15. And the secret. Look at the two examples above: the token came from an environment variable, `$GH_TOKEN`, never from a literal string in the file.
16. That habit is what stops a token being committed to git, pasted into a bug report, or shipped inside a web page.

■ 36.13.4 PLAIN – what is really happening inside

1. A conditional request is a small conversation about versions.
2. On the first response the server sends an ETag, which is just an opaque label for the exact bytes it sent. You store it beside the data.
3. Next time you send If-None-Match with that label. The server compares. If the current label matches, it stops right there and sends 304 Not Modified with a status line, some headers, and no body at all.
4. The saving is the body, which is usually 99 percent of the bytes, plus, on some APIs, the rate-limit charge.
5. There is a second form, If-Modified-Since with a date, which is older and less precise because it only has one-second resolution. Prefer ETags.
6. The same machinery, used the other way round, prevents lost updates. You send If-Match with the ETag on a write, and the server refuses with 412 Precondition Failed if somebody else changed the thing meanwhile.
7. Pagination has two families, and knowing which you are in matters.
8. Offset paging asks for “rows 200 to 299”. It is simple and it is wrong under change: if a row is inserted at the top while you are paging, everything shifts down by one and you see one item twice and miss another.
9. Cursor paging asks for “carry on after this marker”. The marker encodes the position in a stable sort order, so insertions elsewhere do not disturb you.
10. This is why big APIs moved to cursors, and why the Link header pattern hands you a complete next address instead of a page number to compute.
11. Now the secret. When you put a key in browser JavaScript, the browser must be able to read it in order to send it. So the user can read it too.
12. Minifying does not hide it. It is in the network tab of the developer tools the moment the request is made. In a mobile app it is in the binary, and strings on the file finds it in seconds.
13. The correct shape is a small server of your own in the middle. The browser calls your server with the user’s own session. Your server holds the third party’s key and calls the third party. The key never leaves your machine.
14. There is one legitimate exception, and it is a different kind of value: a publishable key designed to be public, such as a Stripe publishable key or a Maps key restricted by referring domain. These are identifiers, not credentials, and the vendor says so in the documentation. If the documentation does not say so, it is a secret.

■ 36.13.5 TECHNICAL – the engineer’s version

1. Tooling, and what each is genuinely for:

Tool	Strength	Weakness
curl	Exact bytes, everywhere	Verbose to type
httpie	Readable, JSON-aware	Extra install
Postman	Collections, teams	Heavy, cloud-linked
jq	Filtering JSON in shell	Own syntax to learn

2. curl flags worth memorizing: `-i` include response headers, `-D` file dump headers to a file, `-o` file body to a file, `-s` silent, `-S` still show errors, `-w '%{http_code}'` print the status, `-L` follow redirects, `--fail-with-body` return a non-zero exit code on 4xx and 5xx while still printing the body, `--compressed` request gzip, `-v` full trace, and `--connect-timeout` with `--max-time` for deadlines.
3. `--fail-with-body`, added in curl 7.76 (March 2021), is the flag that fixes shell scripts which silently succeed on an error response. Plain `-f` discards the body, which throws away the reason.
4. Conditional requests are specified in RFC 9110, June 2022, sections 8.8 and 13. ETag may be strong, "abc", or weak, W/"abc". Weak means semantically equivalent but not byte-identical, which is why weak validators are valid for caching but not for If-Match on writes.
5. Caching headers you should actually read: Cache-Control with max-age, no-cache (revalidate, do not blindly reuse), no-store (do not keep at all), and private versus public. RFC 9111 is the caching specification. no-cache and no-store mean different things and are constantly confused.
6. Rate-limit handling, in the order GitHub itself recommends: if a retry-after header is present, wait that long; else if x-ratelimit-remaining is zero, wait until x-ratelimit-reset; else wait at least one minute and then back off exponentially. GitHub states plainly that continuing to call while limited may get your integration banned.
7. Concurrency is where well-behaved clients go wrong. Two parallel workers share one quota, so eight workers exhaust a 5,000-per-hour budget eight times faster. Rate-limit your own client, do not rely on being refused.
8. Pagination in the wild:

Style	Request looks like	Trouble
Page number	?page=3&per_page=100	Shifts as data changes
Offset	?offset=200&limit=100	Same shift, slow deep
Cursor	?after=Y3Vyc29y0jIwMA	Opaque, cannot jump
Link header	follow rel="next"	Must parse the header

9. The Link header is RFC 8288, October 2017, Web Linking. GitHub sends rel="next", rel="prev", rel="first" and rel="last", and its guidance is to follow the header rather than construct page URLs yourself.
10. Client-side secrets, precisely. A single-page app, a mobile binary, a desktop Electron app and a browser extension are all **public clients** in OAuth terms: they cannot keep a secret. RFC 9700, the January 2025 Best Current Practice for OAuth 2.0 Security, requires public clients to use the authorization code flow with PKCE and forbids the implicit grant, exactly because there is nowhere safe to put a client secret.
11. Detection and cleanup: gitleaks and trufflehog scan repositories, GitHub push protection blocks known secret formats at push time, and GitHub secret scanning notifies partner providers so leaked keys can be revoked automatically. Rotating a leaked key is not optional; git history is permanent, and removing the file does not remove the commit.

12. Server-side proxy pattern, in three rules: the browser authenticates to your server with the user's own session; your server holds the third-party credential in a secrets manager or environment variable; and your server enforces its own rate limit and authorization, because otherwise you have simply published the upstream API with your key attached.
13. Error handling that survives production: distinguish retryable from terminal (see section 36.10), log the request identifier the API returns, such as GitHub's `x-github-request-id`, and quote it in support tickets. Never log the Authorization header, and be careful with `-v`, which prints it in full.
14. Testing your client without the network: record and replay with VCR-style libraries, run a mock server generated from the OpenAPI document, or point at the vendor sandbox. Test the failure paths on purpose: 429, 500, timeouts, and a truncated response body.
15. Observation: `curl -w` with a format string prints `time_namelookup`, `time_connect`, `time_appconnect`, `time_starttransfer` and `time_total`, which separates DNS from TCP from TLS from server think time, and turns "the API is slow" into a specific accusation.

■ 36.13.6 WORDS – remember these

1. Conditional request — asking only if it changed — a request carrying `If-None-Match` or `If-Modified-Since`, answered with 304 when unchanged.
2. ETag — a label for one version of a resource — an opaque validator, strong or weak, compared by the server rather than interpreted by the client.
3. 304 Not Modified — nothing new, keep what you have — a bodyless response confirming the client's cached copy is current.
4. Offset pagination — ask for rows N to M — index-based paging that skips or repeats items when the underlying data changes mid-scan.
5. Cursor pagination — carry on after this marker — keyset paging using an opaque stable position, immune to concurrent insertions.
6. Link header — the server tells you the next address — RFC 8288 web linking with relations such as `next`, `prev`, `first` and `last`.
7. Public client — software that cannot keep a secret — any client whose code or binary is in the user's hands, per RFC 6749 and RFC 9700.
8. Backend for frontend — your own small server in the middle — a proxy holding third-party credentials so the user's device never sees them.

36.14 APIs as a business

■ 36.14.1 PLAIN – in simple words

1. So far an API has been an engineering thing. It is also a commercial thing, and some of the largest companies alive are shaped by that fact.
2. A **product** is something people use. A **platform** is something people build on. An API is how a product becomes a platform.
3. When other people build on you, they do work you did not pay for, and their customers become reachable by you. That is why a company gives data away.
4. **API-first** means designing the interface before writing the software, and making your own teams use the same interface everyone else uses.
5. That sounds like a small internal rule. In one famous case it reshaped an entire company and then created a new industry. Section 36.14.5 gives the dates.
6. Money comes from APIs in several ways: charging per call, charging by tier, giving it away to sell something else, or taking a share of what flows through it.
7. Then the uncomfortable part. If your business runs on somebody else's API, you do not control your own product. They can change the price, change the rules, or switch it off.
8. This is not a theoretical worry. Real companies have been closed by an announcement from a platform they depended on, and section 36.14.3 names them with years.
9. The lesson is not "never build on an API". Almost everything is built on somebody's API. The lesson is to know exactly how exposed you are, and to price that risk instead of ignoring it.

■ 36.14.2 PLAIN – a picture in your head

1. Think of a railway company that has built a line between two cities.
2. It could run only its own trains. It carries its own goods, it keeps all the revenue, and the size of the business is the size of its own trade.
3. Or it can sell track access. Now other companies run trains on its rails. Each of them invests in carriages and customers that the railway never had to pay for.
4. The railway earns a fee from every train, and the line becomes more valuable the more operators use it, because the destinations are better served.
5. That is the platform move. The track is the API.
6. Now notice the power that comes with owning track. The railway sets the access charge, the timetable slots, and the safety rules.
7. An operator who has bought fifty carriages and hired three hundred staff cannot easily leave when the access charge triples. Their investment is welded to that particular line.
8. And if the railway decides tomorrow to run only its own trains again, those fifty carriages are scrap metal.

Where this comparison breaks:

9. Track takes years to build and is regulated by governments. An API can be changed in an afternoon by one team, and almost nowhere regulates it.
10. A railway operator can see the rails and count the trains. An API user often cannot even tell how many other users there are, or whether they are a rounding error to the platform or its main source of value.

■ 36.14.3 PLAIN – a worked example

1. Here are real cases, with dates. Read them as case law, because that is effectively what they are.

Year	Platform	What changed
2014	Facebook Graph API	Friend data cut off
2014	Netflix	Public API closed
2018	Google Maps	Price up about 14 times
2023	Twitter, now X	Free access ended
2023	Reddit	Charged for API calls

2. Netflix, 2014. On 13 June 2014 Netflix announced that its public API would close, and it did close on 14 November 2014. Applications built on it, including catalogue browsers and recommendation tools, simply stopped.
3. Facebook, 2014 to 2015. Graph API v2.0 arrived on 30 April 2014 and removed the broad access to a user's friend list that many social apps had been built on. Version 1.0 was fully retired on 30 April 2015. A generation of "find your friends who also use this" features died in that change.
4. Google Maps, 2018. Announced in early May 2018 and effective on 16 July 2018, the pricing moved from about 0.50 US dollars per 1,000 dynamic map loads to about 7.00 US dollars per 1,000, a rise of roughly fourteen times.
5. The free allowance changed shape too: from 25,000 map loads per day to a 200 US dollar monthly credit, worth roughly 28,000 loads per month.
6. Read those two numbers together. A site doing 25,000 loads a day was inside the free tier before, and afterwards was about 27 times over the monthly credit. Many small mapping sites moved to OpenStreetMap-based providers or closed the feature.
7. Twitter, 2023. On 19 January 2023 the developer agreement was changed to ban third-party clients. Tweetbot and Twittrific, which had existed since 2007 and 2008, shut down; Twittrific had helped popularize the word "tweet" and the bird icon.
8. In February 2023 free API access ended. On 29 March 2023 the new tiers arrived: a free tier limited to 1,500 posts per month and login, a Basic tier at 100 US dollars per month, and an Enterprise tier reported at around 42,000 US dollars per month.
9. Academic research that had depended on cheap Twitter data largely stopped. That was not a side effect anybody had budgeted for.
10. Reddit, 2023. Announced on 18 April 2023 and effective 30 June 2023. Christian Selig, the developer of the popular client Apollo, calculated that the new rate would cost him about 20 million US dollars per year.
11. Apollo, Sync, BaconReader and rif is fun all shut down on 30 June 2023. Between 12 and 14 June 2023, thousands of subreddits went private in protest, and many stayed dark afterwards.
12. Reddit's chief executive, Steve Huffman, gave the reason openly at the time: the data was valuable, and giving it free to the largest companies in the world was no longer the plan. That is the honest commercial logic behind most of these changes, and it arrived alongside the demand for training data for language models.
13. Now the pattern across all five. In every case the platform was inside its rights, the change was announced, the notice was months not years, and the businesses that died were the ones with no second source.

■ 36.14.4 PLAIN – what is really happening inside

1. Why would a company give its data away at all? Five reasons, and they are not charity.
2. Distribution. Every app built on your API is a salesperson. Slack, Stripe and Twilio grew mostly this way.
3. Lock-in through investment. Once a customer has written six months of code against your interface, moving costs them six months.
4. Data. Every call tells you what people want, in aggregate, before your competitors know.
5. Standards. If your shape of the problem becomes the shape everyone uses, the default choice is yours. Amazon's S3 interface became the shape of object storage, and rivals now advertise being compatible with it.
6. Complements. If maps are cheap, more apps need location, and location services are what you actually sell.
7. Then the money models:

Model	You pay	Typical example
Free, funded elsewhere	Nothing	Search, social login
Freemium quota	Above a limit	Maps, weather
Per call	Each request	Messaging, translation
Tiered subscription	Per month	Data and analytics
Revenue share	A cut of value	Payments, marketplaces

8. Revenue share is the strongest of these, because the API provider's income grows with the customer's success rather than with their request count. It is why payment APIs are such durable businesses.
9. Now the risk, from the caller's side. Ask four questions about any API you build on, before you build.
10. What fraction of my product stops working if this disappears tomorrow?
11. Is there a second supplier with a similar interface, and how many days of work is the switch?
12. Am I paying? Free users are the first to be cut, because they are a cost with no contract.
13. Is my usage growing towards a tier boundary that changes the price by a factor rather than a percentage?
14. The mitigations are unglamorous and they work: put an interface of your own in front of theirs so the dependency touches one file rather than fifty; cache what you are permitted to cache; export your own data regularly; read the terms of service for the notice period; and keep a written estimate of the switching cost.
15. The one that people skip is the first. If every part of your codebase calls the vendor's SDK directly, then swapping vendors means editing everything. If they all call your own thin wrapper, it means editing one adapter.

■ 36.14.5 TECHNICAL – the engineer's version

1. The Bezos mandate. Around 2002, Jeff Bezos issued an internal directive at Amazon that all teams must expose their data and functionality through service interfaces, must communicate only through those interfaces, and must design them to be externalizable, with termination as the stated consequence for non-compliance.
2. Be honest about the source. Amazon never published the memo. The wording everyone quotes comes from a long public post written by the engineer Steve Yegge in October 2011, recalling it from his time at Amazon. So treat the exact phrasing as recollection, not as a document.

3. The outcome, however, is checkable. Amazon Web Services launched publicly in 2006: Simple Storage Service in March 2006 and Elastic Compute Cloud in August 2006. A company that had already forced every internal team to speak only through network interfaces was in a position to sell those interfaces.
4. That is the extreme case of taking internal APIs seriously, and it is the subject of Chapter 43, which covers cloud computing, the mandate, and how AWS and Azure turned internal engineering discipline into an industry.
5. Metrics that API businesses actually run on, distinct from web metrics:

Metric	What it measures
Time to first call	Signup to first 200 response
Time to first value	Signup to first real result
Calls per active key	Depth of integration
Error rate by code	Where users get stuck
Version distribution	How much old shape to keep

6. Time to first call is the one that predicts adoption. Stripe’s reputation was built substantially on making it a few minutes, with a working `curl` command on the front page of the documentation.
7. API-first as an engineering practice means: design and review the interface document before implementation, generate server stubs and client libraries from it, run contract tests in continuous integration, and treat a breaking diff as a build failure. Chapter 41 covers the continuous integration machinery this depends on.
8. The internal-external question is a genuine disagreement among architects. One camp says internal and external APIs should be the same interface, because dogfooding is the only reliable quality signal. The other says external APIs need stability guarantees, versioning, documentation and support that would paralyse internal development, so they should be a separate, deliberately narrower layer. Amazon’s mandate takes the first position; many large companies quietly take the second.
9. Deprecation as a commercial act, not just a technical one. The pattern that holds up in court and in public opinion is: written notice period stated in the terms of service, public changelog entry with a date, Deprecation and Sunset headers from day one (RFC 9745 and RFC 8594), direct contact with accounts still calling the endpoint, and a migration guide.
10. Legal reality, briefly and carefully. In Google versus Oracle America, decided by the United States Supreme Court on 5 April 2021, the court held that Google’s reuse of roughly 11,500 lines of Java SE API declaring code in Android was fair use. The court assumed without deciding that the declarations were copyrightable, so it did not settle whether API definitions can be copyrighted; it settled that this particular reuse was fair. Anyone telling you the case made APIs freely copyable is overstating it.
11. Practical consequence of that case for engineers: reimplementing somebody else’s interface is a normal and long-standing practice, and it is how S3-compatible storage, PostgreSQL wire-protocol clones and POSIX reimplementations exist. It is not risk-free, and it is not settled law outside the United States.
12. Failure modes of an API business, in one list: pricing that punishes exactly the users who grow; a free tier that attracts users the business model cannot support; breaking changes without a version; documentation that lags the implementation; and no sandbox, so nobody can evaluate you without committing.

13. And the strategic trap on the other side: a platform that opens an API, lets an ecosystem grow, then builds the most profitable ecosystem product itself and restricts the interface that made the competitor possible. Every example in section 36.14.3 has some version of this in its history, and it is the strongest argument for owning your own second source.

■ 36.14.6 WORDS – remember these

1. Platform — something other people build on — a product whose value grows with third-party integrations rather than only with its own features.
2. API-first — the interface is designed before the code — a practice where the specification is the source of truth and implementations are generated from or validated against it.
3. Ecosystem — the businesses living on your interface — the set of third-party products whose viability depends on a platform’s continued access terms.
4. Lock-in — leaving is expensive — accumulated switching costs from code, data, staff knowledge and contracts tied to one provider.
5. Sunset, commercially — the announced end of an interface — the dated withdrawal of an API, governed by the terms of service notice period.
6. Second source — an alternative supplier ready in advance — an equivalent provider behind your own abstraction layer, bounding your switching cost.
7. Bezos mandate — talk only through interfaces — the roughly 2002 Amazon directive requiring all internal capability to be exposed as externalizable service interfaces, recounted publicly in 2011.

36.98 Common wrong ideas

1. Wrong: an API means a web endpoint. Right: an API is any published contract between software parts. `strlen`, `write` and `Vulkan` are APIs with no network anywhere near them.
2. Wrong: REST means JSON over HTTP. Right: REST is six constraints defined by Roy Fielding in 2000 and says nothing about JSON. Almost every “REST API” in production is really an HTTP JSON API, because it omits the hypermedia constraint.
3. Wrong: GraphQL replaces REST. Right: GraphQL solves over-fetching and round-trip count for client-driven queries, at the cost of caching, observability and query-complexity control. Many systems run both, for different callers.
4. Wrong: an API key is authentication. Right: a key identifies an account. It rarely expires, is rarely scoped, is often committed to git or logged in a query string, and proves nothing about who is holding it right now.
5. Wrong: versioning means adding v2 to the URL. Right: versioning is a policy about how long you will run old behaviour and who pays for it. The URL, a header, a media type and a pinned date are four ways to express that policy, and none of them is the policy itself.
6. Wrong: a failed request means nothing happened. Right: a lost response is indistinguishable from a lost request at the client. The work may be done. That is partial failure, and the fix is idempotency, not a better error message.
7. Wrong: retrying automatically is always safe. Right: retrying a non-idempotent request without an idempotency key can charge a customer twice, and retrying without jitter can turn a brief outage into a long one.
8. Wrong: a JWT is encrypted, so claims inside it are private. Right: a JWT is base64url-encoded and signed, not encrypted. Anyone holding it can read every claim. The signature proves integrity, not secrecy.
9. Wrong: the API key in my mobile app or single-page app is hidden. Right: any credential shipped to a user’s device is public. Minification, obfuscation and native binaries all lose to developer tools

and the `strings` command.

10. Wrong: my webhook handler is fine because it parses the JSON and then checks the signature. Right: the signature covers the exact bytes that arrived. Re-serializing changes the bytes and the check will fail, or worse, be skipped. Verify the raw body first.

36.99 Chapter summary in 20 lines

1. An API is a published contract about how one piece of software may ask another to do something: what you may ask, how, and what comes back.
2. It is not a technology. It exists at every scale, from `strlen` inside one file to a service on the other side of the planet, and the idea never changes.
3. An API has a surface, a shape and a behaviour. Beginners see only the surface; almost all real bugs live in the behaviour.
4. Nobody invented APIs in one moment. Subroutine libraries appeared around 1951, the phrase “application program interface” in print in 1968, remote procedure calls with Birrell and Nelson in 1984, CORBA in 1991, SOAP and Fielding’s REST thesis around 2000, and GraphQL and gRPC from 2015.
5. Every generation that tried to hide the network behind a local-looking call eventually failed at the same place: partial failure and latency.
6. An API is not an ABI. An ABI fixes registers, calling conventions and struct layout, which is why code can compile and still crash at run time.
7. Semantic versioning states the promise in three numbers: patch fixes, minor adds compatibly, major breaks. The promise is social, not enforced.
8. Crossing a machine boundary through system calls costs a mode switch; crossing the planet costs about ten million times a local call.
9. The eight fallacies of distributed computing, from Sun Microsystems between 1994 and 1997, each name a specific cost you pay for believing them.
10. HTTP is a request line, headers, a blank line and an optional body, both ways. Methods differ in safety and idempotency; status codes are grouped 1xx to 5xx; HTTP/2 and HTTP/3 changed the framing, not the semantics.
11. REST is six constraints, and the one almost nobody implements is the uniform interface’s hypermedia requirement. Say “HTTP JSON API” when that is what you mean.
12. GraphQL, gRPC, SOAP, JSON-RPC, WebSockets and server-sent events each win on a different axis: payload size, typing, streaming, tooling or reach.
13. Authentication asks who you are; authorization asks what you may do. API keys answer neither well. Bearer tokens expire, which is their main gain.
14. OAuth 2.0 with PKCE, now required for all client types by RFC 9700 of January 2025, lets an app act for you without ever seeing your password.
15. A JWT is a signed, readable token, not an encrypted one, and a scope limits the token rather than the user.
16. Reliability is six habits: rate limits with token or leaky buckets, retries with exponential backoff and jitter, idempotency keys, timeouts at every layer, circuit breakers, and graceful degradation.
17. Retrying a non-idempotent request without an idempotency key can double charge a real person, and retrying without jitter can extend an outage.
18. Webhooks invert the call: the server posts to you. Verify the HMAC over the raw body, deduplicate on the event identifier, expect any order, answer fast, and use a reverse tunnel to receive them behind NAT.
19. A usable API is consistent, has machine-readable error codes, an honest versioning and depreca-

tion policy with Deprecation and Sunset headers, an OpenAPI description, generated SDKs and a sandbox.

20. Commercially, an API turns a product into a platform, and the shutdowns of Netflix in 2014, Google Maps pricing in 2018, Twitter and Reddit in 2023 show what a platform's decision can do to businesses built on it. Amazon's roughly 2002 internal mandate is the extreme case, and Chapter 43 follows it.

Git — Why It Works With No Network

37.0 What this chapter gives you

1. You will be able to say what version control is for, and name the six real problems it solves, rather than repeating that it “saves your work”.
2. You will be able to state the difference between a centralized and a distributed version control system in one sentence, and defend it.
3. You will be able to explain why the reader could keep committing to branches for the whole length of their network outage, and push everything later.
4. You will be able to open a `.git` directory, name every entry inside it, and say what each one holds.
5. You will be able to say what “the whole history is local” really means, with real numbers: a 316 MiB `.git` next to a 60 MiB working tree.
6. You will be able to explain that `origin` is a nickname for a URL in a text file, that it has no authority, and that a repository may have none.
7. You will be able to explain that `origin/main` is a memory, not a live reading, and say exactly which commands refresh it and which never do.
8. You will be able to classify every common git command as local or remote, and explain why the split falls exactly where it does.
9. You will be able to state the honest costs of the distributed model, and say why large game studios and hardware firms still buy Perforce.
10. You will be able to say plainly that git and GitHub are different things, and point at the line between them.

37.1 What version control is for

■ 37.1.1 PLAIN – in simple words

1. You are writing something over many days. Code, a report, a design.
2. It changes. Sometimes it gets better. Sometimes you break it.
3. A **version control system** is a program that remembers every state your files have ever been in, and lets you go back to any of them.
4. That one sentence hides six separate problems. Here they are.
5. **History.** What did this file look like three weeks ago, before the change that broke everything?
6. **Undo.** Put the whole project back exactly as it was on Tuesday, including files you deleted and files you renamed.
7. **Blame.** Who wrote this strange line, when, and what else did they change at the same moment, and why did they say they did it?

8. **Parallel work.** You want to try a risky idea without disturbing the working version, and abandon it cheaply if it fails.
9. **Backup.** Your laptop is stolen. The work is not gone, because a copy exists somewhere else.
10. **Collaboration.** Four people edit the same project at once and the system combines their work instead of one person silently erasing another.
11. Notice that only two of those six need another computer. Backup needs one. Collaboration needs one. The other four are questions about your own files.
12. Hold on to that. It is the whole answer to why the reader could keep working with no network.

■ 37.1.2 PLAIN – a picture in your head

1. Before version control existed, people did the obvious thing. They copied the folder.
2. `report`, then `report-final`, then `report-final-2`, then `report-final-USE-THIS-ONE`, then `report-final-v2-jan14-shikhar`.
3. It works for about a week. Then it stops working, for reasons that are worth naming one at a time.
4. You cannot tell what changed between two copies without opening both and reading them side by side, line by line.
5. You cannot tell why anything changed. The folder name holds a date at best. It never holds a reason.
6. Disk use grows with the number of copies, not with the amount of change. Ten copies of a 200 MB project is 2 GB, even if you only edited one line.
7. Two people cannot work at once. If both copy the folder and both edit, you have two folders and no way to combine them except by hand.
8. And the names lie. `final` is never final. Everyone has met a `final-v3-really-final`.

Where this comparison breaks:

9. Dated folders are not useless. For a one-person, one-week job with no branching, they are genuinely fine, and pretending otherwise is dishonest.
10. Version control's real win is not that it saves copies. It is that it saves the *reasons* and the *relationships* between copies. A folder cannot say "this version came from that version, and here is why".
11. Also, git does not actually store a full copy per version, which is why ten versions of a big project cost far less than ten folders. That is 37.5.

■ 37.1.3 PLAIN – a worked example

1. Here is a real bug hunt, done with folders, then done with version control.
2. The situation: a program worked on 1 April and is broken on 13 August. There were 400 changes in between.
3. With dated folders you have perhaps twelve snapshots. You open them one at a time until you find the first broken one. Then you diff two folders that are three weeks apart and stare at 6,000 changed lines.
4. With version control you run one command that does a binary search across all 400 changes.
5. Here it is running for real, in a sandbox, on the git project's own source repository, which holds 85,263 commits.

```
$ git bisect start
$ git bisect bad
$ git bisect good <first commit>
Bisecting: 2 revisions left to test after this (roughly 1 step)
[c35f063d2751d7340c2cdd4fe0046a137c9164c4] Offline commit 2
```

6. It halves the range every step. 400 changes needs about 9 tests, because 2 to the power 9 is 512.
7. At the end you do not get “somewhere in three weeks”. You get one commit, one author, one timestamp, one message, and the exact lines it touched.
8. And every single step of that ran with no network at all. That matters later.

■ 37.1.4 PLAIN – what is really happening inside

1. A version control system stores two different kinds of thing.
2. First, **content**: the bytes of your files at various moments.
3. Second, **metadata**: who, when, why, and which earlier state this one came from.
4. The second kind is what turns a pile of backups into a history. Backups are a set. A history is a chain.
5. Systems differ in how they store the content. There are three families.
6. **Full snapshots**: keep every version whole. Simple, fast to read, wasteful on disk unless you compress hard afterwards. This is git’s model.
7. **Forward deltas**: keep the oldest version whole plus a list of edits to move forward. Cheap to store, slow to get the newest version, because you must replay every edit.
8. **Reverse deltas**: keep the newest version whole plus edits to walk backward. Fast for the common case, because you usually want the newest. This is what RCS chose in 1982.
9. Git looks like family one and behaves like family two after packing. It stores each version as a whole object, then later rewrites groups of objects into a compressed pack with deltas inside it. You never see this happen.

■ 37.1.5 TECHNICAL – the engineer’s version

1. The lineage of version control systems, with checkable dates.

System	Year	Origin
SCCS	1972	Rochkind, Bell Labs
RCS	1982	Tichy, Purdue
CVS	1986 / 1990	Grune, then Berliner
Subversion	2000	CollabNet
BitKeeper	2000	McVoy, BitMover
Git	2005	Torvalds

2. **SCCS**, the Source Code Control System, was written by Marc Rochkind at Bell Labs in late 1972, first for an IBM System/370 running OS/360.
3. He rewrote it in C for Unix on a PDP-11 in 1973. The first publicly available version, SCCS version 4, shipped on 18 February 1977 with the Programmer’s Workbench edition of Unix.
4. SCCS stored **interleaved deltas**: all versions of a file woven into one file, with markers saying which lines belong to which revision.
5. **RCS**, the Revision Control System, came from Walter F. Tichy at Purdue University, first published in 1982.
6. RCS switched to **reverse deltas**. Tichy’s argument was that recent revisions are read far more often than old ones, so the newest should be the cheap one to fetch.
7. Both SCCS and RCS versioned **one file at a time**. There was no concept of a project-wide change. A change touching four files was four unrelated events.
8. **CVS**, the Concurrent Versions System, began as shell scripts by Dick Grune in 1986. Brian Berliner’s C rewrite started in April 1989, and CVS version 1.0 was submitted to the Free Software Foundation on 19 November 1990.

9. CVS was a front end over RCS. It added project-wide operation and a client or server split, but it still had no atomic commit: an interrupted commit could leave half the files updated.
10. **Subversion** was founded by CollabNet in 2000, explicitly to be “a better CVS”. Version 1.0 was released in February 2004. It gave the world atomic commits, real directory versioning, and proper renames.
11. Subversion became an Apache Incubator project in November 2009 and a top-level Apache project on 17 February 2010.
12. Subversion is still centralized. That is the hinge of this chapter, and 37.2 takes it apart.
13. Tools to observe history in git: `git log`, `git log -p`, `git log --stat`, `git blame`, `git bisect`, `git show`, `git diff`, `git reflog`.

■ 37.1.6 WORDS – remember these

1. Version control — a program that remembers every state of your files — a system that records content plus metadata and the parent relationships between revisions.
2. Commit — one saved point in the history — an immutable object naming a tree, zero or more parents, an author, a committer and a message.
3. Delta — the difference between two versions — a stored edit script that reconstructs one revision from another.
4. Snapshot — a whole copy of everything at one moment — a tree object recording the complete state of the project at one commit.
5. Blame — finding who last changed each line — per-line attribution computed by walking history backward through renames and copies.
6. Bisect — halving the search to find a breaking change — automated binary search over the commit graph with `git bisect`.

37.2 Centralized versus distributed

■ 37.2.1 PLAIN – in simple words

1. This is the most important section in the chapter. Everything else follows from it.
2. There are two ways to build a version control system, and they differ in one question: **where does the history live?**
3. In a **centralized** system, the history lives on a server. Exactly one copy exists, in one place.
4. What is on your machine is a **working copy**. It is the files as they exist at one chosen revision, plus a little bookkeeping. It is not the history.
5. So when you save a change, that change has to travel to the server, because the server is the only thing that can record it.
6. Which means: **a commit is a network operation**. No network, no commit. The server is down, you cannot save your work as a version. You can only edit files and hope.
7. In a **distributed** system, every copy is a complete repository. Not a working copy. A repository, with the entire history in it.
8. Your machine has every commit, every version of every file, every branch, right there on your own disk.
9. So when you save a change, nothing has to travel anywhere. The change is recorded in a directory on your laptop.
10. Which means: **a commit is a local file operation**. No network needed. No server needed. No permission needed.

11. And that is the whole answer to the reader's question. During their outage they were writing to their own disk. There was nothing for the network to break.

■ 37.2.2 PLAIN – a picture in your head

1. Centralized version control is a **library with one copy of each book**.
2. To read a book you go to the library. To add a chapter you go to the library, because the book is there and nowhere else.
3. If the library is shut, or the road is flooded, you cannot add a chapter. You can write on loose paper at home, but that writing is not in the book yet, and nothing keeps track of it.
4. Distributed version control is **everyone owning a full photocopy of every book, including all the old editions**.
5. You write a new chapter into your own copy immediately. It is a real chapter in a real book, dated and numbered, the moment you write it.
6. Later, when you next meet, you and a friend compare copies and exchange the chapters each of you is missing.
7. There is no library in this picture. There can be one by agreement, and there usually is, but the system does not require it.

Where this comparison breaks:

8. Photocopying every book sounds absurdly wasteful, and for paper it would be. Text compresses extraordinarily well, so in practice it is cheap. Real figures are in 37.5.
9. The picture also suggests that merging two copies is easy. It is not always. If you and your friend both rewrote page 40, someone has to decide. Git can combine most changes automatically and refuses when it genuinely cannot.
10. And the honest version: a distributed system does not remove the need for a shared place. Teams still nominate one. The difference is that the shared place is a **convention**, not a structural requirement.

■ 37.2.3 PLAIN – a worked example

1. Two engineers, same task, same broken link. One uses Subversion. One uses git. Here is what each can do.

Action	Subversion	Git
Edit a file	works	works
See old version	needs server	works
Commit a change	needs server	works
Create a branch	needs server	works
See full log	needs server	works
Merge two branches	local, mostly	works
Get others' new work	needs server	needs server
Publish your work	needs server	needs server

2. In Subversion, five of those eight become impossible the moment the link dies. In git, six of the eight keep working.
3. The two that stop are exactly the two that involve another computer: receiving other people's work, and sending yours.
4. That is not a coincidence or a clever optimization. It is the direct, inevitable consequence of where the history is stored.

5. Here is the real demonstration, run in a sandbox with the remote deliberately made unreachable.

```
$ git commit -m "Committed during the outage"
COMMIT SUCCEEDED with no network

$ git fetch
fatal: Could not read from remote repository.

$ git ls-remote
fatal: Could not read from remote repository.
```

6. Same repository, same second, same broken remote. The local command succeeded. The two remote commands failed instantly and cleanly.

■ 37.2.4 PLAIN – what is really happening inside

1. Let us be exact about what each system keeps on your disk.
2. A Subversion working copy has a `.svn` directory. Historically one per folder, and since Subversion 1.7 in 2011, a single one at the top.
3. Inside it is a record of which revision you checked out, and a **pristine copy** of each file at that revision, so that `svn diff` and `svn revert` can work without the server.
4. That is a genuine and often-forgotten point: Subversion is not fully online. Local diff and local revert work offline, because of those pristine copies.
5. What it does not have is any other revision. It cannot show you revision 900 if you checked out revision 1000. It has two states: yours and the one you checked out.
6. A git repository has a `.git` directory containing every object ever created in that project's history, and the pointers that organize them.
7. So `git log` is a walk over local files. `git diff HEAD~50` reconstructs a fifty-commit-old state from local files. `git checkout` of a two-year-old tag reads local files.
8. Here is the structural difference, drawn.

```
CENTRALIZED (Subversion, CVS, Perforce)

[ server ] <- the history lives here, one copy
      |
      | network required to commit
      |
[ laptop ] working copy = one revision + pristine files


DISTRIBUTED (Git, Mercurial)

[ server ] a repository, by agreement only
      |
      | network required only to sync
      |
[ laptop ] a repository = ALL commits, ALL versions
```

9. In the top picture, cutting the line stops work. In the bottom picture, cutting the line stops sharing.

■ 37.2.5 TECHNICAL – the engineer's version

1. The formal distinction: in a centralized VCS the repository is a single authoritative store, usually addressed by revision numbers that are globally ordered. In a distributed VCS every clone holds the full object database, and revisions are named by content hash because no global ordering exists.

2. That last clause is the deep reason git uses SHA-1 hashes rather than numbers like r4213. With no central authority, nobody can hand out sequential numbers, so identity must be computed from content instead.
3. Subversion can number commits 1, 2, 3 because exactly one process assigns them. Git cannot, so it names a commit e83c5163316f89bfbd7d9ab23ca2e25604af290.

Property	Centralized	Distributed
History location	server only	every clone
Commit needs network	yes	no
Revision names	integers	content hashes
Branch cost	server-side copy	one 41-byte file
Offline log or diff	no	yes
Per-file locking	supported	not supported

4. Real timings, measured in a sandbox on the git project's own repository, 85,263 commits and 4,845 tracked files, git 2.43.0 on Linux.

Command	Kind	Time
git rev-parse HEAD	local	4 ms
git branch -a	local	3 ms
git log --oneline -1000	local	19 ms
git status	local	228 ms
git ls-remote origin	remote	751 ms
git fetch origin (no-op)	remote	456 ms

5. Read that table carefully. The local commands are not merely faster. They are in a different class, because their cost has no round-trip term in it.
6. And the sandbox had an excellent link to github.com. The reader, on home broadband in India reaching a Microsoft edge in Delhi, would see far higher remote numbers on a good day, and unbounded numbers on the bad day this book keeps returning to.
7. The distributed model is not new with git. GNU Arch appeared in 2001, Monotone in 2003, and Darcs in 2003. Git and Mercurial, both from April 2005, are the two that survived at scale.
8. Perforce, founded by Christopher Seiwald in 1995 and now sold as P4, formerly Helix Core, is centralized by design and remains widely used. 37.10 explains why that is a reasoned choice and not a mistake.

■ 37.2.6 WORDS – remember these

1. Centralized VCS — the history lives on one server — a system with a single authoritative repository that all clients must contact to commit.
2. Distributed VCS — everybody has the whole history — a system where every clone contains the complete object database and refs.
3. Working copy — the files as they are right now — a checkout of one revision plus local bookkeeping, without the full history.
4. Repository — the store of all history — the object database plus refs, in git the contents of the .git directory.
5. Clone — your own full copy — a complete repository created from another, including all reachable objects, plus a remote entry pointing back.

6. Round trip — one there-and-back over the network — the request and response latency that every remote operation must pay at least once.

37.3 Why Git was made

■ 37.3.1 PLAIN – in simple words

1. Git was not designed in a calm room over two years. It was written in a hurry, in April 2005, because something was taken away.
2. From 2002 the Linux kernel was developed using **BitKeeper**, a commercial distributed version control system made by a company called BitMover, run by Larry McVoy.
3. BitKeeper was not free software, but McVoy gave the kernel community free use of it. Some kernel developers objected loudly to depending on a proprietary tool. It was used anyway, and it worked well.
4. In April 2005 that arrangement ended. BitMover announced it would stop providing the free version to the community. The official cutoff was 1 July 2005.
5. So the largest and busiest software project in the world suddenly had no version control system, and about three months to find one.
6. Nothing available was good enough. CVS and Subversion were centralized and far too slow for the kernel's volume. The existing distributed tools could not handle the size.
7. So Linus Torvalds wrote a new one. He started on 3 April 2005. He announced it on 6 April. On 7 April it was storing its own source code in itself.
8. That first commit still exists and you can read it today.
9. Then he handed it over. Junio Hamano became the maintainer on 26 July 2005, and has led the project ever since. Version 1.0 was released on 21 December 2005.

■ 37.3.2 PLAIN – a picture in your head

1. Imagine a factory of ten thousand workers that runs on one specialized machine, borrowed free from a neighbour.
2. One morning the neighbour says the machine goes back in ninety days.
3. Nothing on the market fits. The machines you can buy are built for factories of thirty people, and they jam when a thousand parts arrive at once.
4. So the foreman goes into the workshop and builds a replacement himself, from the simplest possible parts, choosing crude and fast over clever and fragile, because the deadline is real.
5. Four days later it is running. Not finished. Running, and holding its own blueprints.
6. That is what happened, and it explains git's personality. The parts are very simple. The joins are visible. The commands were designed for the machine first and the human second.

Where this comparison breaks:

7. Git in 2026 is not the tool of April 2005. Thousands of contributors have worked on it for twenty-one years. The friendly commands, `git switch` and `git restore`, arrived in version 2.23 on 16 August 2019.
8. And the “ten days” story is often told as though one person wrote all of git in ten days. He did not. He wrote a working core in days and led it for under four months. What you type today is mostly other people's work.

■ 37.3.3 PLAIN – a worked example

1. Here is the first commit of git, read out of a real clone of the git project's own repository, in a sandbox, today.

```
$ git log --reverse --max-parents=0 --format='%H%n%an <%ae>%n%ad%n%s'

e83c5163316f89bfbde7d9ab23ca2e25604af290
Linus Torvalds <torvalds@ppc970.osdl.org>
Thu Apr 7 15:13:13 2005 -0700
Initial revision of "git", the information manager from hell
```

2. Every fact in that block is checkable by anyone with git installed and a network connection.
3. The date confirms the timeline: four days after he started, git was already the thing storing git.
4. The email domain osdl.org is the Open Source Development Labs, which employed Torvalds at the time and later merged into the Linux Foundation in 2007.
5. The machine name ppc970 is a PowerPC 970 processor, which is what was inside an Apple Power Mac G5 of that era.
6. The commit message is a joke about how unfriendly the tool was. It was accurate.

■ 37.3.4 PLAIN – what is really happening inside

1. Torvalds stated his goals for the new system. They are worth taking one at a time, because each one shows up in the design you use today.
2. **Speed.** He wanted patching to take no more than three seconds on the kernel. This is why git stores things in a way that is cheap to read and why almost everything is local.
3. **Simple design.** The core is a content-addressed store: a key-value database where the key is the hash of the value. Chapter 38 covers it. The simplicity is why the format has survived twenty-one years unchanged.
4. **Strong support for non-linear development.** Thousands of parallel branches. In git a branch is a file containing a hash. Creating one costs 41 bytes and no network. This is why git users branch casually.
5. **Fully distributed.** Every clone complete. This is the property this whole chapter is about.
6. **Able to handle large projects efficiently,** in both speed and data size. The kernel was already enormous in 2005.
7. He added a further requirement that is often left out: **very strong safeguards against corruption, accidental or malicious.** This is why every object is named by the hash of its own content, and why a commit's hash covers its parent's hash, so tampering with old history changes every hash after it.
8. And one more thing shaped it: the kernel has thousands of contributors and no central authority who may write to everyone's tree. The design had to assume that trust is granted person to person, not by a server.

■ 37.3.5 TECHNICAL – the engineer's version

1. Verified timeline of April to December 2005.

Date	Event
4 May 2000	BitKeeper first public release
2002	Kernel adopts BitKeeper

Date	Event
April 2005	BitMover ends free version
3 April 2005	Git development begins
6 April 2005	Git announced on the list
7 April 2005	Git self-hosting, commit e83c516
19 April 2005	Mercurial announced
26 July 2005	Junio Hamano becomes maintainer
21 December 2005	Git 1.0.0 released
9 May 2016	BitKeeper open-sourced

- The trigger was a dispute over reverse engineering. Larry McVoy alleged that Andrew Tridgell, the author of Samba and rsync, had reverse engineered the BitKeeper protocol to build a tool called SourcePuller. BitMover withdrew the free community licence in response.
- BitKeeper itself was eventually released as open source on 9 May 2016, under the Apache License version 2, eleven years too late to matter.
- Mercurial** was announced by Olivia Mackall on 19 April 2005, for the same reason and in the same month. Both projects were candidates to replace BitKeeper for the kernel. Git won.
- Git's version history, taken from tags in the real repository, gives useful anchors for what is old and what is new.

Version	Released	Notable
v1.0.0	21 Dec 2005	first stable
v2.19.0	10 Sep 2018	partial clone
v2.23.0	16 Aug 2019	switch, restore
v2.29.0	19 Oct 2020	SHA-256 experimental
v2.43.0	20 Nov 2023	used in this chapter
v2.55.0	29 Jun 2026	current at writing

- The object format has been stable since 2005. A repository created with git 1.0 in December 2005 can be read by git 2.55 in 2026 without conversion. That is a rare property and it was deliberate.
- Git remains a C project with a small core and a very large surface. The original design is described in the Documentation/technical directory of the source, which ships with every clone.
- The first Linux kernel release managed entirely with git was 2.6.12 in June 2005, about two months after git's first commit.

■ 37.3.6 WORDS – remember these

- BitKeeper — the commercial tool the kernel used before git — a proprietary distributed VCS by BitMover whose free community licence ended in April 2005.
- Self-hosting — a tool storing its own source in itself — the point at which a system is complete enough to manage its own development.
- Non-linear development — many people working on many branches at once — a commit graph that is a directed acyclic graph rather than a straight line.
- Content-addressed store — you look things up by what they are, not where they are — a key-value store where the key is a cryptographic hash of the value.
- Maintainer — the person who decides what goes in — the individual holding the canonical tree and applying or rejecting contributions.
- Directed acyclic graph — a web of points with one-way arrows and no loops — the DAG formed by commits and their parent pointers.

37.4 What is actually inside a .git directory

■ 37.4.1 PLAIN – in simple words

1. When you run `git init`, git makes one hidden directory called `.git` and puts a handful of files in it.
2. That directory is the repository. Everything else you see is just your files as they currently are.
3. Delete `.git` and you have a plain folder with no history. Copy `.git` somewhere else and you have carried the entire project's past with you.
4. There is no database server. No background process. No registry entry. No cloud account. Files in a directory.
5. This is worth pausing on, because it is the reason for everything in this chapter. If a thing is a local file, reading it does not need a network.
6. Here is what a real `.git` looks like immediately after `git init` and one commit, listed in a sandbox with git 2.43.0.

```
$ ls -a .git
COMMIT_EDITMSG  description  index  objects
HEAD           hooks       info   refs
branches       config      logs
```

7. Eleven entries. That is a complete version control system.

■ 37.4.2 PLAIN – a picture in your head

1. Think of `.git` as a **warehouse with a whiteboard by the door**.
2. `objects` is the warehouse itself. Every crate ever received is in there, stored by a label computed from its contents. Nothing is ever moved or edited, only added.
3. `refs` is the whiteboard. It has a short list of names, and next to each name a crate label. “main: crate 9902ff97”. “v0.1: crate b398d1b0”.
4. `HEAD` is a sticky note on the whiteboard saying which of those names you are currently standing at.
5. `index` is the loading bay: things you have carried out of the aisles and set down, ready to be made into the next crate, but not yet crated.
6. `config` is the notice board with local rules on it, including the phone numbers of other warehouses you exchange crates with.
7. `logs` is the security camera footage of the whiteboard: every time a name was pointed at a different crate, and when.
8. `hooks` are the little printed cards saying “before accepting a delivery, check this”. By default they are all examples and none are active.

Where this comparison breaks:

9. In a real warehouse, taking a crate out removes it. In `objects` nothing is removed by reading, and almost nothing is ever removed at all until a garbage collection runs, which needs no network either.
10. And a real warehouse can run out of aisles. Git periodically rewrites thousands of loose crates into one enormous compressed container, called a pack. The contents are identical, the storage is a different shape.

■ 37.4.3 PLAIN – a worked example

1. Here is a genuine `.git` from a repository created in a sandbox, one file committed, printed in full. Nothing is edited or shortened.

```
$ cat .git/HEAD
ref: refs/heads/main

$ cat .git/refs/heads/main
9902ff97b5e8ac502e8ac752d950f15257c19335

$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[user]
    name = Reader
    email = reader@example.com
```

2. Read `HEAD` first. It contains the text `ref: refs/heads/main`. It does not contain a commit. It contains the *name* of a place that contains a commit.
3. Now read that place. `.git/refs/heads/main` contains 40 hexadecimal characters and a newline. 41 bytes. That is a branch.
4. A branch in git is a 41-byte file whose contents are a commit's name. That is the entire implementation. There is nothing else.
5. This is why branching is instant and free, and why it needs no network. Creating a branch writes 41 bytes to your own disk.
6. And `config` is an INI-style text file you can open in any editor. There is no binary blob, no encrypted store, no service account.
7. Here is what `git init` prints, verbatim, so you can see how little happens.

```
$ git init initdemo
Initialized empty Git repository in /tmp/gitdemo/initdemo/.git/
```

8. One directory made. No server contacted. It works on a plane.

■ 37.4.4 PLAIN – what is really happening inside

1. Now every entry, one at a time, with what it is for.
2. **objects/** — the object database. Every commit, every directory listing, every file version, stored under its own hash. Chapter 38 is entirely about this. It is where nearly all the bytes are.
3. **refs/** — human names for commits. `refs/heads/` holds your branches, `refs/tags/` holds tags, `refs/remotes/` holds remote-tracking refs, which are section 37.7 and the heart of the reader's question.
4. **HEAD** — where you are now. Normally a symbolic reference: text saying `ref: refs/heads/main`. If it holds a raw hash instead, you are in the state git calls detached `HEAD`.
5. **config** — settings for this repository only. Your identity, your remotes, and any per-repository preferences. Text, editable, version 0 format.
6. **index** — the staging area. A binary file listing exactly what will go into the next commit, with each path, its mode, its blob hash and cached file metadata so `git status` can skip unchanged files. Chapter 39 covers it.

7. **hooks/** — scripts git will run at certain moments. On a fresh repository every one is a `.sample` file, and a `.sample` file never runs. Nothing is active until you rename one and make it executable.
8. **logs/** — the reflog. A line-per-change record of every value each ref has held, with who and when. This is your safety net after a bad reset.
9. **info/** — extra bits. `info/exclude` is a private ignore list that is not committed and not shared, unlike `.gitignore`.
10. **description** — used only by the old gitweb viewer. On a normal repository it is dead weight and you can ignore it.
11. **COMMIT_EDITMSG** — a scratch file holding the message you typed last. Not part of the history. Purely a convenience.
12. **branches/** — an obsolete shorthand directory from very early git. It is still created for compatibility and is always empty. Ignore it.
13. **packed-refs** — appears once refs are compressed into one file. See below.
14. The honest version: the exact set of files is an **implementation detail**, not a standard. It has changed over time and can differ with configuration. What is guaranteed is the on-disk *object format* and the general layout.

■ 37.4.5 TECHNICAL - the engineer's version

1. Real du output for the `.git` of a full clone of the git project itself, git 2.43.0, cloned in a sandbox on 13 August 2026.

Entry	Size	Purpose
objects	316 MiB	all history
index	452 KiB	staged tree, 4845 files
packed-refs	104 KiB	2025 refs in one file
hooks	68 KiB	14 inactive samples
logs	32 KiB	reflogs
refs	28 KiB	loose refs
config	4 KiB	settings
HEAD	4 KiB	current position

2. The object store is 99.9 percent of it. Everything else is bookkeeping.
3. **packed-refs**: once a repository has many refs, git writes them into one sorted text file rather than one file per ref, because thousands of tiny files are slow on every filesystem. Real head of that file:

```
# pack-refs with: peeled fully-peeled sorted
165e5ad3169d0fd26637da3383a4514f1a9d1e72 refs/remotes/origin/bisect
426ec014a6630fb78c5a7334461bc33568f07743 refs/remotes/origin/jch
e9019fcafe0040228b8631c30f97aeadb61bcd refs/remotes/origin/maint
745601a9a94110d74769ab605ccd4f61339758d2 refs/remotes/origin/master
```

4. A loose ref file always wins over a packed-refs entry for the same name. That is the lookup rule.
5. **index** format: a binary file beginning with the four bytes DIRC, short for “dircache”, then a version number and an entry count.

```
$ head -c 12 .git/index | od -c
00000000  D  I  R  C  \0  \0  \0 002  \0  \0  \0 002
```

6. Those twelve bytes read: signature DIRC, version 2, two entries. Index versions 2, 3 and 4 exist; version 4 adds path prefix compression.

7. `git ls-files --stage` decodes it for you:

```
100644 bd9f1115522d0cca0ac0e7b22f7d256e08287438 0    README.md
100644 31489c9ee8c455a4964beabcf6dd91ca1cea9ad4 0    src/main.c
```

8. Columns: file mode, blob hash, merge stage number, path. Stage 0 means no conflict. Stages 1, 2 and 3 appear during a conflicted merge and are Chapter 39.
9. The fourteen sample hooks in git 2.43.0 are `applypatch-msg`, `commit-msg`, `fsmonitor-watchman`, `post-update`, `pre-applypatch`, `pre-commit`, `pre-merge-commit`, `pre-push`, `pre-rebase`, `pre-receive`, `prepare-commit-msg`, `push-to-checkout`, `sendemail-validate`, `update`.
10. Useful inspection commands: `git rev-parse --git-dir`, `git count-objects -vH`, `git for-each-ref`, `git cat-file -p HEAD`, `git ls-files --stage`, `git symbolic-ref HEAD`, `git config --list --show-origin --show-scope`.
11. A **bare** repository, made with `git init --bare`, has these same entries at the top level with no working tree and no index. That is what a server stores. It is the same data with the checkout removed.

■ 37.4.6 WORDS – remember these

1. Object database — the store of everything ever committed — the `.git/objects` directory holding loose and packed objects keyed by hash.
2. Ref — a name that points at a commit — a file under `.git/refs` or an entry in `packed-refs` containing a 40-character object ID.
3. HEAD — where you are standing now — a symbolic ref, usually `ref: refs/heads/<branch>`, or a raw object ID when detached.
4. Index — the list of what goes in the next commit — a binary DIRC file of staged path, mode, blob ID and cached stat data.
5. Reflog — the record of where each name used to point — per-ref append-only logs under `.git/logs`, used to recover lost commits.
6. Bare repository — a repository with no working files — a `.git` layout at the top level, with `core.bare` true and no index or checkout.
7. Hook — a script git runs at a set moment — an executable in `.git/hooks` with an exact name, disabled by default and never cloned.

37.5 What “the whole history” really means

■ 37.5.1 PLAIN – in simple words

1. When we say a clone contains the whole history, that phrase is doing a lot of work. Let us make it exact.
2. It means every commit ever made on any branch that was fetched.
3. It means every version of every file, including files that were deleted years ago and no longer exist in your folder.
4. It means every commit message, every author name, every date, every parent link, and every tag.
5. All of it, on your disk, right now, readable with no network.
6. The obvious worry is size. If a project has 85,000 commits, and each one is a snapshot of the whole project, surely the disk fills up?
7. It does not, and the reason is that source code is text, text is enormously repetitive, and git compresses it very hard.

8. Two mechanisms do the work. First, identical content is stored exactly once, because content with the same bytes gets the same name. If a file did not change between two commits, both commits point at the same stored copy.
9. Second, git periodically rewrites the store into a **pack**, which compresses everything and stores similar files as small differences from each other.
10. The result is that the whole past often costs only a few times the size of the present.

■ 37.5.2 PLAIN – a picture in your head

1. Imagine photographing a whiteboard every day for five years. That is 1,800 photographs.
2. Stored naively, 1,800 full-resolution photographs is a lot of disk.
3. But most days almost nothing changes. So instead you keep one full photograph and, for each other day, a short note: “same as yesterday except the top-right box now says 42”.
4. Now 1,800 days of history costs slightly more than one photograph plus 1,799 short notes.
5. That is a pack file. Git finds files that resemble each other, picks one to store whole, and stores the others as instructions relative to it.
6. And it compresses everything on top of that, the way a zip file does.

Where this comparison breaks:

7. Git does not compare “yesterday to today” the way the photo story suggests. It compares any object to any similar object, regardless of date, using a heuristic based on file name and size.
8. Also a chain of differences cannot be infinitely long, or reading the newest version would require replaying thousands of steps. Git limits chain depth, by default to 50, so reads stay fast.
9. And binary files break the trick. A photograph, a compiled program or a video changes almost entirely with every edit and compresses badly. This is a real limitation and 37.10 returns to it.

■ 37.5.3 PLAIN – a worked example

1. These are real measurements from a full clone of the git project’s own source repository, made in a sandbox on 13 August 2026.

Measure	Value
Commits in history	85,263
Files in working tree	4,845
Working tree on disk	60 MiB
.git directory	316 MiB
Objects stored	418,900
Pack files	1

2. So twenty-one years and 85,263 commits cost about 5.3 times the size of the current checkout. That is the answer to “surely it fills the disk”.
3. Now the compression, measured object by object with `git cat-file --batch-all-objects`.

Object type	Count	Raw / On disk
blob (file data)	165,095	5364 / 194 MiB
tree (directories)	167,536	2453 / 74 MiB
commit	85,263	59 / 33 MiB
tag	1,006	0.7 / 0.6 MiB

4. Add them up: about 7,877 MiB of raw object content stored in about 302 MiB on disk. A ratio of roughly 26 to 1.
5. File data compresses best, 5,364 MiB down to 194 MiB, about 27 to 1, because most versions of most files are nearly identical to another version.
6. Commits compress worst, 59 MiB down to 33 MiB, about 1.8 to 1, because every commit contains different hashes, and hashes are random-looking text that does not compress.
7. That last line is a nice piece of evidence: compression works on repetition, and hashes have none by design.

■ 37.5.4 PLAIN – what is really happening inside

1. Two storage shapes exist and both are always valid.
2. **Loose objects.** One file per object, at `.git/objects/ab/cdef...`, where `ab` is the first two hex characters of the hash and the rest is the file name. The content is compressed with `zlib`.
3. The two-character directory split exists because filesystems slow down badly with hundreds of thousands of files in one directory. It spreads them over 256 directories.
4. **Packed objects.** Many objects in one `.pack` file, with a `.idx` file beside it giving the byte offset of each object so lookup stays fast.
5. Inside a pack, an object may be stored whole, or as a **delta** against another object in the same pack. The delta is a tiny instruction list: copy these bytes from the base, then insert these new bytes.
6. Packing happens automatically. `git gc` runs when loose objects accumulate, and a fresh `git clone` always arrives as a pack because the sending side builds one for the transfer.
7. That is why a freshly cloned repository shows `count: 0` loose objects and one large pack.
8. Nothing about this needs a network. Packing, unpacking, delta resolution and garbage collection are all local CPU and disk work.

■ 37.5.5 TECHNICAL – the engineer's version

1. Object naming: `git 1.0` through the present names objects by SHA-1 of the header plus content. SHA-256 repositories became available as an experimental feature in `git 2.29`, released 19 October 2020, and interoperability with SHA-1 repositories is still incomplete as of `git 2.55` in June 2026.
2. Default packing parameters, from `git config` documentation for `git 2.43`: `pack.depth 50`, `pack.window 10`, `core.compression -1` meaning `zlib` default level 6, `gc.auto 6700` loose objects.
3. Loose object path scheme: `.git/objects/<first 2 hex>/<remaining 38 hex>`. Real listing from a sandbox repository:

```
.git/objects/b1/a8becadd478624373132f066d206859ade455f
.git/objects/fb/560ef022402ab49df96a3b868aa7c540bela57
.git/objects/e6/69826a4efde29351886aa6e636e14635a29827
.git/objects/c3/5f063d2751d7340c2cdd4fe0046a137c9164c4
```

4. **Shallow clone:** `git clone --depth N` fetches only the last `N` commits per branch. The repository records a `.git/shallow` file listing the commits whose parents are deliberately absent.
5. **Partial clone:** `git clone --filter=blob:none` fetches all commits and trees but no file contents, downloading blobs on demand. Introduced in `git 2.19`, 10 September 2018. It sets `remote.origin.promisor = true` and `remote.origin.partialclonefilter` in the config.
6. Real measurements of the same repository, four ways, same sandbox, same day.

Clone type	.git size	Clone time
full	316 MiB	34 s

Clone type	.git size	Clone time
blobless filter	130 MiB	38 s
treeless filter	51 MiB	25 s
shallow depth 1	14 MiB	4.6 s

7. What you lose, stated precisely.
8. Shallow: `git log` shows one commit, not 85,263. `git blame` cannot see past the cut. `git bisect` cannot search history you do not have. Pushing from a shallow clone works but has restrictions, and deepening requires the network.
9. Blobless: history commands work, but any command needing file *content* from an old commit, such as `git blame` or `git log -p`, silently triggers a network fetch. Offline, those commands now fail.
10. That is the sting. A partial clone quietly reintroduces the network into operations that were local. During an outage, a blobless clone behaves less like git and more like Subversion for exactly those commands.
11. Real config written by a blobless clone:

```
[remote "origin"]
  url = https://github.com/git/git.git
  fetch = +refs/heads/*:refs/remotes/origin/*
  promisor = true
  partialclonefilter = blob:none
```

12. Guidance: use shallow clones for continuous integration runners that build once and are destroyed. Use full clones for machines where people work, especially machines that go offline.

■ 37.5.6 WORDS - remember these

1. Blob — the stored contents of one file version — a git object holding raw file bytes with no name and no permissions.
2. Pack file — one big compressed container of many objects — a `.pack` with a companion `.idx` giving offsets, using delta chains internally.
3. Delta — a small “difference from another object” — copy and insert instructions against a base object within the same pack.
4. zlib — the common compression used inside git — DEFLATE compression, RFC 1950 and 1951, applied to every loose object and pack entry.
5. Shallow clone — a copy with only the newest commits — a clone with truncated history recorded in `.git/shallow`, created by `--depth`.
6. Partial clone — a copy that fetches file contents on demand — a promisor clone created with `--filter`, requiring the remote for missing objects.
7. Garbage collection — tidying the store — `git gc`, which packs loose objects and prunes unreachable ones after a grace period, default 14 days.

37.6 What “origin” actually is

■ 37.6.1 PLAIN - in simple words

1. Almost every git user believes something slightly wrong about origin, so let us be blunt.
2. origin is a nickname for a URL, stored in a text file on your own computer.
3. That is the entire definition. There is nothing else to it.

4. It is not the master copy. It is not the real repository. It is not the authority. It does not own your work.
5. The name `origin` is not special to git. It is the name `git clone` happens to use by default. You may rename it to anything.
6. That default is a **convention**, not a **standard**. No specification requires it. It is a habit that became universal.
7. A repository can have zero remotes, one remote, or twenty remotes. All of them are equal. None outranks another.
8. Zero is not a broken state. A repository with no remotes is a completely normal, fully functional git repository. You can commit to it forever.
9. And the URL a remote points at does not have to be on the internet. It can be another folder on the same disk, or a USB stick.

■ 37.6.2 PLAIN – a picture in your head

1. `origin` is an entry in your phone's contact list.
2. The contact is called "Mum". That name means nothing to the phone network. It is a label you chose, stored on your handset, mapping to a number.
3. Rename the contact to "Mother" and nothing anywhere else changes. The phone number still works. Your mother does not notice.
4. Delete the contact entirely and your phone still works perfectly. You simply have to type the number when you want to call.
5. You may have twenty contacts. None of them is in charge of your phone.
6. And your mother has her own contact list, in which you are an entry. Neither list is the true one.
7. That is `origin` exactly. A label on your machine mapping a short name to an address you would otherwise have to type in full.

Where this comparison breaks:

8. A phone contact is only a convenience. A git remote also carries a **refspec**, a rule saying which of their branches map to which of your remote-tracking names. That is real behaviour, not just a label.
9. And in a team, `origin` usually does have authority, because the team agreed it does and the hosting service enforces that agreement. That authority is social and administrative. It is not in the git data model.

■ 37.6.3 PLAIN – a worked example

1. Here is a real `.git/config` immediately after a clone, printed verbatim from a sandbox.

```
[core]
  repositoryformatversion = 0
  filemode = true
  bare = false
  logallrefupdates = true
[remote "origin"]
  url = /tmp/gitdemo/server.git
  fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
  remote = origin
  merge = refs/heads/main
```

2. Look at the [remote "origin"] block. Two lines. A URL and a fetch rule. That is what origin is made of.
3. Now rename it and watch what happens, and what does not.

```
$ git remote rename origin upstream
$ git config --get-regexp '^remote\.|^branch\. '
remote.upstream.url /tmp/gitdemo/server.git
remote.upstream.fetch +refs/heads/*:refs/remotes/upstream/*
branch.main.remote upstream
branch.feature/offline-work.remote upstream
```

4. Git rewrote the config, renamed the ref directory from refs/remotes/origin/ to refs/remotes/upstream/, and updated every branch that referred to it.
5. No commit changed. No hash changed. The server was never contacted. It was a local text edit plus a directory rename.
6. Now remove every remote and keep working.

```
$ git remote remove origin
$ git remote
(empty: no remotes at all)
$ git commit -m "Commit in a repo with no remotes at all"
$ git log --oneline -1
4e2462c Commit in a repo with no remotes at all
```

7. Zero remotes. Commit succeeded. That is the proof that origin is optional.

■ 37.6.4 PLAIN – what is really happening inside

1. A remote is three things stored together in .git/config.
2. **A name.** The text between the quotes in [remote "origin"]. Arbitrary.
3. **A URL.** Where to reach it. This may be https://..., git@host:path for SSH, ssh://..., git://..., or a plain filesystem path.
4. **A refspec.** The line fetch = +refs/heads/*:refs/remotes/origin/* reads: take everything under refs/heads/ on their side, and store it under refs/remotes/origin/ on mine. The leading + means overwrite even if the change is not a fast-forward.
5. Optionally a **pushurl**, if you fetch from one place and push to another.
6. Separately, each branch may record an **upstream**: the [branch "main"] block saying which remote and which branch it pairs with. This is what lets you type git push with no arguments.
7. When you run a remote command, git looks up the name, finds the URL, chooses a transport based on the URL scheme, and connects. Chapter 40 covers the transports and their ports.
8. Everything before the connect step is reading a text file.
9. The honest version: people say “push to origin” as though pushing were sending to a superior. It is not. It is one repository asking a peer to update some of its refs, and the peer may refuse. Chapter 40 covers refusal.

■ 37.6.5 TECHNICAL – the engineer’s version

1. Config precedence, highest wins: command line, then .git/config, then ~/.gitconfig or ~/.config/git/config, then /etc/gitconfig. Verify with git config --list --show-scope --show-origin.

```
system file:/etc/gitconfig filter.lfs.required=true
global file:/root/.gitconfig user.name=Shikhar
global file:/root/.gitconfig user.email=shikhar@example.com
local file:./git/config remote.origin.url=...
```


2. Remote management commands, all of which are pure local config edits except the last two.

Command	Touches network
<code>git remote -v</code>	no
<code>git remote add</code>	no
<code>git remote rename</code>	no
<code>git remote remove</code>	no
<code>git remote set-url</code>	no
<code>git remote show origin</code>	yes
<code>git remote update</code>	yes

3. `git remote show origin` contacts the server, which surprises people. Use `git remote show -n origin` to stay offline and read only cached data.
4. A repository may hold many remotes and they are peers in the data model. The common three-remote fork workflow is:

```
$ git remote -v
mirror /tmp/gitdemo/mirror.git (fetch)
mirror /tmp/gitdemo/mirror.git (push)
origin /tmp/gitdemo/server.git (fetch)
origin /tmp/gitdemo/server.git (push)
```

5. Naming conventions widely used but not required by git: `origin` for the place you cloned from, `upstream` for the original project when `origin` is your fork, `fork` for a personal copy. All are **convention**.
6. The default remote name is set by `clone.defaultRemoteName`, added in git 2.30, January 2021. Setting it to `upstream` makes `git clone` name it that.
7. A remote URL of a local path is fully supported and often useful: a USB drive, a network mount, or a bare repository elsewhere on the same disk. The sandbox examples in this chapter use exactly that.
8. There is no field anywhere in git's data model marking one repository as canonical. Authority is entirely external: server permissions, branch protection rules and team agreement.

■ 37.6.6 WORDS – remember these

1. Remote — a saved nickname for another repository's address — a named config section holding a URL and one or more refsspecs.
2. `origin` — the default nickname `git clone` gives the place you cloned from — a conventional remote name with no special meaning to git.
3. Refspec — the rule mapping their branch names to yours — a `+src:dst` pattern, typically `+refs/heads/*:refs/remotes/origin/*`.
4. Upstream — the branch yours is paired with — the `branch.<name>.remote` and `branch.<name>.merge` config pair, shown as `@{upstream}`.
5. Bare repository — a repository with no checked-out files — what a server holds, so nobody's working tree is disturbed by a push.
6. Peer — an equal repository, not a superior — the correct mental model for every remote in a distributed VCS.

37.7 Remote-tracking refs and staleness

■ 37.7.1 PLAIN – in simple words

1. This section is the reader's own case, so we will be very careful.
2. During the outage, their machine reported `origin/main` as `0a95cc8`.
3. It is natural to read that as "the server's main branch is at `0a95cc8` right now". That reading is wrong.
4. The correct reading is: "the last time this computer successfully talked to that server, its main branch was at `0a95cc8`".
5. `origin/main` is a **memory**, not a **measurement**. It is a note your machine wrote to itself, at some past moment, and has not revised since.
6. It lives in a file on your disk. Nothing about it is live. Git will keep reporting that value cheerfully, forever, with the network cable cut.
7. And git will not warn you. There is no "possibly stale" marker in the normal output. It looks exactly the same when it is one second old and when it is six weeks old.
8. This is not a bug. It is the only thing a distributed system can do. Asking the server on every command would be exactly the centralized design git was built to escape.
9. So git gives you a fast, honest, local answer to a slightly different question than the one you thought you asked.

■ 37.7.2 PLAIN – a picture in your head

1. You keep a note on your fridge that says: "Priya's flight lands at 18:40."
2. You wrote it on Tuesday, after speaking to Priya.
3. On Friday you look at the fridge and read 18:40. The note has not changed.
4. But the note is not the flight. It is what you were told on Tuesday. The flight may have moved to 21:10 and nobody has told the fridge.
5. The fridge cannot be wrong, because it is not claiming to know the flight. It is claiming to record what you wrote. It does that perfectly.
6. To learn the real time you must call the airline. That is `git fetch`.
7. If the phone line is down, you can still read the note. You just cannot refresh it. And the note gives you no hint of its own age.

Where this comparison breaks:

8. Git does keep a dated record of when the note changed, in the reflog, so you can find out how old it is. The fridge does not. Section 37.7.5 shows how.
9. And unlike a flight time, the true value of the remote's `main` can only move forward under normal rules. A stale `origin/main` is usually an *underestimate* of what the server now has, not a random wrong value.

■ 37.7.3 PLAIN – a worked example

1. Here is the whole effect, produced in a sandbox by making the remote unreachable, exactly as an outage would.
2. Before the outage: clone, one commit, pushed. Both sides agree.

```
$ git rev-parse main
01d088e5131d972ed904171845efe601d994fea5
$ git rev-parse origin/main
01d088e5131d972ed904171845efe601d994fea5
```

3. Now the remote becomes unreachable, and three commits are made locally.

```
$ git rev-parse main
2c9e7c6d79ab36d4a7ca7b229c750fc0ce5dda53
$ git rev-parse origin/main
01d088e5131d972ed904171845efe601d994fea5
```

4. main moved. origin/main did not, and cannot, because moving it requires hearing from the server.
5. Here is the file itself, which is the point of the whole section.

```
$ cat .git/refs/remotes/origin/main
01d088e5131d972ed904171845efe601d994fea5
```

6. Forty-one bytes on your own disk. That is origin/main. It is not a connection. It is not a query. It is a file.
7. The reader's 0a95cc8 was the first seven characters of exactly such a file's contents, on their own machine, written the last time a fetch or push had succeeded.
8. Git will happily use it offline:

```
$ git log --oneline -1 origin/main
01d088e Initial project layout
```

9. That command ran with the remote unreachable and printed a confident answer.

■ 37.7.4 PLAIN – what is really happening inside

- Now the exact rules. Which commands update a remote-tracking ref and which never do.
- git fetch** updates them. That is its entire job: connect, download new objects, and move refs/remotes/<remote>/* to match what the server said.
- git pull** updates them, because a pull is a fetch followed by a merge or rebase. The fetch part does the updating.
- git push** updates them, for the branches it pushed. If your push succeeds, git knows the server now holds what you sent, so it moves the matching remote-tracking ref locally. This is a small but important detail.
- git remote update** and **git fetch --all** update them for all remotes.
- Nothing else does. Not **git commit**, not **git status**, not **git log**, not **git branch -vv**, not **git merge**, not **git rebase**, not **git checkout**.
- That last group includes every command that displays “ahead 3, behind 2”. Those numbers are computed by comparing two local refs. Both are local, so both can be stale.
- So “your branch is behind origin/main by 2 commits” means “behind the value I last heard, which was some time ago”. It is not a live comparison.
- Here is the sequence in a diagram.

your disk	the server
-----	-----
refs/heads/main	refs/heads/main
moved by: commit,	moved by: pushes
merge, rebase, reset	from anyone
refs/remotes/origin/main	
moved ONLY by:	
fetch, pull, push	
otherwise frozen <-- the reader's 0a95cc8	

- The left column can change every minute while you work. The middle entry changes only when a network operation succeeds. During an outage it is frozen by definition.

■ 37.7.5 TECHNICAL – the engineer’s version

- Storage: remote-tracking refs live at `.git/refs/remotes/<remote>/<branch>` when loose, or as `refs/remotes/<remote>/<branch>` lines in `.git/packed-refs` once packed. Both are read the same way; loose wins on conflict.
- They are ordinary refs in a reserved namespace. Git forbids committing onto them, which is why `git checkout origin/main` puts you in detached HEAD rather than on a branch.
- How to tell how stale it is.** The reflog for a remote-tracking ref records every update with a timestamp. Real output:

```
$ git reflog show origin/main --date=iso
d39bbb7 refs/remotes/origin/main@{2026-08-13 03:12:21 +0000}: update by push
01d088e refs/remotes/origin/main@{2026-08-13 03:09:20 +0000}: update by push
```

- The top line’s timestamp is the last moment this machine had confirmed knowledge of that branch. Everything since is unverified.
- Note the reason strings: `update by push` here, `fetch` or `fetch origin` after a fetch. So the reflog also tells you *how* you learned it.
- Other ways to measure staleness:

Command	What it tells you
<code>git reflog show origin/main</code>	when it last moved
<code>stat -f %Sm .git/refs/...</code>	file mtime, macOS
<code>stat -c %y .git/refs/...</code>	file mtime, Linux
<code>git ls-remote origin main</code>	the live value, needs net

- The file modification time is a decent proxy but imperfect: a fetch that finds no change may not rewrite the file, and packing rewrites timestamps. Prefer the reflog.
- `git ls-remote origin main` is the only entry in that table that answers the question you actually wanted. It contacts the server and prints the live value without downloading any objects. It is the correct tool for “is my origin/main current”, and it needs the network.
- `git status` output such as `## main...origin/main [ahead 3]` is computed by `git rev-list --left-right --count main...origin/main`, entirely locally.
- `git fetch --dry-run` still contacts the server. It only skips writing.
- Configuration worth knowing: `fetch.prune` or `git fetch --prune` deletes remote-tracking refs whose upstream branch no longer exists. Without it, `refs/remotes/origin/` accumulates names of branches deleted months ago, which is another form of staleness.
- The honest version: `origin/main` is sometimes described as “the remote branch”. It is not the remote branch. It is a local cache of the remote branch, with no expiry, no freshness metadata in the ref itself, and no invalidation. The remote branch is on the remote.

■ 37.7.6 WORDS – remember these

- Remote-tracking ref — your machine’s note of where their branch was — a ref under `refs/remotes/<remote>/` updated only by fetch, pull or push.
- Stale — out of date without saying so — holding a cached value with no freshness check and no invalidation signal.

3. Fetch — go and get their latest, without changing my work — download objects and update refs/remotes/*, leaving refs/heads/* untouched.
4. Ahead and behind — how many commits each side has that the other lacks — counts from `git rev-list --left-right --count A...B`, computed locally.
5. Prune — forget remote branches that no longer exist — `git fetch --prune`, deleting stale refs/remotes entries.
6. ls-remote — ask the server what it has right now — a network query printing the remote's refs without transferring objects.

37.8 Which git operations need the network

■ 37.8.1 PLAIN – in simple words

1. Here is the whole answer in one line: a git command needs the network if and only if it must read from, or write to, another repository.
2. Everything else is reading and writing files inside `.git`, and files do not need a network.
3. That rule is not a rough guide. It is exact, and you can apply it to a command you have never seen.
4. Ask yourself: does this command need to know something only another machine knows, or change something only another machine holds?
5. If yes, it is a network command. If no, it is local.
6. `git log` reads commits. All commits are local. Local command.
7. `git branch` writes a 41-byte file. Local command.
8. `git merge` combines two commits you already have. Local command.
9. `git fetch` needs objects you do not have. Network command.
10. `git push` must change refs on another machine. Network command.
11. There is no third category and no exceptions worth memorizing, apart from a couple of traps listed at the end.

■ 37.8.2 PLAIN – a picture in your head

1. Think of your repository as a personal notebook and the remote as a friend's notebook in another city.
2. Anything you can do by turning pages in your own notebook is instant and always available. Reading old pages. Copying a page. Rewriting a draft.
3. Anything requiring your friend's notebook needs a phone call. Learning what they wrote. Dictating your pages to them.
4. Nothing about a broken phone stops you turning your own pages.
5. And notice: dictating a page to your friend needs a call, but *writing* the page did not. The writing already happened, in your own notebook, days ago.
6. That separation between writing and sending is the whole distributed idea.

Where this comparison breaks:

7. A few git commands look local but quietly place a call. `git remote show origin` does. `git submodule update` does. In a partial clone, `git log -p` does. These are the traps, and 37.8.5 lists them.

■ 37.8.3 PLAIN – a worked example

1. Every one of these ran successfully in a sandbox with the remote made unreachable, so the outage was real, not simulated by a flag.

```

git branch -a          rc=0  * feature/offline-work
git tag v0.1           rc=0
git diff HEAD~1 --stat rc=0  README.md | 1 +
git log --oneline -3   rc=0  d39bbb7 Committed during the outage
git blame -L 1,2 file  rc=0  ^01d088e (Reader 2026-08-13 ...)
git reflog -3          rc=0  d39bbb7 HEAD@{0}: commit: ...
git switch main        rc=0
git merge feature/...  rc=0
git bisect start/good  rc=0  Bisecting: 2 revisions left to test

```

2. Branching, tagging, diffing, logging, blaming, reflogging, switching, merging and bisecting. All with no reachable remote.
3. And in the same session, at the same second, these failed:

```

$ git fetch
fatal: Could not read from remote repository.

$ git ls-remote
fatal: Could not read from remote repository.

```

4. The failures are immediate and specific. Git did not hang or half-work. It attempted a connection, failed, and told you.

■ 37.8.4 PLAIN – what is really happening inside

1. The definitive classification. Local commands first.

Command	Local	What it touches
git init	yes	creates .git
git add	yes	index
git commit	yes	objects, refs, logs
git status	yes	index, working tree
git diff	yes	index, objects
git log	yes	objects
git branch	yes	refs
git checkout	yes	index, working tree
git switch	yes	index, working tree
git restore	yes	index, working tree
git merge	yes	objects, refs, index
git rebase	yes	objects, refs, index
git reset	yes	refs, index
git stash	yes	objects, refs
git tag	yes	refs, objects
git blame	yes	objects
git bisect	yes	objects, refs
git cherry-pick	yes	objects, refs
git reflog	yes	logs
git gc	yes	objects
git show	yes	objects
git worktree	yes	.git, filesystem

2. Now the network commands.

Command	Direction	Purpose
git clone	in	copy a whole repository
git fetch	in	download new objects, refs
git pull	in	fetch then merge or rebase
git push	out	send objects, move their refs
git ls-remote	in	read their refs only
git remote show	in	query their state
git submodule update	in	clone or fetch sub-repos
git request-pull	in	reads the remote to compare
git archive -remote	in	ask server to build a tarball

- Why the line falls exactly there. Local commands read and write four things: the object database, the refs, the index, and your working files. All four are directories on your disk.
- Network commands do exactly one extra thing: they move objects and ref values between two object databases over a transport.
- That is why the split has no fuzzy middle. A git command is a manipulation of a local file store, or it is a synchronization between two file stores.
- The commit itself is proof. `git commit` writes some objects, moves one ref, appends to one log file, and stops. It has no idea a remote exists.

■ 37.8.5 TECHNICAL - the engineer's version

- The traps, stated precisely, because each one has caught working engineers.
- git pull is not git fetch plus nothing.** It is fetch plus merge, or fetch plus rebase if `pull.rebase` is true. Offline it fails at the fetch and does not perform the merge.
- git remote show origin contacts the server.** `git remote show -n origin` does not, and prints cached data only.
- git fetch --dry-run contacts the server.** Dry run means it does not write refs, not that it does not connect.
- In a partial clone, previously local commands become remote.** With `--filter=blob:none`, commands needing historical file content, such as `git log -p`, `git blame` on old commits, or `git checkout` of an old tag, fetch blobs on demand and fail offline.
- Submodules.** `git submodule update --init` clones each submodule. Offline it fails, even though the parent repository's own history is complete.
- Git LFS.** Large File Storage replaces big files with pointer text and fetches the real bytes on checkout. A repository using LFS is not fully offline-capable for those files. LFS was released by GitHub in April 2015.
- Credential helpers and signing.** A commit signed with a hardware key or an SSH agent may need that device present, though not a network. A commit signed via a remote signing service does need one.
- Hooks.** A pre-commit hook that runs a linter which downloads packages turns a local command into a network one. That is your hook's doing, not git's.
- Measured cost difference, real numbers from the sandbox on the git project repository, so you can see why the split matters for speed as well as availability.

Operation	Class	Time
git rev-parse HEAD	local	4 ms
git log --oneline (85k)	local	928 ms

Operation	Class	Time
git checkout HEAD~100 - .	local	119 ms
git ls-remote origin	remote	751 ms
git fetch origin (no change)	remote	456 ms

- Walking 85,263 commits locally took 928 ms. Asking the server a single question over a fast link took 751 ms. On the reader's link that day, it would never have completed at all.
- Exit code discipline: remote commands fail with a non-zero status and a `fatal:` message on `stderr`. Scripts should check status rather than parse text, because the messages are not a stable interface.

■ 37.8.6 WORDS - remember these

- Local operation — something git does entirely on your own disk — a command touching only the object database, refs, index and working tree.
- Remote operation — something needing another repository — a command opening a transport connection to synchronize objects or refs.
- Transport — the way two repositories talk — HTTPS, SSH, the git protocol on port 9418, or a local filesystem path.
- Dry run — do the checking but not the writing — `--dry-run`, which for fetch and push still performs the network connection.
- Submodule — another repository nested inside this one — a gitlink entry recording a commit ID, with the sub-repository cloned separately.
- Git LFS — a system for keeping huge files out of history — Large File Storage, storing pointers in git and content on a separate server.

37.9 The reader's outage, worked through

■ 37.9.1 PLAIN - in simple words

- Now we put the whole chapter onto the reader's actual day.
- They were in a flat in India, on home broadband, on a Mac. Their router was at 192.168.0.1 and their DNS resolver was 1.1.1.1.
- github.com resolved to 20.207.73.82, an address in a Microsoft-owned range, because GitHub has been owned by Microsoft since 2018 and fronts traffic through Microsoft's network edge.
- `curl -v https://github.com` printed `Trying 20.207.73.82:443...` and then timed out after 15 seconds with no response at all. No refusal, no error message from any router. Silence.
- The same site loaded instantly on mobile data on the same phone, which showed the site itself was fine and the path from that flat was not.
- So: GitHub was reachable from the world and unreachable from that sofa.
- And the reader kept working for the whole outage. They edited files, ran `git add`, ran `git commit`, created branches, committed again.
- Every one of those commands succeeded. Not partially. Not queued for later. Fully succeeded, with real commits and real hashes.
- The reason is now, we hope, boring: none of those commands involved another computer. They wrote files into `.git` on that Mac's own disk.
- Later, connectivity returned. One `git push` sent everything that had piled up, in a single operation.

■ 37.9.2 PLAIN – a picture in your head

1. Think of a writer in a cottage during a storm that has taken out the phone line.
2. They cannot post anything. They cannot receive anything. The postbox at the end of the lane may as well not exist.
3. But the desk still works. The pen still works. The filing cabinet still works. They write chapters four, five and six, date each one, and file them.
4. When the line comes back they walk to the postbox once and post all three chapters together.
5. Nothing was lost, nothing had to be redone, and nothing had to be remembered in their head rather than on paper.
6. The one real cost is that during the storm they did not know what their co-author had been writing. They found that out afterwards, and might then have had to reconcile.

Where this comparison breaks:

7. The writer knows the line is down. Git does not always make it obvious, because `git status` and `git log` behave identically offline and online. The reader's `origin/main` looked like a fact and was a memory.
8. And posting three chapters at once is not always safe in git. If the co-author moved `main` while you were offline, your push may be rejected, and you must fetch and reconcile. That is Chapter 40.

■ 37.9.3 PLAIN – a worked example

1. The full sequence, with the reader's own values where known, reconstructed step by step.

TIME	EVENT	origin/main	local main
-----	-----	-----	-----
t0	last successful fetch or push	0a95cc8	0a95cc8
t1	network path to GitHub goes dark	0a95cc8	0a95cc8
t2	edit files, git add, git commit	0a95cc8	(new #1)
t3	git switch -c feature/...	0a95cc8	(new #1)
t4	more commits on the branch	0a95cc8	(new #1)
t5	git status: "ahead 1" <- cached	0a95cc8	(new #1)
t6	git fetch -> fatal, no route	0a95cc8	(new #1)
t7	connectivity returns	0a95cc8	(new #1)
t8	git push origin main	(new #1)	(new #1)
t9	git push -u origin feature/...	(new #1)	(new #1)

2. The `origin/main` column is frozen from t0 to t8. That column is a file on the reader's disk, and only a successful network operation writes to it.
3. Here is the equivalent moment reproduced in a sandbox, with real output.

```
before push: origin/main = 01d088e5131d972ed904171845efe601d994fea5
before push: main       = d39bbb76dc18fd2151d333cbf82d9405449342f8
commits waiting to be sent: 5

$ git push origin main
To /tmp/gitdemo/server.git
    01d088e..d39bbb7  main -> main

after push:  origin/main = d39bbb76dc18fd2151d333cbf82d9405449342f8
```

4. Five commits, accumulated with no network, sent in one operation, and only then did `origin/main` move.
5. And the second branch, created entirely offline, published in one go:

```
$ git push -u origin feature/offline-work
To /tmp/gitdemo/server.git
 * [new branch] feature/offline-work -> feature/offline-work
branch 'feature/offline-work' set up to track
'origin/feature/offline-work'.
```

■ 37.9.4 PLAIN – what is really happening inside

1. Let us be precise about what each offline command wrote.
2. `git add README.md` read the file, computed its SHA-1, wrote a blob object into `.git/objects` if that content was new, and updated `.git/index`.
3. `git commit -m "..."` built a tree object from the index, wrote a commit object naming that tree and the previous commit as parent, moved `.git/refs/heads/main` to the new commit's hash, and appended two reflog lines.
4. Total external dependencies of that sequence: none. Total network packets: zero.
5. `git switch -c feature/offline-work` wrote one new 41-byte file at `.git/refs/heads/feature/offline-work` and rewrote `.git/HEAD` to point at it. Two small writes.
6. Now what was **not** possible, stated honestly, because this matters as much as what was.
7. They could not fetch anyone else's new work. If a colleague pushed during those hours, the reader's machine had no way to learn it and no way to include it.
8. They could not know whether anyone had pushed at all. `origin/main` said `0a95cc8` and would have said `0a95cc8` if fifty commits had landed.
9. They could not open a pull request, review one, comment, merge one, or read an issue. All of those are GitHub features reached over the network, not git features. Section 37.11 draws that line.
10. They could not see continuous integration results, because those live on GitHub's servers.
11. They could not verify that their branch would merge cleanly into the real current main, only into the main they last saw.
12. So the honest summary: **authoring** was fully available, and **coordination** was fully unavailable. Git splits those two cleanly, and that split is the single most useful thing to remember from this chapter.

■ 37.9.5 TECHNICAL – the engineer's version

1. Evidence classification for that session, keeping proof separate from inference, as this book does everywhere.
2. **Proven by observation.** DNS resolution succeeded, returning `20.207.73.82`. A TCP connection to port 443 at that address received no response of any kind within 15 seconds. The same request over a different access network succeeded immediately.
3. **Also proven.** Local git commands returned exit status 0 and produced new objects and refs during the same period.
4. **Suggested, not proven.** That traffic was being silently discarded somewhere on the path between the reader's ISP and Microsoft's edge. The absence of any RST and any ICMP unreachable is consistent with a silent drop and inconsistent with the server being down or actively refusing.
5. **Not established.** Which party dropped it, or why. Traceroute showed * * * from hop 13 onward, but many routers suppress ICMP time-exceeded replies, so silence at the end of a trace proves nothing on its own.
6. What the git behaviour adds as evidence: the failure was at the transport layer, not at git's application layer. `git fetch` failing with `Could not read from remote repository after a long stall`

is what a TCP connect timeout looks like from inside git.

7. Recovery pattern for this situation, in order:

Step	Command	Why
1	<code>git status</code>	see local state
2	<code>git log --oneline --graph</code>	see what accumulated
3	<code>git fetch --all</code>	refresh the caches
4	<code>git log origin/main..main</code>	what you still owe
5	<code>git log main..origin/main</code>	what you missed
6	<code>git push</code>	publish

8. Step 3 is the one to insist on. Until a fetch succeeds, every “ahead” and “behind” number on the screen is arithmetic on a stale cache.
9. Note the asymmetry between steps 4 and 5. `origin/main..main` lists commits you have that the remote lacked. `main..origin/main` lists the reverse. The two-dot syntax is not symmetric and the order matters.
10. If the push had failed midway with `send-pack: unexpected disconnect while reading side-band packet`, the correct response is not to guess. It is to re-query the server, because a disconnect while reading the response tells you nothing about whether the write landed. Chapter 40 covers that case in full.
11. Long-outage tip: `git bundle create out.bundle main` writes a single file containing commits, which can travel on a USB stick and be cloned or fetched from on another machine. It is the offline transport of last resort and it is built into git.

■ 37.9.6 WORDS - remember these

1. Outage — a period when the network path does not work — loss of reachability between two endpoints, distinct from either endpoint being down.
2. Silent drop — packets vanish with no error returned — a discard with no RST and no ICMP unreachable, leaving the sender to time out.
3. Accumulate — pile up locally until you can send — build a series of local commits with no intervening synchronization.
4. Authoring versus coordination — making your own work versus agreeing with others — the local-capable and network-required halves of a git workflow.
5. Bundle — a whole repository transfer in one file — `git bundle`, a pack plus refs, usable as a remote for clone and fetch.
6. Two-dot range — commits on one side and not the other — `A..B` meaning reachable from B but not from A, and not symmetric.

37.10 The trade-offs of the distributed model

■ 37.10.1 PLAIN - in simple words

1. This book does not sell things, so here are the costs of the distributed model as honestly as the benefits.
2. The benefits first, briefly. You can work with no network. Operations are fast because they are local. Every clone is a backup of the whole history. No single machine’s failure loses the project.
3. Now the costs.

4. **The mental model is harder.** In Subversion there is one history and one revision number. In git there are as many histories as there are clones, and they may disagree, and you must reason about which is which.
5. **You cannot lock a file.** In some teams, a designer needs to say “nobody else touch this 400 MB scene file while I work on it”. Git has no way to enforce that, because there is no central place to hold the lock.
6. **Files that cannot be merged are a real problem.** Two people edit the same image, the same 3D model, the same compiled asset. Git will detect the conflict and then be unable to help. One person’s work must be thrown away.
7. **Whole-history clones are large.** For source code that is fine. For a project with twenty years of large binary assets it can be hundreds of gigabytes, and every new team member downloads all of it.
8. **There is no single source of truth** unless the team agrees on one. Git does not provide it, so the team, the server permissions and the process must.

■ 37.10.2 PLAIN – a picture in your head

1. Distributed version control is like everyone owning a car instead of sharing one bus.
2. You go when you like. You do not wait for the timetable. If the bus company strikes, you still get to work.
3. But now there are twelve cars to park, twelve to fuel, and no driver telling everyone which route is correct. If two of you must arrive together, you have to arrange it yourselves.
4. And a bus is genuinely better for some journeys. If the whole team must move one enormous object, one vehicle with one driver is the right design.
5. Nobody thinks the bus company is foolish for existing. They are solving a different problem.

Where this comparison breaks:

6. Cars and buses cost about the same to run per person. Git and Perforce do not: git is free software and Perforce is licensed per user. Cost is part of why git spread, and it should be named.
7. And unlike vehicles, git and Perforce can coexist. Some studios run Perforce for art assets and git for engine code in the same building.

■ 37.10.3 PLAIN – a worked example

1. Concretely, why a game studio chooses Perforce. Take a project with 800 GB of assets and 300 people.

Problem	Git	Perforce
New joiner setup	clone all history	sync latest only
Lock a scene file	not possible	exclusive checkout
Merge a 2 GB binary	impossible	prevented by lock
Check out one folder	awkward	native, standard
Work offline	full	very limited
Cost	free	per-seat licence

2. Rows one to four favour Perforce for this workload. Row five favours git. Row six favours git.
3. For a 300-person studio where everyone sits in one building with a fast local network, row five is worth little and rows one to four are worth a great deal.

4. For a distributed open-source project with 4,000 contributors on five continents who have never met, the weighting inverts completely.
5. Neither team is confused. They are optimizing different things, and the correct engineering answer is “it depends on the workload”, which is an unsatisfying sentence that happens to be true.

■ 37.10.4 PLAIN – what is really happening inside

1. The locking problem deserves precision, because it is the clearest structural limit of the distributed model.
2. A lock is a statement that everyone must agree on, at the same time. “Only Priya may edit `level3.blend` right now.”
3. Agreement at the same time requires a single place that answers the question. That is exactly what a distributed system does not have.
4. So the absence of file locking in git is not laziness or a missing feature. It is a direct consequence of the architecture. You cannot have both.
5. Git LFS adds a lock command, and it works by contacting the LFS server. That is the same admission: to lock, you must centralize something.
6. The binary merge problem is different and is about file formats. Git merges text line by line. A `.psd`, a `.fbx` or a `.docx` has no lines, so there is nothing to merge.
7. Git will happily store binary files and version them. It just cannot combine two divergent edits, and it cannot compress them well, so the repository grows roughly linearly with every binary change.
8. The size problem compounds because git history is immutable. A 500 MB file committed once by mistake in 2019 is in every clone forever, unless someone rewrites history and every clone is re-made.

■ 37.10.5 TECHNICAL – the engineer’s version

1. Honest comparison across the main systems still in use in 2026.

System	Model	Typical domain
Git	distributed	source code, everywhere
Perforce P4	centralized	games, chips, film
Subversion	centralized	legacy enterprise
Mercurial	distributed	some large monorepos
Plastic SCM	centralized+	Unity game teams

2. Perforce, founded 1995 by Christopher Seiwald, remains the standard in game development, semiconductor design and visual effects. It received the 2019 CEDEC Award for Engineering, an award from the Japanese game industry, which tells you where its users are.
3. Perforce’s exclusive checkout is implemented with the `+l` file type modifier, which makes the server grant a write lock to one client at a time.
4. Perforce’s other decisive feature for these teams is the **client workspace view**, a mapping that lets a user sync one subdirectory of a 5 TB depot. Git has partial workarounds, sparse-checkout and partial clone, but they are less complete and less familiar.
5. Where experts disagree: some argue git plus LFS plus sparse-checkout now covers the game studio case and Perforce persists mainly through inertia and tooling lock-in. Others argue locking and workspace views are structural and git will never match them. Both positions are held by serious practitioners and the disagreement is genuine.

6. Scaling work on the git side is real and ongoing. Microsoft's VFS for Git, released in 2017 as GVFS, and later Scalar, merged into git itself in version 2.38 in October 2022, exist to make very large repositories usable.
7. Mitigations for the size problem, all mainstream as of 2026.

Technique	Reduces	Cost
Git LFS	clone size	needs LFS server
partial clone	clone size	fetches on demand
shallow clone	clone size	loses history
sparse-checkout	working tree	not history
commit-graph file	traversal time	none, local
multi-pack index	lookup time	none, local

8. Note which column those mitigations sit in. The last two are free local speedups. The first three all trade offline capability or history for size. That trade is the theme of this section.
9. `git maintenance start`, added in git 2.31, March 2021, schedules background packing, commit-graph writing and prefetching, and is the modern replacement for manually running `git gc`.

■ 37.10.6 WORDS – remember these

1. File locking — one person at a time may edit a file — a server-granted exclusive write lock, unavailable in a purely distributed model.
2. Binary file — a file with no lines to compare — content without a text structure, so it cannot be three-way merged or delta-compressed well.
3. Monorepo — one repository holding many projects — a single history covering multiple products, common at very large companies.
4. Sparse checkout — having only part of the project in your folder — a cone-mode pattern list restricting which paths are materialized.
5. Source of truth — the copy everyone agrees is correct — an external convention plus server permissions, not a property of git itself.
6. Depot — Perforce's name for the server-side store — the centralized repository from which client workspaces are mapped and synced.

37.11 Git is not GitHub

■ 37.11.1 PLAIN – in simple words

1. This confuses more beginners than anything else in the subject, so it goes in plainly.
2. **Git** is a program on your computer. You install it. It makes and reads files in a `.git` directory. It is free software, first released in 2005.
3. **GitHub** is a company that runs servers. You make an account. It stores copies of repositories and adds features around them. It launched on 10 April 2008 and Microsoft bought it in 2018.
4. Git does not need GitHub. GitHub needs git.
5. You can use git for thirty years and never make a GitHub account.
6. You can host repositories on GitLab, on Bitbucket, on a Raspberry Pi in a cupboard, on a USB stick, or on a company server. Git does not care.
7. The names are similar because GitHub chose a name based on git. That naming choice has cost the world a great deal of confusion.
8. Here is the sharpest possible test, and the reader lived it: during the outage, git kept working and GitHub did not.

9. That is not a coincidence. It is the boundary made visible.

■ 37.11.2 PLAIN – a picture in your head

1. Git is a **word processor**. GitHub is a **print shop and a noticeboard**.
2. The word processor runs on your laptop. It works on a train. It never asks anyone's permission.
3. The print shop takes your document, keeps a copy, lets others read it, lets people leave notes on it, and runs a checking machine over it.
4. If the print shop closes, your word processor is unaffected and your document is still on your laptop.
5. If your laptop dies, the print shop still holds the copy you dropped off.
6. Neither one is the other. They cooperate through one narrow channel: you carrying documents in and out.

Where this comparison breaks:

7. A print shop holds a dumb copy. A GitHub repository is a real git repository and could replace yours entirely, which is why it feels authoritative.
8. And some things blur the line deliberately. GitHub Actions can commit to your repository. The GitHub web editor makes real commits. But those are GitHub running git on its own machines, which does not change the boundary.

■ 37.11.3 PLAIN – a worked example

1. The division of labour, feature by feature.

Feature	Git	GitHub
commit, branch, merge	yes	no
full local history	yes	no
pull requests	no	yes
code review comments	no	yes
issues and projects	no	yes
CI via Actions	no	yes
access control, teams	no	yes
releases page	no	yes
hosting a remote	partly	yes

2. The partly in the last row: git includes `git daemon` and can serve over SSH, so a plain server with git installed is a perfectly good remote. It simply has no web interface.
3. During the reader's outage, the left column all worked and the right column all failed, because every item in the right column is an HTTPS request to a server in that unreachable direction.
4. The clearest evidence in that session: plain git operations behaved as described in 37.8, and GitHub API calls failed. Same laptop, same minute, different dependency.

■ 37.11.4 PLAIN – what is really happening inside

1. What git actually is: a set of programs, mostly C, plus some shell and Perl. On a typical install it is a git binary and a directory of helper executables, around a hundred subcommands.
2. It speaks two wire protocols for talking to other repositories: the smart HTTP protocol and the SSH protocol, plus the older `git://` daemon protocol on port 9418. Chapter 40 covers all three.

3. What GitHub actually is: a large web application and API, backed by many servers that hold bare git repositories, wrapped in authentication, permissions, a review system, an issue tracker and a CI system.
4. When you `git push` to GitHub, `git` on your laptop opens a connection, runs the standard protocol, and a program on GitHub's side receives the pack. That receiving program is `git-receive-pack`, part of `git`.
5. So the push path is pure `git` at both ends. GitHub's own features do not participate.
6. When you open a pull request, none of that happens. Your browser or the `gh` command makes an HTTPS API call to GitHub. There is no `git` protocol involved and nothing is written to your `.git`.
7. That is why one worked offline and the other did not. They are different systems reached by different means.
8. The honest version: people say "push to GitHub" and "my code is on GitHub" as if GitHub were where the code lives. Your code lives on your disk. GitHub has a copy, which is extremely useful and is not the original.

■ 37.11.5 TECHNICAL - the engineer's version

1. Git the software: created 2005, maintained by Junio Hamano since 26 July 2005, licensed under GPL version 2, current release 2.55.0 dated 29 June 2026. Written mainly in C.
2. GitHub the service: launched 10 April 2008 by Tom Preston-Werner, Chris Wanstrath, P. J. Hyett and Scott Chacon. Microsoft announced acquisition on 4 June 2018 for 7.5 billion US dollars in stock. The deal closed on 26 October 2018.
3. Alternatives, all of which speak the same `git` protocols: GitLab from 2011, Bitbucket from 2008, Gitea, Forgejo, Sourcehut, AWS CodeCommit, Azure Repos, and a plain SSH account on any Linux box.
4. Interface boundary in one table.

Thing you do	Protocol	Offline
<code>git clone</code> , <code>fetch</code> , <code>push</code>	<code>git</code> over HTTPS/SSH	needs net
<code>git commit</code> , <code>log</code> , <code>merge</code>	none	works
open a pull request	GitHub REST/GraphQL	needs net
<code>gh pr list</code>	GitHub REST/GraphQL	needs net
view Actions run	GitHub web/API	needs net

5. The `gh` command line tool is a GitHub client, not a `git` client. It has nothing to do with the `git` binary and fails independently of it. During the reader's outage this distinction was visible directly.
6. GitHub-specific concepts that do not exist in `git` at all: pull request, fork relationship, issue, label, milestone, review, required status check, branch protection rule, `mergeStateStatus`, Actions workflow, release, Gist.
7. Git concepts GitHub merely displays: `commit`, `tree`, `blob`, `tag`, `branch`, `ref`, `merge`, `rebase`, `cherry-pick`, `reflog`. GitHub cannot show you a `reflog`, because a `reflog` is local to a working repository.
8. Chapter 40 covers remotes, transports, authentication, tokens and scopes, and the exact division of labour when a push both moves refs and triggers a GitHub workflow.

■ 37.11.6 WORDS - remember these

1. Git — the version control program on your machine — a distributed VCS implemented as a suite of C programs, GPLv2, first released 2005.

2. GitHub — a company hosting git repositories with extra features — a web service owned by Microsoft since 2018, offering pull requests, issues and CI.
3. Forge — the general word for a code hosting service — GitHub, GitLab, Bitbucket, Gitea and similar, all built around bare git repositories.
4. Pull request — a request to merge, with discussion — a forge feature, not a git feature, implemented over the forge's own API.
5. git-receive-pack — the program on the server that accepts a push — the receiving half of git's push protocol, run by the hosting service.
6. API — a way for programs to talk to a service — here, GitHub's REST and GraphQL interfaces over HTTPS, entirely separate from the git protocol.

37.12 Getting started properly

■ 37.12.1 PLAIN - in simple words

1. There are exactly two ways a git repository comes to exist on your machine.
2. **git init** makes a new empty one, here, now, from nothing. No network.
3. **git clone <url>** copies an existing one, including all its history, and sets up a remote called origin pointing back at where it came from.
4. That is the complete list. Everything else is one of these two, possibly done by a tool on your behalf.
5. Before your first commit, tell git who you are. It records a name and an email address in every commit, permanently.
6. That email does not have to be real or reachable. Nothing sends mail to it. It is a label. But it is baked into the commit hash and cannot be quietly changed later without changing every hash after it.
7. Then create a `.gitignore` file listing things git should not track: build outputs, dependency folders, editor files, secrets.
8. And learn one hard fact early, because it catches everybody: `.gitignore` does **not** untrack a file that is already tracked.

■ 37.12.2 PLAIN - a picture in your head

1. `git init` is starting a new notebook. Blank, yours, immediate.
2. `git clone` is photocopying someone's finished notebook, cover to cover, including their crossings-out, and writing their address inside your front cover so you can find them again.
3. `.gitignore` is a note on the front of the notebook saying "do not copy receipts into this book".
4. And here is the catch that the note cannot fix: if you already glued a receipt onto page 4, the note does nothing. The receipt is on page 4. It will be on page 4 in every photocopy anyone ever makes.
5. To stop tracking it you must deliberately remove it, and even then it stays in the old pages, because old pages are never edited.

Where this comparison breaks:

6. A photocopy is a dead copy. A clone is fully alive: it can commit, branch and serve as the source for further clones.
7. And unlike a notebook, you can start with `git init` and later attach a remote, which is the normal path for a project that begins locally.

■ 37.12.3 PLAIN – a worked example

1. The full first-time setup, exactly as it should be typed.

```
git config --global user.name "Shikhar Singh"
git config --global user.email "you@example.com"
git config --global init.defaultBranch main
git config --global pull.rebase false
git config --global core.autocrlf input    # false on Windows
```

2. Then a new project from nothing:

```
mkdir myproject && cd myproject
git init
printf 'node_modules/\n*.log\n.env\n' > .gitignore
git add .
git commit -m "Initial commit"
```

3. That commit exists now, fully, with no account and no network.
4. Now the .gitignore trap, demonstrated with real output from a sandbox. A secret file was committed by mistake, then added to .gitignore.

```
$ git ls-files
.gitignore
app.py
config.env          <- still tracked

$ printf 'CHANGED\n' >> config.env
$ git status --short
 M config.env       <- git is still watching it
```

5. The .gitignore had no effect at all, because ignore rules apply only to files git is not already tracking.
6. The fix, and its limit:

```
$ git rm --cached config.env
$ git commit -m "stop tracking config.env"
$ git ls-files
.gitignore
app.py          <- now untracked

$ git show HEAD~2:config.env
secret-key      <- STILL IN HISTORY, forever
```

7. Read that last line carefully. The secret is out of the current commit and still recoverable from an old one. If it was a real credential, it is compromised and must be rotated. Removing it from history requires rewriting every commit after it, and every clone must be remade.

■ 37.12.4 PLAIN – what is really happening inside

1. `git init` creates the `.git` directory, writes a default config, writes HEAD containing `ref: refs/heads/main`, and installs the sample hooks. It contacts nothing.
2. Note that `refs/heads/main` does not exist yet at that point. HEAD points at a branch that has no file. That is the “unborn branch” state, and it is why `git status` on a fresh repository says “No commits yet”.
3. `git clone` does more. It creates the directory, runs `init`, adds the `[remote "origin"]` config, fetches everything, writes the remote-tracking refs, creates a local branch matching the remote’s default, and checks it out.

4. Real output, so you can see how little it says:

```
$ git clone /tmp/gitdemo/server.git clonedemo
Cloning into 'clonedemo'...
done.
$ git branch -a
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/feature/offline-work
  remotes/origin/main
```

5. Notice `remotes/origin/HEAD -> origin/main`. That records which branch the server said was its default, so `git checkout` of a bare remote name works.
6. `.gitignore` mechanics: on `git add` and on `git status`, git checks each **untracked** path against the ignore rules. Tracked paths skip the check entirely. That is the whole reason for the trap in 37.12.3.
7. Ignore rules come from several files, most specific first: the path's nearest `.gitignore`, then parent directories, then `.git/info/exclude`, then the file named by `core.excludesFile`.
8. `.gitignore` is committed and shared. `.git/info/exclude` is private and never leaves your machine. Use the second for editor droppings that only you produce.

■ 37.12.5 TECHNICAL – the engineer's version

1. Settings worth changing on day one, with what each does.

Setting	Value	Effect
<code>user.name</code>	your name	commit author
<code>user.email</code>	your address	commit author
<code>init.defaultBranch</code>	main	branch on init
<code>pull.rebase</code>	false or true	pull behaviour
<code>push.default</code>	simple	push safety
<code>core.autocrlf</code>	input or false	line endings
<code>fetch.prune</code>	true	drop dead refs
<code>rerere.enabled</code>	true	reuse conflict fixes
<code>diff.algorithm</code>	histogram	better diffs

2. `init.defaultBranch` was added in git 2.28, July 2020. Before that the default was `master`, hard-coded. Git still defaults to `master` if you do not set it, and prints a hint saying so.
3. `push.default = simple` has been the default since git 2.0 in 2014. It pushes the current branch to its upstream of the same name and refuses otherwise, which prevents accidental mass pushes.
4. `core.autocrlf` matters only across operating systems. On Windows, `true` converts to CRLF on checkout and back to LF on commit. On macOS and Linux, `input` or `false`. Better still, commit a `.gitattributes` with `* text=auto` and stop thinking about it.
5. `rerere` stands for “reuse recorded resolution”. With it on, git remembers how you resolved a conflict and applies the same resolution if it sees the same conflict again. It is invaluable during a long rebase.
6. Verify what is set and from where:

```
$ git config --list --show-scope --show-origin
system file:/etc/gitconfig filter.lfs.required=true
global file:/root/.gitconfig user.name=Shikhar
global file:/root/.gitconfig user.email=shikhar@example.com
```

7. `git check-ignore -v --no-index <path>` tells you which rule in which file at which line number is ignoring a path. Without `--no-index` it deliberately ignores tracked files, which is itself a source of confusion.

```
$ git check-ignore -v --no-index config.env
.gitignore:1:config.env config.env
```

8. For removing a secret from all history the current recommended tool is `git filter-repo`, which replaced the much slower `git filter-branch`. The git project has discouraged `filter-branch` since version 2.24, November 2019. Either way, every existing clone becomes invalid and the credential must be rotated regardless.
9. `.gitignore` pattern rules follow `gitignore(5)`: `#` is a comment, a trailing `/` matches directories only, a leading `/` anchors to the `.gitignore`'s own directory, `**` matches across directories, and a leading `!` negates an earlier rule. A negation cannot re-include a file if its parent directory is itself excluded, which is the most common pattern bug.

■ 37.12.6 WORDS – remember these

1. `init` — start a new empty repository here — create a `.git` directory with default config, `HEAD` and hooks, contacting nothing.
2. `Clone` — take a full copy of an existing repository — `init` plus a remote plus a full fetch plus a checkout of the default branch.
3. `.gitignore` — a list of things git should not track — patterns matched against untracked paths only, per `gitignore(5)`, committed and shared.
4. `Untracked file` — a file git knows nothing about — a working tree path with no index entry, eligible for ignore rules.
5. `Tracked file` — a file git is watching — a path present in the index, immune to `.gitignore` until removed with `git rm --cached`.
6. `Unborn branch` — `HEAD` points at a branch with no commits yet — the state after `git init`, before the first commit creates the ref.
7. `rerere` — git remembering how you solved a conflict before — reuse recorded resolution, enabled with `rerere.enabled true`.

37.98 Common wrong ideas

1. Wrong: git needs a server. Right: git needs a directory. `git init`, `git add`, `git commit`, `git branch` and `git merge` all work on a machine that has never been connected to anything.
2. Wrong: `origin` is the real repository. Right: `origin` is a nickname for a URL stored in `.git/config`. It has no authority in git's data model. Rename it, remove it, or have twenty of them.
3. Wrong: `origin/main` tells you where the remote's main branch is. Right: it tells you where it was the last time a fetch, pull or push succeeded on this machine. It is a file on your disk and it never expires.
4. Wrong: cloning downloads the current version of the project. Right: cloning downloads every commit, every version of every file, every branch and every tag, then checks out one of them into your folder.
5. Wrong: git and GitHub are the same thing. Right: git is a program from 2005 that runs on your computer. GitHub is a company from 2008, owned by Microsoft since 2018, that hosts copies and adds pull requests, issues and CI.

6. Wrong: a commit is saved to the cloud. Right: a commit writes objects and one ref into `.git` on your own disk. Nothing leaves your machine until you push.
7. Wrong: `git pull` is the safe way to get up to date. Right: `git pull` is `git fetch` plus a merge or rebase that changes your branch. `git fetch` alone updates your knowledge and changes nothing of yours.
8. Wrong: adding a file to `.gitignore` removes it from the repository. Right: ignore rules apply only to untracked files. An already-tracked file keeps being tracked until `git rm --cached`, and stays in history forever.
9. Wrong: “ahead 3, behind 2” is a live comparison with the server. Right: it is arithmetic between two local refs, one of which is a cache that may be weeks old. Run `git fetch` before believing it.
10. Wrong: a shallow clone is just a smaller clone. Right: a shallow clone has no old history, so `git log`, `git blame` and `git bisect` cannot see past the cut, and deepening requires the network.

37.99 Chapter summary in 20 lines

1. Version control solves six problems: history, undo, blame, parallel work, backup and collaboration. Only the last two need another computer.
2. Dated folders fail because they record copies without recording reasons or relationships, and because two people cannot use them at once.
3. The lineage runs SCCS 1972, RCS 1982, CVS from 1986 and 1990, Subversion from 2000 with 1.0 in February 2004, then git in April 2005.
4. In a centralized system the history lives on a server and your machine holds a working copy, so a commit is a network operation.
5. In a distributed system every clone is a complete repository, so a commit is a local file write. This is why the reader kept committing during the outage.
6. Git exists because BitMover ended the kernel’s free BitKeeper licence in April 2005. Torvalds started on 3 April and git was self-hosting on 7 April.
7. His stated goals were speed, simple design, strong support for non-linear development, fully distributed operation, and handling the Linux kernel.
8. A `.git` directory holds objects, refs, HEAD, config, index, hooks, logs, info, description and sometimes packed-refs. That is the entire repository.
9. A branch is a 41-byte file containing a commit hash. That is why branching is instant, free and offline.
10. In a real clone of the git project, `.git` was 316 MiB against a 60 MiB working tree, holding 418,900 objects and 85,263 commits.
11. Compression is why: about 7,877 MiB of raw object content stored in about 302 MiB, a ratio near 26 to 1, via zlib and pack deltas.
12. Shallow and partial clones trade history or content for size, and both reintroduce the network into operations that were previously local.
13. `origin` is a nickname for a URL in `.git/config`. It is a convention, not a standard, it carries no authority, and a repository may have zero remotes.
14. `origin/main` is a local file recording where you last saw the remote’s main. It is a memory, not a measurement, and it never expires by itself.
15. Only `fetch`, `pull` and `push` update remote-tracking refs. `git reflog show origin/main --date=iso` tells you how stale yours is.
16. Local commands touch only objects, refs, index and working tree. Remote commands synchronize two object databases. The split has no fuzzy middle.

17. During the outage, authoring was fully available and coordination was fully unavailable: no fetching others' work, no pull requests, no CI results.
18. The costs of the distributed model are a harder mental model, no file locking, no useful merging of binaries, large clones, and no built-in truth.
19. That is why game studios, chip designers and visual effects houses still buy Perforce, founded 1995, for its exclusive checkout and workspace views.
20. Git is a program on your computer. GitHub is a company that hosts copies. During the outage git worked and the GitHub API did not. That is the line.

What a Commit Actually Is

38.0 What this chapter gives you

1. You will be able to say what content-addressed storage is: the name of a thing is worked out from the thing itself, so equal content always gets equal names.
2. You will be able to explain what a hash function is, name the five properties that matter, and say why SHA-1 gives 160 bits printed as 40 hexadecimal characters.
3. You will be able to state honestly that SHA-1 was broken in public on 23 February 2017 by Google and CWI Amsterdam, say what git did about it, and say why git's use of it was never purely a security control.
4. You will be able to name git's four object types, blob, tree, commit and tag, describe the exact bytes each one stores, and rebuild any object's hash by hand with one shell command.
5. You will be able to draw the complete object graph of a small repository, every blob, every tree, every commit and every arrow, and say exactly which objects are created and which are reused when one file changes.
6. You will be able to explain why history is a directed acyclic graph and not a line, what a merge commit with two parents looks like inside, and why the tip hash seals everything behind it.
7. You will be able to say what a branch really is, a small text file holding 40 hexadecimal characters, and why that makes branching almost free.
8. You will be able to read and repair the state of HEAD, including detached HEAD, and recover work that looked lost after a hard reset by using the reflog.
9. You will be able to use the plumbing commands that show the machinery: `git cat-file`, `git rev-parse`, `git hash-object`, `git ls-tree`, `git count-objects`, `git verify-pack` and `git rev-list`.
10. You will be able to explain the reader's own observation that `origin/main` was `0a95cc8`, and why a rebased branch inevitably came out with a different hash.

38.1 Content-addressed storage, the central idea

■ 38.1.1 PLAIN – in simple words

1. Normally you give a thing a name and then put content inside it. A folder called `notes` holds whatever you put in it today.
2. Git does the opposite. You give it content, and git works out the name from the content.
3. The name is a long string of letters and digits, produced by feeding every byte of the content through a fixed piece of arithmetic.
4. This is called **content-addressed storage**. The address of a thing is computed from the thing.
5. Two consequences follow, and they are the whole reason git works the way it does.

6. First, the same content always gets the same name. So git can never store the same content twice. It looks up the name, finds it already there, and stops.
7. Second, if anyone changes the content even by one bit, the name no longer matches. So changing content without being noticed is not possible.
8. Everything else in this chapter is a consequence of those two sentences.

■ 38.1.2 PLAIN - a picture in your head

1. Think of a very strange left-luggage office at a railway station.
2. You hand over a bag. The clerk does not give you a numbered ticket chosen from a pile.
3. Instead the clerk weighs the bag, measures it, catalogues every object inside it, and turns all of that into one long code.
4. That code is your ticket. It describes the bag rather than pointing at a shelf.
5. If you hand in an identical bag tomorrow, you get the identical ticket. The clerk says “I already have that one” and stores nothing new.
6. If somebody opens your bag and removes one sock, then re-runs the measurement, they get a different code. Your old ticket no longer matches anything. The swap is visible.

Where this comparison breaks:

7. A real clerk could be lazy and reuse a code for two nearly identical bags. A hash function cannot be lazy. It is arithmetic, and it runs the same way every time.
8. A real bag can be measured many ways. Git measures exactly one way, over exactly the bytes, in exactly one order, so the answer is never a matter of judgement.
9. And a left-luggage ticket does not tell you what is in the bag. A git hash does not either, but it is enough to prove that the bag you get back is the bag you handed in.

■ 38.1.3 PLAIN - a worked example

1. Here is a real repository built for this chapter. Four files were created, three of them holding exactly the same line of text.
2. The command `git ls-tree -r HEAD` lists every file with the name git gave to its content.

```
100644 blob 6fe3e87371433ed417646e3cb2354cfb9cff08d3    a/one.txt
100644 blob c9330452ddba7055160034759c594f78381fd24d    a/other.txt
100644 blob 6fe3e87371433ed417646e3cb2354cfb9cff08d3    b/two.txt
100644 blob 6fe3e87371433ed417646e3cb2354cfb9cff08d3    c/three.txt
```

3. Three files, three different paths, one identical name: 6fe3e87...
4. Now count how many pieces of file content git actually stored:

```
6fe3e87371433ed417646e3cb2354cfb9cff08d3 blob 22
c9330452ddba7055160034759c594f78381fd24d blob 18
```

5. Two. Not four. The three identical files share one stored copy of 22 bytes.
6. Nobody asked for that. Nobody ran a de-duplication tool. It falls out of naming things by their content.
7. Now rename `a/one.txt` to `a/renamed.txt` and commit. The listing changes, but the stored content list does not change at all.

```
100644 blob c9330452ddba7055160034759c594f78381fd24d    a/other.txt
100644 blob 6fe3e87371433ed417646e3cb2354cfb9cff08d3    a/renamed.txt
100644 blob 6fe3e87371433ed417646e3cb2354cfb9cff08d3    b/two.txt
100644 blob 6fe3e87371433ed417646e3cb2354cfb9cff08d3    c/three.txt
```

8. The file name lives in the directory listing, not in the content. Renaming is free.

■ 38.1.4 PLAIN – what is really happening inside

1. When you run `git add file.txt`, git reads every byte of the file.
2. It puts a small header in front of those bytes. The header is the word `blob`, a space, the length in bytes, and a zero byte.
3. It runs the header plus the content through the hash function. Out comes a 160-bit number.
4. It prints that number as 40 hexadecimal characters. That is the object's name.
5. It then looks on disk for a file at `.git/objects/` plus the first two characters, a slash, then the other 38 characters.
6. If that file already exists, git does nothing more. The content is already stored, by definition, because the name proves it.
7. If it does not exist, git compresses the header plus content with `zlib` and writes it there.
8. Reading works in reverse. Given a name, git knows the path, reads the file, decompresses it, and hands back the bytes.
9. Because the file is named after its own content, git can check itself at any time. Re-hash the content, compare with the file name, and any difference is damage or tampering.

■ 38.1.5 TECHNICAL – the engineer's version

1. Git implements a content-addressable object store. The key space is the output of a cryptographic hash over `<type> <size>\0<content>`.
2. The store is also a **Merkle tree**, the structure Ralph Merkle described in his 1979 doctoral work: a tree in which every parent node contains the hash of its children.
3. That is why one hash at the top, the commit hash, transitively covers every byte in the tree below it.
4. Loose objects live at `.git/objects/ab/cdef...`. The two-character sub-directory exists because early filesystems degraded badly with hundreds of thousands of entries in one directory.
5. Here is a real loose object, on disk, byte for byte:

```
00000000 78 01 4b ca c9 4f 52 30 34 61 f0 cc 2b 29 ca 4f
00000010 29 4d 2e c9 cc cf d3 e3 02 00 54 63 07 5f
```

6. That file is 30 bytes. The leading `78 01` is a `zlib` header (RFC 1950, May 1996, by Jean-loup Gailly and Mark Adler). Decompressed it is 22 bytes:

```
b'blob 14\x00Introduction.\n'
```

7. So the stored object is `blob`, space, 14, a NUL byte, then the 14 bytes of file content. The hash is taken over all 22 of those bytes.
8. Deduplication is exact-match only. Git does not find near-duplicates at store time. Similarity is exploited later, inside packfiles, and that is a separate mechanism covered in 38.11.

Property of the store	Value in git
Key derivation	SHA-1 of header+content
Key length	160 bits, 40 hex chars
Loose object path	objects/xx/38-char-rest
Object compression	<code>zlib</code> deflate, RFC 1951
Duplicate handling	write skipped if key exists

9. The honest version: content addressing gives integrity, not secrecy. Anyone with the object store can read every byte. Encryption is a separate concern that git does not provide.

■ 38.1.6 WORDS – remember these

1. Content-addressed storage — the name comes from the content — key space is the image of a hash function over the stored bytes.
2. Blob — a stored piece of file content — a git object of type blob holding raw bytes with no name and no permissions.
3. Deduplication — storing one copy of identical things — exact-match coalescing by hash key at write time.
4. Merkle tree — a tree where each parent names its children by hash — hash tree, described by Ralph Merkle in 1979, giving transitive integrity.
5. zlib — the squeezing format git uses on disk — DEFLATE stream in a zlib wrapper, RFC 1950 and RFC 1951, both May 1996.

38.2 Hashing, and the honest truth about SHA-1

■ 38.2.1 PLAIN – in simple words

1. A **hash function** takes any amount of data and produces a short fixed-size number that stands for it.
2. Five things matter about it.
3. It is **deterministic**: the same input always gives the same output, on any machine, in any year.
4. It has a **fixed size** output: one byte in or one gigabyte in, the answer is the same length.
5. It has the **avalanche effect**: change one bit of input and about half the output bits flip. There is no gentle drift.
6. It is **one-way**: given the output you cannot work backwards to the input, other than by guessing.
7. It is **collision resistant**: it should be impractical to find two different inputs with the same output.
8. Git used SHA-1, which produces 160 bits. Printed four bits per character in hexadecimal, that is 40 characters.
9. The honest version: the fifth property, collision resistance, is broken for SHA-1. That is not a rumour, it is a demonstrated public fact from 2017.

■ 38.2.2 PLAIN – a picture in your head

1. Think of a machine that turns any document into a 40-character summary code.
2. Feed it a shopping list, get a code. Feed it the complete works of a novelist, get a code of exactly the same length.
3. Change one comma anywhere in the novel and the code changes beyond recognition, not slightly.
4. You cannot run the machine backwards. Given a code, there is no way to get the novel out of it.
5. Collision resistance is the promise that nobody can make two different documents that produce the same code.
6. That promise held for SHA-1 for about twenty-two years. Then it did not.

Where this comparison breaks:

7. A summary suggests something readable. A hash output is not readable and carries no meaning at all. It is only a label.
8. Also, a real summary of a longer document loses information gently. A hash loses nearly all information at once. From 160 bits you can prove a match, but you can never reconstruct anything.

■ 38.2.3 PLAIN – a worked example

1. Here are real SHA-1 values computed in the sandbox used to write this chapter, on an Intel Xeon at 2.10 GHz.
2. Determinism. The same five letters, hashed three times:

```
$ printf 'hello' | shasum
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
```

3. Avalanche. Change the last letter from o to p:

```
hello -> aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
hellp -> d4baaad7a68a379b1e133e0fea0603051b0124ca
```

4. Of the 40 hexadecimal characters, only 3 happen to be equal. Of the 160 bits, 85 differ. The ideal for a good hash is 80. It is behaving correctly.
5. Fixed size. One byte in, and one million bytes in:

```
1 byte -> 86f7e437faa5a7fce15d1ddcb9eaeaea377667b8
1,000,000 B -> bef3595266a65a2ff36b700a75e8ed95c68210b6
```

6. Both answers are 40 characters. The length of the input is invisible in the output.
7. Now the git-specific part. Git does not hash the file. It hashes a header plus the file. Here is the proof, done by hand:

```
$ printf 'Introduction.\n' | wc -c
14
$ printf 'blob 14\0Introduction.\n' | shasum
0e2a325c7de3369b164d48fcb823bca9c2cec4e
$ git hash-object docs/intro.md
0e2a325c7de3369b164d48fcb823bca9c2cec4e
```

8. Identical. `shasum` and `git` agree, because `git`'s rule is exactly “hash the header plus the content”.

■ 38.2.4 PLAIN – what is really happening inside

1. SHA-1 works on the input in blocks of 64 bytes.
2. It keeps five 32-bit working registers, 160 bits in total, initialized to fixed constants written into the specification.
3. For each block it mixes the block into those registers through 80 rounds of addition, rotation and bit logic, then adds the result back into the running state.
4. When the input runs out, it appends a single 1 bit, then zero bits, then the original bit length as a 64-bit number, and processes the final block.
5. The five registers, printed end to end, are the answer.
6. The reason a one-bit change scatters the output is that the mixing is designed so every input bit influences every register within a few rounds.
7. A **collision** means two different inputs that end with the same five registers. Because inputs are unlimited and outputs are only 160 bits, collisions must exist. The question is only whether anyone can find one.
8. In 2017 somebody did, on purpose, with two files chosen in advance to be different in a way that mattered.

■ 38.2.5 TECHNICAL – the engineer’s version

1. SHA-1 is specified in FIPS PUB 180-1, published by the United States National Institute of Standards and Technology in 1995. It replaced SHA-0 from FIPS PUB 180 in 1993, which had a design flaw.
2. Output is 160 bits. Internal state is five 32-bit words. Block size is 512 bits. Rounds per block: 80.
3. On 23 February 2017 researchers at CWI Amsterdam and Google announced **SHAttered**, the first public SHA-1 collision: two different PDF files with the identical SHA-1 digest.
4. The published cost was approximately 2 to the power 63.1 SHA-1 evaluations, described as the equivalent of 6,500 CPU-years plus 110 GPU-years, and about 100,000 times cheaper than a brute-force birthday search.
5. On 5 January 2020 Gaetan Leurent and Thomas Peyrin announced **SHAmbles**, a chosen-prefix collision at about 2 to the power 63.4, costed at roughly 45,000 US dollars of rented GPU time. Chosen-prefix is the dangerous kind, because the attacker picks both meaningful beginnings.
6. NIST formally deprecated SHA-1 in 2011, disallowed it for digital signatures from 2013, and in 2022 set 31 December 2030 as the full phase-out date.
7. Here is what those numbers mean in the machine used for this chapter. It hashes at about 1,241 MB/s, which is about 1.95×10^7 blocks per second.

Work item	Evaluations	Time on this one machine
SHAttered, $2^{63.1}$	9.9×10^{18}	about 16,000 years
Birthday bound 2^{80}	1.2×10^{24}	about 2×10^9 years
Full preimage 2^{160}	1.5×10^{48}	about 2×10^{33} years

8. What git did. From **Git 2.13, released in May 2017**, git ships the collision-detecting SHA-1 implementation by Marc Stevens of CWI and Dan Shumow of Microsoft. The Git 2.13 release notes say it “has been integrated and made the default”.
9. That code computes SHA-1 normally but also watches for the internal disturbance vectors that any known collision attack must use. If it sees them, git aborts rather than storing the object. The two SHAttered PDFs cannot be committed to a modern git.
10. Separately, git has been growing a SHA-256 object format. The design document dates from 2017 and 2018, experimental support began landing from **Git 2.19 in September 2018**, and `git init --object-format=sha256` arrived in **Git 2.29 in October 2020**.
11. It works. In the sandbox:


```
$ git init --object-format=sha256 sha256demo
$ git hash-object intro.md
22c6af3b0fccae1e36b21923804fd096c8636e54da038f04c424208
d2ecb1650
```
12. That is 64 hexadecimal characters, and it verifies by hand exactly the same way: `printf 'blob 14\0Introduction.\n' | sha256sum` gives the same value.
13. But it is still not the default in 2026, and git’s own documentation says that until the wire protocol gains SHA-256 support, using SHA-256 storage on public-facing servers is strongly discouraged. Repositories in the two formats cannot yet talk to each other.
14. The honest version, and the part people argue about: Linus Torvalds has said publicly that git’s hash was chosen mainly to guard against accidental corruption and to give a stable name, not as a signature scheme. Critics reply that in practice people do rely on commit hashes as identity, in deploy pipelines and in audit trails. Both points are true.

15. Where experts disagree: some argue the SHA-256 migration is overdue and the ecosystem cost is worth paying now. Others argue that with collision detection turned on, a real attack on a git repository is far harder than an attack on a bare hash, so the migration can proceed slowly. There is no settled answer.

■ 38.2.6 WORDS – remember these

1. Hash function — arithmetic that turns any data into a short fixed code — a map from arbitrary-length input to fixed-length digest.
2. Avalanche effect — one small change scrambles the whole answer — a one-bit input change flips about half the output bits.
3. Collision — two different inputs with the same code — two distinct pre-images mapping to one digest.
4. Preimage resistance — you cannot work backwards from the code — finding any input for a given digest should cost about 2^{160} for SHA-1.
5. SHAttered — the 2017 demonstration that SHA-1 collisions are buildable — CWI Amsterdam and Google, 23 February 2017, cost about $2^{63.1}$.
6. sha1dc — git's hardened SHA-1 that spots attacks — the collision-detecting SHA-1 of Marc Stevens and Dan Shumow, default since Git 2.13, May 2017.

38.3 The four object types

■ 38.3.1 PLAIN – in simple words

1. Git stores exactly four kinds of object. Everything in a repository is one of these four.
2. A **blob** is the content of one file. Just the bytes. It does not know its own name, its folder, or when it was made.
3. A **tree** is one directory listing. It maps names to blobs and to other trees, and records each entry's permission mode.
4. A **commit** is a snapshot marker. It names exactly one tree, names zero or more parent commits, records who wrote it and who recorded it with times, and carries a message.
5. A **tag object** is a named, described pointer, normally to a commit, with its own author and message, and optionally a signature.
6. That is the whole vocabulary. Four nouns.
7. Notice what is missing. There is no “file” object holding a name, no “change” object, no “diff” object, and no “branch” object. Those are all built from the four.

■ 38.3.2 PLAIN – a picture in your head

1. Think about a warehouse that stores photographs of a whole office, taken every day.
2. A blob is one printed page in a filing cabinet. It has no label on it. It is just the paper.
3. A tree is an index card for one room. It says “the page called README is card number f6da25f, the cupboard called docs is index card 80b4ad5”.
4. A commit is the front cover of one day's survey. It says “here is the index card for the whole office, taken at 10:00 on 5 January, by this person, for this reason, and yesterday's cover was that one over there”.
5. A tag object is a brass plaque screwed on to one particular day's cover, saying “this one is version 1.0, blessed by this person on this date”.

Where this comparison breaks:

6. In a warehouse, tomorrow's survey needs a whole new set of pages. In git, tomorrow's cover reuses every index card and every page that did not change. Only the changed cards are re-cut.
7. And in a warehouse the covers are dated by the clerk. In git the cover's own identity is computed from its contents, including the date, so you cannot alter the date and keep the identity.

■ 38.3.3 PLAIN - a worked example

1. Here is a real repository. Three commits, two directories, five files by the end. Every hash below came out of the sandbox.
2. A blob. `git cat-file -t` says the type, `-s` says the size in bytes, `-p` pretty-prints the content:

```
$ git cat-file -t 0e2a325
blob
$ git cat-file -s 0e2a325
14
$ git cat-file -p 0e2a325
Introduction.
```

3. Notice: no file name anywhere. This blob does not know it is `intro.md`.
4. A tree. This is the top directory of the first commit:

```
$ git cat-file -t b39d894
tree
$ git cat-file -p b39d894
100644 blob f6da25f44e6006699370496c1d1033320df8802b    README.md
040000 tree 80b4ad5b31db187c032b03fca7a3b9a0cbdc2747    docs
040000 tree ed46dd19f6204c92947780231982716d4016b9bb    src
```

5. Three entries. One blob and two sub-directories. The names live here, in the tree, not in the blobs.
6. A commit. This is the third commit:

```
$ git cat-file -t b53a9f4
commit
$ git cat-file -p b53a9f4
tree 19f6e3d84c4955832c7f9d39e5205c273dd15d91
parent fb17f98cd37f546dfedb2f91e1b7a9762bf45d17
author KedByte Reader <reader@example.com> 1767594600 +0530
committer KedByte Reader <reader@example.com> 1767594600 +0530

Third commit: add src/util.c
```

7. That is the entire commit. One tree line, one parent line, two people lines with timestamps, a blank line, then the message.
8. A tag object, made with `git tag -a v1.0 -m "First release of the demo":`

```
$ git cat-file -t v1.0
tag
$ git cat-file -p v1.0
object b53a9f46d432aa53e687328841c6dd2b89963809
type commit
tag v1.0
tagger KedByte Reader <reader@example.com> 1767598200 +0530

First release of the demo
```

9. The tag object is a separate object with its own hash, `d54ba1826a59225bc14a2b55e832536cb7a8e4ac`, pointing at the commit.

■ 38.3.4 PLAIN – what is really happening inside

1. Every one of the four objects is stored the same way: a header, then the content, hashed and compressed together.
2. The header is the type word, a space, the content length in bytes, then a single zero byte.
3. So the whole rule of the object store is one line: `hash = SHA1(type + " " + size + "\0" + content)`.
4. Let us prove it for all four types with real numbers.
5. Blob. Content is 14 bytes.

```
$ printf 'blob 14\0Introduction.\n' | shasum
0e2a325c7de3369b164d48fcb823bca9c2cec4e
```

6. Tree. Content is 34 bytes of binary, so we pipe git's own raw bytes back in:

```
$ git cat-file tree ed46dd19 | wc -c
34
$ { printf 'tree 34\0'; git cat-file tree ed46dd19; } | shasum
ed46dd19f6204c92947780231982716d4016b9bb
```

7. Commit. Content is 208 bytes:

```
$ git cat-file commit d9cbb74 | wc -c
208
$ { printf 'commit 208\0'; git cat-file commit d9cbb74; } | shasum
d9cbb74f1fb9bce240d20db0c8e545caf68c802b
```

8. Tag. Content is 156 bytes:

```
$ { printf 'tag 156\0'; git cat-file tag v1.0; } | shasum
d54ba1826a59225bc14a2b55e832536cb7a8e4ac
```

9. Four for four. There is no secret step. You can reproduce every hash in a git repository with `printf` and `shasum`.
10. One warning about the tree. `git cat-file -p` shows you a friendly version. The bytes on disk are not that. Here is the truth:

```
00000000 31 30 30 36 34 34 20 52 45 41 44 4d 45 2e 6d 64
00000010 00 f6 da 25 f4 4e 60 06 69 93 70 49 6c 1d 10 33
00000020 32 0d f8 80 2b 34 30 30 30 30 20 64 6f 63 73 00
00000030 80 b4 ad 5b 31 db 18 7c 03 2b 03 fc a7 a3 b9 a0
00000040 cb dc 27 47 34 30 30 30 30 20 73 72 63 00 ed 46
00000050 dd 19 f6 20 4c 92 94 77 80 23 19 82 71 6d 40 16
00000060 b9 bb
```

11. Each entry is: the mode in ASCII, a space, the file name in ASCII, a zero byte, then the 20 raw bytes of the hash. Not 40 hex characters. 20 binary bytes.
12. Also notice the directory mode is stored as `40000`, five characters, even though `git ls-tree` prints `040000`. The printed leading zero is cosmetic.

■ 38.3.5 TECHNICAL – the engineer's version

1. Object format, canonical for both SHA-1 and SHA-256 repositories: `<type> SP <size in bytes, decimal ASCII, no leading zeros> NUL <content>`.
2. Tree entry format, repeated and concatenated with no separator: `<mode ASCII> SP <name> NUL <20-byte raw object id>`.

3. Tree entries are sorted by name, with directory names compared as if they ended in a slash. This ordering is part of the format, not a convention: two git implementations must agree or the hashes differ.
4. Only five modes are legal in a tree. All five were produced in the sandbox:

```
040000 tree 0479003445f4e5a5ff25360c607ca79ffe4e4ea1 d
120000 blob dab8c79946b1756dcd7db770a986ad40d00c07f4 link.txt
100644 blob b9bca019c83a65e6d717d0b6da86215f45dde1b3 plain.txt
100755 blob 4163036efa65bd4a469e752267498f01ea36a55c run.sh
160000 commit 0f0c4412ec5e57f084c7a39a254a1f9d67f3e965 sub
```

Mode	Meaning
100644	regular file
100755	executable regular file
120000	symbolic link
040000	directory, a sub-tree
160000	gitlink, a submodule commit

5. Git records only the executable bit. It does not record read or write permission, owner, group, or modification time. Those are deliberately outside the model.
6. A symbolic link is stored as a blob whose content is the target path. The blob above holds exactly the nine bytes `plain.txt`.
7. Commit object grammar, in order: one tree line, then zero or more parent lines, then author, then committer, then optional headers such as `gpgsig` and `encoding`, then a blank line, then the message.
8. Timestamps are two fields: seconds since the Unix epoch of 1 January 1970 UTC, then the author's local UTC offset as `+HHMM`. The offset is stored so the original wall-clock time can be reproduced. It does not affect ordering.
9. Two universal object ids you will meet in real work, and they are the same in every SHA-1 repository on earth:

```
$ git hash-object -t tree /dev/null
4b825dc642cb6eb9a060e54bf8d69288fbee4904
$ git hash-object -t blob /dev/null
e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
```

10. The first is the empty tree, useful for diffing a first commit against nothing. The second is the empty file.
11. You can construct all of this by hand. The plumbing path from bytes to a checked-out commit, run in the sandbox:

```
$ B=$(printf 'Hello from plumbing.\n' | git hash-object -w --stdin)
$ printf '100644 blob %s\thello.txt\n' "$B" | git mktree
ae0afea72a045c4653880d3b19ffd7a4979d8d0d
$ git commit-tree ae0afea -m "Made entirely by plumbing"
f0803fab3f189b2ed4f674bf2c63655cc7d7cfde
$ git update-ref refs/heads/main f0803fa
$ git log --oneline
f0803fa Made entirely by plumbing
```

12. No `git add`, no `git commit`. The high-level commands are convenience over exactly these four steps.

■ 38.3.6 WORDS – remember these

1. Blob — stored file content with no name — an object of type blob, raw bytes, addressed by SHA of blob <size>\0<bytes>.
2. Tree — one directory listing — an object mapping sorted names to modes and object ids, using 20 raw bytes per id.
3. Commit — a snapshot marker with a message and a parent — an object naming one tree, zero or more parents, author, committer and message.
4. Annotated tag — a named marker that is itself a stored object — an object of type tag with tagger, message and optional signature.
5. Mode — the small number in front of each tree entry — the six-digit file mode; only 100644, 100755, 120000, 040000 and 160000 are valid.
6. Gitlink — a pointer to a commit in another repository — a tree entry with mode 160000 recording a submodule's commit id.

38.4 How the four types compose

■ 38.4.1 PLAIN – in simple words

1. The four object types stack into one picture.
2. A commit points down at one tree. That tree points at blobs and at more trees. Those point at more blobs.
3. The whole content of the project at one moment hangs off that single commit hash.
4. A commit also points sideways, at its parent commit, which has its own tree, which shares most of the same blobs and sub-trees.
5. When you change one file, git does not copy the project. It creates one new blob for that file, one new tree for the directory containing it, one new tree for each directory above that, and one new commit.
6. Everything else is pointed at again, unchanged.
7. So the cost of a commit is proportional to the depth of the change, not to the size of the project.

■ 38.4.2 PLAIN – a picture in your head

1. Imagine a company organization chart printed on a wall, with a card for each department and a card for each person.
2. Somebody changes their phone number. You do not reprint the wall.
3. You print one new card for that person, one new card for their team naming the new person card, one new card for their division naming the new team card, and one new card for the whole company.
4. Every other card on the wall is still valid and is still pointed at by the new cards.
5. Now you have two complete company charts on the wall, yesterday's and today's, and they share almost all their cards.

Where this comparison breaks:

6. Cards on a wall are pointed at by position. Git objects are pointed at by hash, so a “new card” for identical content is impossible. If two people ended up with the identical card, there would be exactly one card.
7. And a wall chart has to be walked from the top. Git can jump directly to any card, at any depth, if it knows the hash.

■ 38.4.3 PLAIN – a worked example

- Here is the complete object graph of the real three-commit repository, with every object and every arrow. Fifteen objects: 3 commits, 7 trees, 5 blobs.

COMMITTS (each points left to its parent)

```

d9cbb74 <----- fb17f98 <----- b53a9f4
  C1              C2              C3
  |              |              |
  | tree         | tree         | tree
  v              v              v

```

ROOT TREES

```

b39d894      5b9f378      19f6e3d
| | |      | | |      | | |
| | +--src-->| | +--src-->| | +--src-->
| | +--docs-+| | +--docs-+| | +--docs-+|
+--README-+| +--README-+| +--README-+|
      ||      ||      ||
      vv      vv      vv v

```

- Rather than draw crossing lines, here is the same information written out. Each root tree has exactly three entries:

```

b39d894 (root tree of C1)
  README.md -> blob f6da25f
  docs      -> tree 80b4ad5
  src       -> tree ed46dd1

5b9f378 (root tree of C2)
  README.md -> blob f6da25f      SAME as C1
  docs      -> tree dbec85e      NEW
  src       -> tree ed46dd1      SAME as C1

19f6e3d (root tree of C3)
  README.md -> blob f6da25f      SAME as C1 and C2
  docs      -> tree dbec85e      SAME as C2
  src       -> tree 069dcc5      NEW

```

- And the sub-trees, with their blobs:

```

80b4ad5 docs at C1      dbec85e docs at C2 and C3
  intro.md -> 0e2a325    intro.md -> bce6753

ed46dd1 src at C1 and C2 069dcc5 src at C3
  main.c -> 78f2de1      main.c   -> 78f2de1
                        util.c   -> df76ea6

```

- The five blobs, with their real sizes:

Blob	Bytes	It is the content of
f6da25f	25	README.md, all commits
0e2a325	14	docs/intro.md at C1
bce6753	38	docs/intro.md at C2, C3
78f2de1	29	src/main.c, all commits
df76ea6	22	src/util.c at C3

Blob	Bytes	It is the content of

5. Now the important part. Commit 2 changed exactly one file, docs/intro.md. Here is what that cost.

```

EDIT ONE FILE IN ONE DIRECTORY: C1 -> C2

NEW objects (4)                REUSED objects (4)
-----
blob bce6753 intro.md          blob f6da25f README.md
tree dbec85e docs/             tree ed46dd1 src/
tree 5b9f378 root              blob 78f2de1 src/main.c
commit fb17f98                 blob 0e2a325 old intro.md
                                (kept, still in C1)

```

- One blob for the new content. One tree for the directory it is in. One tree for the root above it. One commit. Four objects.
- The src directory was not touched, so tree ed46dd1 is pointed at again by the new root tree. Not copied. Pointed at.
- That is the rule: new objects equal one blob, plus one tree per directory level from the change up to the root, plus one commit.
- Adding src/util.c in commit 3 cost the same shape: new blob df76ea6, new src tree 069dcc5, new root tree 19f6e3d, new commit b53a9f4.

■ 38.4.4 PLAIN - what is really happening inside

- When you run `git commit`, git works from a staging area called the index, which holds the intended next snapshot as a flat list of paths and blob ids.
- Git writes the deepest directories first. For each directory it builds the tree bytes, hashes them, and writes the tree object if it is not already there.
- Because it hashes before writing, an unchanged directory produces exactly the same tree bytes as last time, so exactly the same hash, so no write happens at all.
- Then it builds the parent directory's tree, which now contains the just-computed child hashes, and so on up to the root.
- Finally it builds the commit object: the root tree hash, the current branch tip as parent, your identity and the time, and your message.
- It hashes that, writes it, and moves the branch file to the new commit hash.
- Nothing was diffed. Nothing was compared with the previous commit. The reuse is automatic because equal content produces equal names.
- And here is the proof that git stores whole content, not changes. The second version of docs/intro.md is stored complete:

```

$ git cat-file -s bce6753
38
$ git cat-file -p bce6753
Introduction.
Now with a second line.

```

- Thirty-eight bytes: the full new file, both lines. Not "add one line at the end".
- The diff you see in `git show` is worked out at the moment you ask, by comparing two trees:

```

$ git diff-tree -r d9cbb74 fb17f98
:100644 100644 0e2a325... bce6753... M docs/intro.md

```


- Git compared the two trees, saw one entry whose hash differed, and only then computed the line-by-line difference for display.

■ 38.4.5 TECHNICAL – the engineer’s version

- The object graph is a Merkle DAG. Nodes are objects, edges are embedded object ids, and it is acyclic because an object’s id depends on the ids it contains, so a cycle would require a hash to contain itself.
- Write cost of a commit is $O(d + 1)$ new objects, where d is the number of directory levels from the deepest change to the root, plus one commit and one blob per changed file.
- That is why deep directory hierarchies cost slightly more per commit than flat ones, and why a repository with one enormous directory costs more per commit than you might expect: the whole tree object for that directory is rewritten whenever any entry in it changes.
- Real figures from the sandbox repository at the three-commit point:

Measure	Value
Distinct objects	15
Commits / trees / blobs	3 / 7 / 5
Files in final snapshot	4
Tree objects if no reuse	9 instead of 7

- Only two tree objects were saved in this tiny example. In a repository with thousands of directories and a hundred thousand commits, the saving is the difference between a workable repository and an impossible one.
- Verify the reuse yourself with `git rev-parse` on paths:


```
$ git rev-parse d9cbb74^{tree}:src
ed46dd19f6204c92947780231982716d4016b9bb
$ git rev-parse fb17f98^{tree}:src
ed46dd19f6204c92947780231982716d4016b9bb
```
- Identical, from two different commits. One stored object.
- Implementation detail, not part of the model: git also keeps a cache-tree inside `.git/index` so that unchanged directories do not even need their tree bytes rebuilt during commit. That is a speed optimization; the resulting hashes are identical either way.

■ 38.4.6 WORDS – remember these

- Snapshot — the complete state of the project at one commit — the transitive closure of objects reachable from a commit’s root tree.
- Structural sharing — new versions reusing old parts — persistent data structure behaviour, where unchanged sub-trees are referenced not copied.
- Index — git’s staging area — `.git/index`, a binary file listing paths, modes, blob ids and stat data for the next commit.
- Merkle DAG — a graph where every node names its children by hash — a directed acyclic graph with hash-derived edges, giving whole-graph integrity.
- diff-tree — the command that compares two snapshots — plumbing that walks two trees and reports entries whose object ids differ.

38.5 Why a commit points at its parent

■ 38.5.1 PLAIN – in simple words

1. Every commit except the very first one records the hash of the commit that came before it.
2. That single line, parent <hash>, is what turns a pile of snapshots into a history.
3. Follow the parent links backwards and you reach the beginning. That is what `git log` does.
4. It is not a line, though. It is a **graph**. A commit can have two parents, which is what a merge is. And two different commits can share one parent, which is what a branch is.
5. It is **acyclic**, meaning you can never walk forwards and end up where you started. That is guaranteed by the hashes, not by a rule someone enforces.
6. Because a commit's hash covers its parent's hash, and that parent's hash covers its own parent, one hash at the tip covers the entire history behind it.
7. Change anything anywhere in the past and every hash from that point forward changes. History is therefore tamper-evident.
8. Tamper-evident is not the same as tamper-proof. Anyone can rewrite history and hand you the new version. They just cannot do it invisibly if you already know the old hash.

■ 38.5.2 PLAIN – a picture in your head

1. Think of a family of numbered wax seals.
2. Each seal is made by melting together the day's document and a copy of yesterday's seal.
3. So today's seal depends on yesterday's, which depended on the day before, all the way to the first day.
4. If somebody quietly edits a document from three years ago, they have to make a new seal for that day. That changes the next day's seal, and every seal after it.
5. You hold only the most recent seal in your hand. That one seal proves the whole chain.

Where this comparison breaks:

6. A wax seal can be forged by someone with the same stamp. A hash cannot be forged without solving arithmetic nobody can currently solve.
7. But the comparison also flatters git. Your seal only proves things if you wrote down what the seal looked like. If you never recorded it, someone can hand you a completely different chain of seals and you cannot tell.
8. And unlike a family seal, git history can legitimately fork. Two people can both build on yesterday's seal and both be right.

■ 38.5.3 PLAIN – a worked example

1. Here is a real history from the sandbox with a real merge, drawn by git itself:

```
$ git log --graph --oneline --all
* 7fc6cf8 Merge feature into main
|\
| * 7c7506f Fourth commit: add docs/feature.md on feature
* | 25478c1 Fifth commit: extend README on main
|/
* b53a9f4 Third commit: add src/util.c
* fb17f98 Second commit: edit docs/intro.md
* d9cbb74 First commit: readme, source and docs
```

2. That is a diamond, not a line. Two commits both have b53a9f4 as parent, and one commit has both of them as parents.

3. Look inside the merge commit. It has two parent lines:

```
$ git cat-file -p 7fc6cf8
tree b704d3e37ec1cb98ac5712b56206dbe28b5aac6e
parent 25478c16e00d681e80a0db23a6c7bca204e07404
parent 7c7506f6a3b659163ac6f5d9f155a1a9b89f298e
author KedByte Reader <reader@example.com> 1767681000 +0530
committer KedByte Reader <reader@example.com> 1767681000 +0530

Merge feature into main
```

4. The order of the parent lines matters and is not arbitrary. The first parent is the branch you were on. The second is the branch you merged in.
5. That is what the ^ numbers select:

```
$ git rev-parse HEAD^1
25478c16e00d681e80a0db23a6c7bca204e07404
$ git rev-parse HEAD^2
7c7506f6a3b659163ac6f5d9f155a1a9b89f298e
```

6. And the very first commit has no parent line at all. Compare:

```
$ git cat-file -p d9cbb74
tree b39d894c54aa6e9c3c2545eef059431d7b208ebf
author KedByte Reader <reader@example.com> 1767587400 +0530
committer KedByte Reader <reader@example.com> 1767587400 +0530

First commit: readme, source and docs
```

7. No parent line. That is a **root commit**.

■ 38.5.4 PLAIN – what is really happening inside

1. The word “ancestor” has an exact meaning here. Commit A is an ancestor of commit B if you can get from B to A by following parent links, in any number of steps.
2. “Descendant” is the same relation read the other way.
3. Merging looks for the **merge base**, the closest commit that is an ancestor of both sides. In the example above that is b53a9f4.
4. Immutability is not a rule that git enforces with a lock. It is a consequence of naming.
5. If you want to change a commit’s message, git cannot edit the object, because the object’s name is derived from its bytes including the message. Changing the message produces a different object with a different name.
6. So `git commit --amend` does not amend anything. It builds a brand new commit object and moves the branch file to point at it. The old object is still on disk.
7. The same is true of rebase, of `git filter-repo`, and of every other “history rewriting” tool. They all create new objects.
8. The sealing works like this. Suppose an attacker wants to change one character in the first commit’s README.
9. That changes the README blob’s hash. That changes the root tree’s hash. That changes commit 1’s hash. Commit 2 contains commit 1’s hash, so commit 2’s hash changes. And so on to the tip.
10. If you wrote down the tip hash yesterday and it still matches today, then every byte of every file in every commit behind it is unchanged.

■ 38.5.5 TECHNICAL – the engineer’s version

1. Git history is a directed acyclic graph, a DAG, with commits as vertices and parent references as edges. Merge commits have out-degree 2 or more.
2. Octopus merges with three or more parents are legal. The default merge strategy in modern git, ort, refuses to resolve conflicts in an octopus merge, so they are rare in practice.
3. Acyclicity is structural. A commit’s id is a hash over content that includes its parents’ ids, so creating a cycle would require finding a hash preimage, which is why nobody argues about whether cycles are possible.
4. `git merge-base A B` computes the lowest common ancestor. With criss-cross histories there can be several, and the ort strategy handles that by recursively merging the candidate bases.
5. Topological order is a partial order, not a total one. `git log` defaults to showing commits in reverse commit-date order, which can mislead when clocks disagree. Use `git log --topo-order` when the shape matters.
6. Signed commits. A commit signature is stored as a `gpgsig` header inside the commit object itself, so the signature is covered by the commit hash. Here is a real one, created and verified in the sandbox with an SSH key:

```
$ git cat-file -p HEAD
tree 4588a5c622c7b494a501343b782d2d928c5e447f
author KedByte Reader <reader@example.com> 1767846600 +0530
committer KedByte Reader <reader@example.com> 1767846600 +0530
gpgsig -----BEGIN SSH SIGNATURE-----
U1NIU0lHAAAAAQAADMAAAALc3NoLWVkJU1MTkAAAAgrLzsFF...
...
-----END SSH SIGNATURE-----

A signed commit
```

7. The base64 lines are trimmed here to fit the page. The unsigned commit object was 208 bytes; the signed one is 492 bytes.
8. Verification, again real output:

```
$ git verify-commit HEAD
Good "git" signature for reader@example.com with ED25519 key
SHA256:32dP45eSMmVSt/G/CGvcxL/P+M03Nwj9xeTh/GSA2wc
```

9. What a signature adds, precisely: it binds a key to that exact commit hash, and therefore to the whole history behind it. It answers “who asserted this tip”, which the hash alone cannot.
10. What it does not add: it does not stop rewriting. An attacker with commit access rewrites history and signs the new version with their own key. Verification then fails only if you check the key against a list you trust.
11. This was visible in the sandbox. Before an allowed-signers file was configured, git reported the signature as cryptographically good but with “No principal matched”. Good maths, unknown signer. Those are different questions and git keeps them separate.

Question	Answered by
Are the bytes unchanged?	the object hash
Is the history unchanged?	the tip commit hash
Who asserted this tip?	a valid signature
Should I trust that key?	your allowed-signers list

12. The honest version: git's tamper-evidence protects you against silent modification of a repository you already have a hash for. It does not tell you whether the history you were handed in the first place is the real one.

■ 38.5.6 WORDS - remember these

1. Parent — the commit that came before — an object id recorded in a parent header inside the commit object.
2. Root commit — a commit with no parent — a commit object with zero parent headers; a repository may have several.
3. Merge commit — a commit joining two lines of work — a commit object with two or more parent headers, first parent being the branch merged into.
4. Ancestor — reachable by walking parents backwards — the transitive closure of the parent relation.
5. Merge base — the closest shared ancestor of two commits — the lowest common ancestor in the commit DAG, computed by `git merge-base`.
6. Tamper-evident — changes cannot be hidden — any modification alters the object id and propagates to every descendant hash.

38.6 What a branch really is

■ 38.6.1 PLAIN - in simple words

1. A branch is a file. That is the whole answer.
2. The file lives at `.git/refs/heads/` followed by the branch name.
3. Inside that file are 40 hexadecimal characters and a newline. Forty-one bytes.
4. Those 40 characters are the hash of one commit: the tip of the branch.
5. A branch does not contain commits. It points at one commit. The commits it “contains” are simply the ones you reach by walking parents backwards from there.
6. Creating a branch writes one 41-byte file. Nothing is copied. No files are duplicated. That is why branching is nearly free.
7. Deleting a branch deletes one 41-byte file. The commits are untouched.
8. Committing means writing a new commit object and then overwriting those 41 bytes with the new hash.

■ 38.6.2 PLAIN - a picture in your head

1. Think of a long shelf of books, in order, each book naming the one before it.
2. A branch is not a copy of the shelf. It is a sticky note on the spine of one book saying “main”.
3. Adding a branch means writing one more sticky note. It takes a second and costs one note.
4. Moving a branch forward means peeling the note off one book and putting it on the next.
5. Removing a branch means throwing away a sticky note. Every book is still on the shelf.

Where this comparison breaks:

6. A sticky note can fall off and be lost. Git writes the move into a log, the reflog, so you can put it back. Section 38.9 covers that.
7. And a shelf is one physical thing. In git, two people can have sticky notes with the same name on completely different books, and neither is wrong until they try to agree.

■ 38.6.3 PLAIN – a worked example

1. Real output from a fresh sandbox repository with two commits.

```
$ ls -l .git/refs/heads/
-rw-r--r-- 1 root root 41 Aug 13 03:18 main
$ cat .git/refs/heads/main
bc7fc422aec1054a108922e49253ce8271b1e957
```

2. Forty-one bytes. Forty hexadecimal characters and a newline, confirmed by dumping the raw bytes:

```
| 0000040 7 1 b 1 e 9 5 7 \n
```

3. Now create two more branches, one at the tip and one at the previous commit:

```
$ git branch topic
$ git branch spike HEAD~1
$ ls -l .git/refs/heads/
-rw-r--r-- 1 root root 41 main
-rw-r--r-- 1 root root 41 spike
-rw-r--r-- 1 root root 41 topic
```

4. Three files, 41 bytes each, 123 bytes in total. Their contents:

```
main bc7fc422aec1054a108922e49253ce8271b1e957
topic bc7fc422aec1054a108922e49253ce8271b1e957
spike 1dfe2f9b9521c9a46513a095c135831a8b54ef01
```

5. main and topic hold the same 40 characters. They are two names for one commit, and the repository is not one byte larger for it.
6. Deleting a branch does not delete the commit:

```
$ git branch -D spike
Deleted branch spike (was 1dfe2f9).
$ git cat-file -t 1dfe2f9
commit
```

7. The commit is still there, still readable, still has its tree and its blobs. Only the pointer went.
8. You can move a branch by hand, because it really is just a file:

```
$ git update-ref refs/heads/topic 1dfe2f9b9521c9a4...
$ git branch -v
* main 0f0c441 three
  topic 1dfe2f9 one
```

■ 38.6.4 PLAIN – what is really happening inside

1. Sometimes you look for .git/refs/heads/main and it is not there. The branch still exists. It has simply been packed.
2. Git can move all its ref files into one file called .git/packed-refs, to avoid thousands of tiny files.
3. Watch it happen for real:

```
$ git pack-refs --all
$ ls .git/refs/heads/
(empty)
$ cat .git/packed-refs
# pack-refs with: peeled fully-peeled sorted
bc7fc422aec1054a108922e49253ce8271b1e957 refs/heads/main
```

```
1dfe2f9b9521c9a46513a095c135831a8b54ef01 refs/heads/spike
bc7fc422aec1054a108922e49253ce8271b1e957 refs/heads/topic
```

4. The directory is empty and everything still works, because git looks in both places.
5. The rule is: a loose file wins over the packed file. Commit once more and the loose file comes back for that branch only:

```
$ git commit -m "three"
$ cat .git/refs/heads/main
0f0c4412ec5e57f084c7a39a254a1f9d67f3e965
$ grep main .git/packed-refs
bc7fc422aec1054a108922e49253ce8271b1e957 refs/heads/main
```

6. Two different values for refs/heads/main exist on disk at once, and that is fine. The loose file is the current one. The packed entry is stale and will be tidied later.
7. This is why you should never read ref files yourself in a script. Ask git: `git rev-parse main`, or `git show-ref`, or `git for-each-ref`.
8. Contrast with a system where a branch is a copy. If branching meant copying the working tree, then in the sandbox's 200-commit repository each branch would cost 109,408 bytes of file content. A thousand branches would be about 104 MiB.
9. In git, a thousand branches cost 41,000 bytes of actual content.

■ 38.6.5 TECHNICAL – the engineer's version

1. A ref is a name in a hierarchical namespace, refs/heads/* for local branches, that resolves to an object id.
2. Storage back-ends, as of Git 2.43: loose files under .git/refs, plus .git/packed-refs. A newer refable back-end exists as an option in recent releases and is not the default.
3. Measured in the sandbox on a 200-commit repository:

Operation	Measured cost
Create 1,000 branches	2.28 s, 41,000 bytes
Disk blocks for those refs	4.0 MB (4 KiB per file)
After <code>git pack-refs --all</code>	one file of 56,996 bytes
Disk blocks after packing	28 KB

4. Note the gap between 41,000 bytes of content and 4.0 MB of disk. Each 41-byte file still occupies a whole filesystem block. That is the real reason packed-refs exists, and why repositories with tens of thousands of refs feel slow before it is used.
5. Ref updates are made atomic by writing a temporary file and renaming it over the target, and by taking a .lock file. This is why two concurrent commits in the same repository cannot half-write a branch.
6. Historical contrast, and it is worth knowing why git felt revolutionary in 2005. In CVS, released in 1990, a branch was recorded inside every single file's history, so branching a large tree touched every file. In Subversion, released in 2000, a branch is a cheap server-side copy, but it appears as a real directory path and switching a working copy is not free.
7. Git's design goal, stated by Linus Torvalds when he began the project on 3 April 2005 after the BitKeeper licence withdrawal, was that branching and merging should be cheap enough to do casually. Writing 41 bytes achieves that.

- Convention, not standard: the default branch name. Git shipped `master` as the default from 2005. GitHub changed the default for new repositories to `main` in October 2020, and git itself added `init.defaultBranch` so the choice is configurable. Neither name means anything to the software.

■ 38.6.6 WORDS - remember these

- Ref — a name that points at a commit — an entry in the `refs/` namespace resolving to an object id.
- Branch — a moving pointer to the tip of a line of work — a ref under `refs/heads/` that is advanced automatically when you commit.
- packed-refs — one file holding many refs — `.git/packed-refs`, a sorted text file consulted when no loose ref file exists.
- Loose ref — a ref stored as its own small file — a 41-byte file under `.git/refs/`, which takes precedence over the packed entry.
- update-ref — the command that sets a ref by hand — plumbing that writes a ref value with locking and reflog recording.

38.7 HEAD

■ 38.7.1 PLAIN - in simple words

- HEAD is another file, at `.git/HEAD`, and it answers one question: where am I right now.
- Normally it does not contain a commit hash. It contains the name of a branch.
- Its whole content is the text `ref: refs/heads/main` and a newline. Twenty-one bytes.
- That is called **attached** HEAD. You are “on” a branch. When you commit, git writes the new commit and then moves that branch forward.
- Sometimes HEAD contains a commit hash directly instead of a branch name. That is **detached** HEAD.
- Detached means you are looking at a commit without being on any branch. You can look, build, test, even commit.
- It is not an error. It is a normal state. Git even prints a helpful notice when you enter it.
- The danger is only this: if you commit while detached and then walk away, nothing points at your new commit, so it is easy to lose track of it.

■ 38.7.2 PLAIN - a picture in your head

- Think of the shelf of books again, with sticky notes for branch names.
- HEAD is your finger. Normally your finger rests on a sticky note, not on a book. When the note moves, your finger goes with it.
- Detached HEAD is when you take your finger off the notes and put it directly on a book.
- You can still read that book, and you can still write new books after it. But no sticky note follows you.
- If you then move your finger back to a note, the books you wrote are still on the shelf, but with nothing labelling them, they are easy to overlook.

Where this comparison breaks:

- On a real shelf you would still see the new books. In git nothing is visible in `git branch` or `git log` unless something points at it.
- But git is kinder than the shelf: it writes down where your finger has been. That log is the reflog, and it is how you find those books again.

■ 38.7.3 PLAIN – a worked example

1. Attached state, real output:

```
$ cat .git/HEAD
ref: refs/heads/main
$ git symbolic-ref HEAD
refs/heads/main
$ git status -sb
## main
```

2. Now detach on purpose. Git prints a long, genuinely useful notice:

```
$ git checkout HEAD~2
Note: switching to 'HEAD~2'.

You are in 'detached HEAD' state. You can look around, make
experimental changes and commit them, and you can discard any
commits you make in this state without impacting any branches
by switching back to a branch.

If you want to create a new branch to retain commits you create,
you may do so (now or later) by using -c with the switch command.
Example:

    git switch -c <new-branch-name>

Or undo this operation with:

    git switch -

HEAD is now at b53a9f4 Third commit: add src/util.c
```

3. Look at the file now. No branch name. A raw hash:

```
$ cat .git/HEAD
b53a9f46d432aa53e687328841c6dd2b89963809
$ git symbolic-ref HEAD
fatal: ref HEAD is not a symbolic ref
$ git status -sb
## HEAD (no branch)
```

4. Commit something while detached and HEAD simply advances by itself:

```
$ git commit -m "Work done while detached"
$ git rev-parse HEAD
3a0a3d62188c8a51d4d914b5efd000ea8483695b
$ cat .git/HEAD
3a0a3d62188c8a51d4d914b5efd000ea8483695b
```

5. No branch moved, because no branch was involved.

6. Now leave. Git warns you, by name and by hash:

```
$ git checkout main
Warning: you are leaving 1 commit behind, not connected to
any of your branches:

    3a0a3d6 Work done while detached

If you want to keep it by creating a new branch, this may be a
good time to do so with:
```

```
git branch <new-branch-name> 3a0a3d6
```

```
Switched to branch 'main'
```

7. That warning contains everything you need to recover. Copy the hash and run `git branch keep-this 3a0a3d6`.

■ 38.7.4 PLAIN – what is really happening inside

1. HEAD is a **symbolic reference**: a ref whose value is the name of another ref rather than an object id.
2. When git needs “the current commit”, it reads HEAD. If the content starts with `ref :`, it follows that name and reads the branch file. Otherwise it uses the hash directly.
3. That one indirection is why committing on a branch moves the branch. Git updates whatever HEAD points to.
4. How you end up detached, in practice:
 1. `git checkout <sha>` or `git checkout <tag>` or `git checkout origin/main`.
 2. `git rebase` while it is replaying commits.
 3. A continuous integration job doing `git checkout $COMMIT_SHA`, which is why almost every CI build runs detached.
 4. `git bisect`, which walks you across commits.
5. How to recover, in order of preference:
 1. If you have not committed: `git switch -` or `git checkout main`. Nothing is at risk.
 2. If you have committed and want to keep it: `git branch <name>` right there, or `git switch -c <name>`.
 3. If you already left and lost the hash: use the reflog, section 38.9.
6. The commits you made while detached are ordinary objects. They are not marked, not special, not fragile. They are simply unreferenced, and unreferenced objects are eventually swept up by garbage collection.

■ 38.7.5 TECHNICAL – the engineer’s version

1. `.git/HEAD` is 21 bytes in the attached state and 41 bytes in the detached state. Both are plain text.
2. The plumbing commands are `git symbolic-ref HEAD` to read or write the symbolic form, and `git rev-parse HEAD` to resolve either form to an object id.
3. `git rev-parse --abbrev-ref HEAD` prints the branch name when attached and the literal string HEAD when detached. Scripts use this to decide.
4. Other pseudo-refs live in `.git/` alongside HEAD and appear during operations. All of these were observed in the sandbox:

Pseudo-ref	Present during / after
ORIG_HEAD	reset, merge, rebase
FETCH_HEAD	after <code>git fetch</code>
MERGE_HEAD	an unfinished merge
CHERRY_PICK_HEAD	an unfinished cherry-pick
AUTO_MERGE	ort strategy’s merged tree

5. `advice.detachedHead` controls the long notice. Setting it to false silences it, which is standard practice in CI images where every checkout is detached by design.

6. A worktree created with `git worktree add` gets its own HEAD, stored under `.git/worktrees/<name>/HEAD`. That is why two worktrees of one repository can sit on different commits while sharing one object store.
7. The honest version: people describe detached HEAD as dangerous. It is not. It is exactly as safe as any commit that no branch points at. The risk is purely about visibility, and the reflog removes most of that risk.

■ 38.7.6 WORDS – remember these

1. HEAD — the file saying where you are — a pseudo-ref at `.git/HEAD`, usually symbolic.
2. Symbolic ref — a ref pointing at another ref by name — content of the form `ref: refs/heads/<name>`.
3. Attached HEAD — you are on a branch — HEAD is symbolic and commits advance the named branch.
4. Detached HEAD — you are on a commit with no branch — HEAD holds an object id directly; new commits are unreferenced.
5. Pseudo-ref — a special one-off ref in the git directory — names such as `ORIG_HEAD` and `MERGE_HEAD`, written by particular operations.

38.8 The rest of the refs

■ 38.8.1 PLAIN – in simple words

1. Branches are only one kind of ref. All refs live in the same namespace and all of them work the same way: a name that resolves to an object id.
2. `refs/heads/` holds your local branches.
3. `refs/tags/` holds tags. There are two kinds and they differ on disk.
4. A **lightweight tag** is just a ref file pointing straight at a commit. No extra object exists.
5. An **annotated tag** points at a tag object, and that tag object points at the commit. It carries a tagger, a date, a message and, optionally, a signature.
6. `refs/remotes/` holds remote-tracking refs, such as `origin/main`. These are your local cached memory of what a remote looked like the last time you spoke to it.
7. `refs/notes/` holds notes, a way of attaching text to a commit without changing the commit.
8. `refs/stash` holds stashed work. There are a few other special names too.

■ 38.8.2 PLAIN – a picture in your head

1. Think of the shelf of books once more, and the sticky notes.
2. A branch note is one you move forward every day.
3. A lightweight tag is a note you stick on and never move.
4. An annotated tag is not a note at all. It is a small printed card in a sleeve, with a date, a signature and a sentence explaining why this book matters, and the card names the book.
5. A remote-tracking note is a note in a different colour saying “when I last visited the other library, their note called `main` was on this book”.
6. It does not move when their note moves. It moves when you next visit.

Where this comparison breaks:

7. On a shelf you could check the other library instantly by looking across the room. In git, checking costs a network round trip, which is exactly why the cached note exists and exactly why it goes stale.

■ 38.8.3 PLAIN – a worked example

1. The complete ref tree of the sandbox repository, exactly as git reports it:

```
$ git for-each-ref --format='%(objectname:short) %(objecttype) %(refname)'
7c7506f commit refs/heads/feature
70a5d62 commit refs/heads/fix
39bf0c8 commit refs/heads/main
cc1849e commit refs/notes/commits
b53a9f4 commit refs/remotes/origin/main
b53a9f4 commit refs/tags/v1-light
d54ba18 tag refs/tags/v1.0
```

2. Look at the last two rows. Both are tags, both point at the same release, but the object types differ.
3. refs/tags/v1-light has type commit. The ref points at the commit directly. That is a lightweight tag.
4. refs/tags/v1.0 has type tag. The ref points at a tag object, which in turn points at the commit. That is an annotated tag.
5. On disk, both are 41-byte files with 40 hex characters. The difference is what those characters name:

```
.git/refs/tags/v1-light -> b53a9f46... (a commit)
.git/refs/tags/v1.0 -> d54ba182... (a tag object)
```

6. And the tag object itself, which the lightweight tag simply does not have:

```
$ git cat-file -p v1.0
object b53a9f46d432aa53e687328841c6dd2b89963809
type commit
tag v1.0
tagger KedByte Reader <reader@example.com> 1767598200 +0530

First release of the demo
```

■ 38.8.4 PLAIN – what is really happening inside

1. Remote-tracking refs are updated by exactly two things: git fetch and git push. Nothing else touches them.
2. So origin/main is not a live reading of the server. It is a note of what the server said last time. This matters enormously and section 38.12 returns to it.
3. git ls-remote is the command that asks the server right now, over the network, without changing anything locally:

```
$ git ls-remote origin
b53a9f46d432aa53e687328841c6dd2b89963809 HEAD
b53a9f46d432aa53e687328841c6dd2b89963809 refs/heads/main
b53a9f46d432aa53e687328841c6dd2b89963809 refs/tags/v1-light
d54ba1826a59225bc14a2b55e832536cb7a8e4ac refs/tags/v1.0
b53a9f46d432aa53e687328841c6dd2b89963809 refs/tags/v1.0^{}
```

4. Notice the last line. The suffix ^{} means “the object this tag finally resolves to after following it through”. Servers advertise both, so clients need not fetch the tag object just to learn what it points at.
5. The packed-refs file records the same fact with a caret line:

```
d54ba1826a59225bc14a2b55e832536cb7a8e4ac refs/tags/v1.0
^b53a9f46d432aa53e687328841c6dd2b89963809
```

6. A note is stored as a whole commit and tree of its own, under `refs/notes/commits`. The tree maps a commit's hash, used as a file name, to a blob holding the note text.
7. That is why notes can be added and changed without altering the commit they describe. The commit does not know about them.
8. A stash entry is a merge commit with two or three parents: the commit you were on, a commit holding the index state, and if needed one holding untracked files.

```
$ git cat-file -p refs/stash
tree a5419a1446acbb442a6da16df29e43c605ebd018
parent 39bf0c8354d383aaf21ff05eaff4304acc24f346
parent 96a1b317649d3354ec613c5f61bcc7ba7eba40c1
```

9. So a stash is not a special storage area. It is ordinary commits with a ref pointing at them.

■ 38.8.5 TECHNICAL – the engineer's version

1. The ref namespace, with the parts you will actually meet:

Namespace	Holds
<code>refs/heads/*</code>	local branches
<code>refs/tags/*</code>	tags, both kinds
<code>refs/remotes/<r>/*</code>	remote-tracking refs
<code>refs/notes/*</code>	note trees

2. Beyond those: `refs/stash`, `refs/bisect/*` during a bisect, `refs/replace/*` for object replacement, and on GitHub the server-side `refs/pull/<n>/head` and `refs/pull/<n>/merge`, which are a hosting convention rather than part of git.
3. Lightweight versus annotated is a real operational difference, not a style choice:

Aspect	Lightweight	Annotated
Extra object	none	one tag object
Has message	no	yes
Has tagger and date	no	yes
Can be signed	no	yes
describe prefers it	no	yes

4. `git describe` ignores lightweight tags unless you pass `--tags`. Release engineering therefore uses annotated tags almost universally.
5. The default `fetch refspec`, written into `.git/config` by `git clone`, is `+refs/heads/*:refs/remotes/origin/*`. The leading plus means non-fast-forward updates are allowed for these refs, which is why a force-push upstream quietly rewrites your `origin/*` refs.
6. `git for-each-ref` is the scriptable interface. `git show-ref` is the older one. Both read loose and packed refs correctly, which hand-written scripts that cat ref files do not.

■ 38.8.6 WORDS – remember these

1. Lightweight tag — a fixed pointer to a commit — a ref under `refs/tags/` whose value is a commit id.
2. Annotated tag — a tag with a message and author — a ref pointing at a tag object carrying tagger, date, message and optional signature.

3. Remote-tracking ref — your cached memory of a remote branch — a ref under `refs/remotes/`, updated only by fetch and push.
4. Refspec — the rule mapping their refs to yours — for example `+refs/heads/*:refs/remotes/origin/*`.
5. Peeled ref — the object a tag finally points to — shown as `<tag>^{}` and recorded with a caret line in `packed-refs`.
6. Note — text attached to a commit after the fact — a blob in a tree under `refs/notes/`, keyed by the commit's object id.

38.9 The reflog

■ 38.9.1 PLAIN – in simple words

1. Every time a ref changes value, git writes a line in a log saying where it was and where it went.
2. That log is the **reflog**. There is one for HEAD and one for each branch.
3. It records the old hash, the new hash, who did it, when, and which command caused it.
4. This log is the reason “I destroyed my work with git” is almost always false.
5. A hard reset, a bad rebase, a deleted branch, an amended commit: in every case the old commit object is still on disk and the reflog still knows its hash.
6. The reflog is local and private. It is never pushed, never fetched, and never shared. Cloning a repository does not bring its reflog.
7. Entries do not last forever. By default reachable entries survive 90 days and unreachable ones 30 days.

■ 38.9.2 PLAIN – a picture in your head

1. Think of a security camera pointed at the sticky notes on the shelf.
2. It does not record the books. It records every time a note moved, and to where.
3. If you come back and find a note in the wrong place, you rewind the tape, read off which book it used to be on, and put it back.
4. The books never went anywhere. Only the note moved. That is the whole trick.

Where this comparison breaks:

5. A camera records what everyone did. The reflog records only what happened in your copy, on your machine. Your colleague's reflog is a different tape.
6. And the tape is wiped on a schedule. After about 30 days, a commit that nothing points at can genuinely be collected and deleted.

■ 38.9.3 PLAIN – a worked example

1. Here is a real disaster and a real recovery, done in the sandbox. The starting point is a repository whose main is at a merge commit.

```
$ git log --oneline -3
7fc6cf8 Merge feature into main
25478c1 Fifth commit: extend README on main
7c7506f Fourth commit: add docs/feature.md on feature
```

2. Now the mistake. A hard reset two commits back, which throws away the merge and everything after it:

```
$ git reset --hard HEAD~2
HEAD is now at b53a9f4 Third commit: add src/util.c
```



```
$ git log --oneline -3
b53a9f4 Third commit: add src/util.c
fb17f98 Second commit: edit docs/intro.md
d9cbb74 First commit: readme, source and docs
$ ls
README.md docs src
```

3. The merge is gone from the log. Files that only existed after it are gone from the working directory. This is the moment people panic.
4. It is not gone. The commit object is still there and still readable:

```
$ git cat-file -t 7fc6cf8
commit
$ git cat-file -p 7fc6cf8 | head -3
tree b704d3e37ec1cb98ac5712b56206dbe28b5aac6e
parent 25478c16e00d681e80a0db23a6c7bca204e07404
parent 7c7506f6a3b659163ac6f5d9f155a1a9b89f298e
```

5. And even without knowing that hash, the reflog tells you it:

```
$ git reflog
b53a9f4 HEAD@{0}: reset: moving to HEAD~2
7fc6cf8 HEAD@{1}: checkout: moving from 3a0a3d6... to main
3a0a3d6 HEAD@{2}: commit: Work done while detached
```

6. HEAD@{1} is where HEAD was one move ago: 7fc6cf8. That is the merge.
7. The branch has its own reflog, which is usually the cleaner one to read:

```
$ git reflog show main
7fc6cf8 main@{0}: merge feature: Merge made by the 'ort' strategy
25478c1 main@{1}: commit: Fifth commit: extend README on main
b53a9f4 main@{2}: commit: Third commit: add src/util.c
```

8. The recovery is one command:

```
$ git rev-parse main@{1}
7fc6cf85594a850b727418867d4054af04412e88
$ git reset --hard main@{1}
HEAD is now at 7fc6cf8 Merge feature into main
$ git log --oneline -3
7fc6cf8 Merge feature into main
25478c1 Fifth commit: extend README on main
7c7506f Fourth commit: add docs/feature.md on feature
```

9. Everything is back. Total elapsed effort: one lookup and one command.

■ 38.9.4 PLAIN – what is really happening inside

1. The reflog is a plain text file per ref. HEAD's log is `.git/logs/HEAD`. A branch's log is `.git/logs/refs/heads/<name>`.
2. Each line has five fields, tab-separated at the end: old object id, new object id, the person, the timestamp with offset, then the reason.
3. Here is one real line, wrapped across two lines to fit this page:

```
fb17f98cd37f546dfedb2f91e1b7a9762bf45d17 b53a9f46d432aa53e6873
28841c6dd2b89963809 KedByte Reader <reader@example.com>
1767594600 +0530 commit: Third commit: add src/util.c
```

- The very first line of a branch's log has all zeros as the old id, because the branch did not exist before:

```
0000000000000000000000000000000000000000000000000000000000000000 d9cbb74f1fb9bce240d20d
b0c8e545caf68c802b ... commit (initial): First commit ...
```

- The `@{n}` syntax counts entries in that file. `main@{0}` is the current value, `main@{1}` is the value before the most recent change, and so on.
- There is also a time form. `main@{2026-01-06}` asks what main pointed at on that date. If the log does not go back that far, git says so:


```
warning: log for 'main' only goes back to Thu, 13 Aug 2026 03:12:55
```
- Notice `@{n}` and `~n` are completely different questions. `HEAD~1` means “the parent commit in the history”. `HEAD@{1}` means “wherever HEAD was pointing one move ago”. They frequently give different answers.
- Reflogs are only kept if `core.logAllRefUpdates` is on, which it is by default for non-bare repositories. Bare repositories, the kind that sit on a server, do not keep reflogs by default. That is worth remembering before you rely on one.

■ 38.9.5 TECHNICAL - the engineer's version

- Expiry defaults, from git's own documentation:

Setting	Default
<code>gc.reflogExpire</code>	90 days
<code>gc.reflogExpireUnreachable</code>	30 days
<code>gc.pruneExpire</code>	2 weeks ago
<code>gc.auto</code>	6700 loose objects
<code>gc.autoPackLimit</code>	50 packs

- “Unreachable” here means the reflog entry names a commit that is no longer reachable from any ref. Those are the entries you most need after a mistake, and they are the ones that expire soonest.
- Reflog entries keep their objects alive. `git gc` treats reflog-referenced objects as reachable, which is why a commit lost by reset is not collected the same afternoon.
- Proof from the sandbox. Immediately after abandoning a detached commit, `git fsck --unreachable` reported nothing, because the reflog still held it. Adding `--no-reflogs` revealed the truth:

```
$ git fsck --unreachable --no-reflogs
unreachable tree bcb2c24a564e8ff3d64369f03559123b68b04b03
unreachable blob 9a5abed11a92dda94398c590ec782181a789f4c5
unreachable commit 3a0a3d62188c8a51d4d914b5efd000ea8483695b
```

- The recovery toolkit, in the order you should reach for it:
 - `git reflog` or `git reflog show <branch>` — find the old hash.
 - `git reset --hard <sha>` — move the branch back, discarding current work.
 - `git branch rescue <sha>` — safer: give the lost commit a name without touching your current branch.
 - `git fsck --lost-found` — last resort, when the reflog was pruned or the repository is bare. Writes dangling objects into `.git/lost-found/`.

6. `git log -g` prints the reflog as commits, with full messages, which is often easier to read than `git reflog` when you need to identify work by content.
7. Danger commands and what saves you from each:

Command	Reflog entry that saves you
<code>git reset --hard X</code>	<code>HEAD@{1}</code> and <code><branch>@{1}</code>
<code>git rebase</code>	<code><branch>@{1}</code> , before the rebase
<code>git commit --amend</code>	<code>HEAD@{1}</code> holds the pre-amend commit
<code>git branch -D X</code>	the deletion message prints the hash
<code>git checkout detach</code>	the “leaving N commits behind” warning

8. What the reflog does not save: uncommitted changes in the working tree. `git reset --hard`, `git checkout -- <file>` and `git clean -fd` destroy work that was never turned into an object, and no log exists for it. Anything you staged with `git add` survives, as a dangling blob findable with `git fsck --lost-found`.

■ 38.9.6 WORDS – remember these

1. Reflog — a local log of where each ref has been — per-ref append-only log in `.git/logs/`, recording old id, new id, actor, time and reason.
2. `@{n}` — the value a ref had `n` moves ago — reflog index selector, distinct from the `~n` ancestry selector.
3. Unreachable object — an object no ref points at — a candidate for pruning once its reflog entry expires.
4. Dangling object — an unreachable object with nothing pointing at it at all — reported by `git fsck` and recoverable with `--lost-found`.
5. Prune — permanently delete unreachable objects — `git prune`, or the pruning step inside `git gc`, honouring `gc.pruneExpire`.

38.10 The commands that reveal the machinery

■ 38.10.1 PLAIN – in simple words

1. Git has two layers of commands, and it is worth knowing which is which.
2. **Porcelain** commands are the ones you use daily: `add`, `commit`, `log`, `merge`, `status`, `push`. They are built for people. Their output is meant to read nicely and is allowed to change between versions.
3. **Plumbing** commands are the low-level ones underneath: `cat-file`, `hash-object`, `ls-tree`, `rev-parse`, `rev-list`, `update-ref`. They are built for scripts. Their output format is stable on purpose.
4. The names are git’s own, from its documentation, and they come from the idea that you see the porcelain of a bathroom while the plumbing does the work.
5. Rule for scripts: never parse porcelain output. Use plumbing, or use the porcelain commands’ `--porcelain` flags, which are stable despite the confusing name.
6. This section is a tour of the plumbing, with real output for each command.

■ 38.10.2 PLAIN – a picture in your head

1. A car has a dashboard and it has an engine.
2. The dashboard tells you the speed in a form you can act on. The designer may redesign it next year and nobody minds.
3. The engine has bolts of exact sizes that have not changed in a decade, because tools depend on them.

4. Porcelain is the dashboard. Plumbing is the bolts.
5. You drive with the dashboard. You diagnose with the bolts.

Where this comparison breaks:

6. In a car you rarely need the engine. In git the plumbing is genuinely useful during ordinary work, especially when something has gone wrong and you need an answer that is exactly true rather than nicely presented.

■ 38.10.3 PLAIN – a worked example

1. `git cat-file` with its three most useful flags. `-t` gives the type, `-s` the size in bytes, `-p` the content:

```
$ git cat-file -t b53a9f4
commit
$ git cat-file -s b53a9f4
247
$ git cat-file -t b53a9f4^{tree}
tree
```

2. `git rev-parse` turns any way of naming a commit into the one true hash:

```
$ git rev-parse HEAD
39bf0c8354d383aaf21ff05eaff4304acc24f346
$ git rev-parse --short HEAD
39bf0c8
$ git rev-parse --short=12 HEAD
39bf0c8354d3
$ git rev-parse --abbrev-ref HEAD
main
$ git rev-parse --git-dir
.git
$ git rev-parse --show-toplevel
/tmp/kb38/demo
```

3. It also resolves paths inside a commit, which is how you get a blob's hash without looking at a tree:

```
$ git rev-parse HEAD:docs/intro.md
bce6753328d0de8aaa42692aba92b99182ded5ce
```

4. `git hash-object` computes a name without storing anything, and with `-w` stores it:

```
$ printf 'Introduction.\n' | git hash-object --stdin
0e2a325c7de3369b164d48fcb823bca9c2cec4e
```

5. `git ls-tree` reads a tree. Plain, recursive, and with sizes:

```
$ git ls-tree HEAD
100644 blob 6af3a5ad4b99c155602b8f78298d89fc15b61209    README.md
040000 tree 8c33b40fe5eb264b8d43c66f01dc437cc4c90aa5    docs
040000 tree 069dcc578e4a416937da34a3dcc4d209c1048c63    src

$ git ls-tree -r HEAD
100644 blob 6af3a5ad... README.md
100644 blob 76c95066... docs/feature.md
100644 blob bce67533... docs/intro.md
100644 blob 61a98438... docs/mainnote.md
100644 blob 78f2de10... src/main.c
100644 blob df76ea63... src/util.c
```

6. `git count-objects -v` tells you the shape of the object store:

```
$ git count-objects -vH
count: 15
size: 60.00 KiB
in-pack: 31
packs: 1
size-pack: 4.60 KiB
```

7. `git rev-list` lists commit hashes, and counts them:

```
$ git rev-list --count HEAD
7
$ git rev-list --count --all
9
```

■ 38.10.4 PLAIN – what is really happening inside

1. The revision syntax deserves its own explanation because two of its forms look alike and mean different things.
2. `X~n` means “go back *n* steps, always taking the first parent”. `HEAD~2` is the grandparent.
3. `X^n` means “the *n*-th parent of this one commit”. `HEAD^2` is the second parent, which only exists on a merge commit.
4. On the real merge commit in the sandbox:

```
$ git rev-parse HEAD      # the merge
7fc6cf85594a850b727418867d4054af04412e88
$ git rev-parse HEAD^1    # first parent, the branch merged into
25478c16e00d681e80a0db23a6c7bca204e07404
$ git rev-parse HEAD^2    # second parent, the branch merged in
7c7506f6a3b659163ac6f5d9f155a1a9b89f298e
$ git rev-parse HEAD~1    # same as ^1
25478c16e00d681e80a0db23a6c7bca204e07404
$ git rev-parse HEAD~2    # two first-parent steps back
b53a9f46d432aa53e687328841c6dd2b89963809
```

5. `HEAD~1` and `HEAD^1` gave the same answer. `HEAD~2` and `HEAD^2` gave completely different answers. That is the trap.
6. And `branch@{1}` is a third thing again: not ancestry at all, but the reflog. In the sandbox `main@{1}` was `7fc6cf8` while `main~1` was a different commit entirely.
7. `git show` is porcelain that stitches several plumbing operations together. Given a commit it prints the metadata plus a computed diff. Given a blob it prints content. Given a tree it prints a listing.
8. `git log --graph --oneline --all` is the single most useful shape-revealing command. `--all` is the important part: without it you only see history reachable from `HEAD`.

■ 38.10.5 TECHNICAL – the engineer’s version

1. Command classification, with the stability promise:

Command	Layer	Output stability
<code>git status</code>	porcelain	may change
<code>git status --porcelain=v2</code>	porcelain	stable, versioned
<code>git cat-file</code>	plumbing	stable
<code>git rev-parse</code>	plumbing	stable
<code>git for-each-ref</code>	plumbing	stable, formattable

2. `git cat-file --batch-check` is the high-throughput form. It reads object names on standard input and writes name, type and size, avoiding one process per object:

```
$ git cat-file --batch-all-objects --batch-check
0e2a325c7de3369b164d48fcb823bca9c2cec4e blob 14
6af3a5ad4b99c155602b8f78298d89fc15b61209 blob 52
78f2de106c92b0d60772bd5aa6c1e6da7bf71005 blob 29
```

3. `git verify-pack -v` lists a packfile's contents. Columns are: object id, type, size, size in the packfile, offset in the packfile, and for deltified objects the chain depth and the base object id:

```
5087b1a6efba7b591bda30d3badbd5056b4fab4b blob 111810 15280 30437
23841aba9acef22ead7d8505ba90653bc3544934 blob 78 93 45717 1 5087
6578910fb9573331428c33593424cb14b02cb49a blob 38 52 45810 2 2384
```

4. Read that carefully. The first blob's real size is 111,810 bytes and it occupies 15,280 bytes in the pack. The second's "size" column reads 78, but that is the size of the delta, not of the object. Depth 1 means one hop to its base.
5. `git rev-list` is the engine under `git log`. It walks the DAG and emits object ids. `--objects` also emits the trees and blobs reachable from each commit, which is exactly what `git pack-objects` consumes during a push.
6. `git fsck` verifies the whole store: every object's hash against its content, every referenced object present, every ref resolvable. It is the command that turns content addressing into a usable integrity check.
7. Useful one-liners worth memorizing:

```
git cat-file -p HEAD^{tree}          # top directory of HEAD
git rev-list --count main ^origin/main # commits not pushed yet
git for-each-ref --sort=-committerdate refs/heads/
git log --graph --oneline --all --decorate
git rev-parse --verify --quiet <sha> # test existence, no output
```

■ 38.10.6 WORDS – remember these

1. Porcelain — the commands people use — the human-facing layer whose output format carries no stability promise.
2. Plumbing — the commands scripts use — the low-level layer with stable, machine-readable output.
3. `rev-parse` — turn a name into a hash — the revision-parsing plumbing that resolves every form of object naming.
4. `~` and `^` — steps back, versus which parent — `~n` is `n` first-parent steps, `^n` selects the `n`-th parent of one commit.
5. `--batch-check` — ask about many objects at once — `git cat-file` batch mode giving id, type and size without one process per object.

38.11 How git stores things efficiently

■ 38.11.1 PLAIN – in simple words

1. Everything so far has described whole snapshots. That sounds wasteful, and on its own it is.
2. Git solves this in two stages.
3. Stage one, **loose objects**. Each object gets its own file, squeezed with zlib compression. Simple, fast to write, wasteful of disk blocks.

4. Stage two, **packfiles**. Git gathers thousands of objects into one big file and, inside that file, stores similar objects as differences from one another.
5. This is the important sentence of the whole section: those differences are a storage trick only. Git's idea of history is still whole snapshots. The deltas exist below that, and you never see them.
6. Git does the packing when you push, when you clone, and when housekeeping runs.
7. `git gc` is the command that does the housekeeping: pack loose objects, pack refs, trim reflogs, and delete objects nothing points at any more.

■ 38.11.2 PLAIN – a picture in your head

1. Imagine keeping every draft of a long report in a filing cabinet, one folder per draft.
2. At first you photocopy the whole report each time. Two hundred drafts, two hundred thick folders. That is loose objects.
3. Then a clerk goes through the cabinet. She keeps one full copy of the report, and replaces the other 199 folders with slips of paper saying “same as the previous draft, but line 812 now reads this”.
4. The cabinet drawer that was full is now nearly empty.
5. Crucially, when you ask for draft 137, she reconstructs it and hands you a complete report. You never see the slips.

Where this comparison breaks:

6. A clerk works from the previous draft. Git compares each object against a window of candidates and picks whichever gives the smallest difference, which is often not the chronological predecessor.
7. And the clerk's slips would be useless if the full copy were lost. Git guarantees every chain terminates at a complete object inside the same packfile.

■ 38.11.3 PLAIN – a worked example

1. Here is a real repository built for this section: one text file of about 110 KB, edited and committed 200 times, one line changed each time.
2. Before any packing:

```
$ git count-objects -vH
count: 600
size: 6.25 MiB
in-pack: 0
packs: 0
$ du -sh .git/objects
7.2M    .git/objects
```

3. Six hundred loose objects: 200 blobs, 200 trees, 200 commits.
4. The total content, uncompressed, is much larger than what is on disk already:

```
$ git cat-file --batch-all-objects --batch-check='%(objectsize)'
(summed)
22463780 bytes = 21.42 MiB
```

5. So zlib alone took 21.42 MiB of content down to about 4.6 MiB of file bytes, occupying 7.2 MB of disk blocks.
6. Now run housekeeping:

```
$ git gc
$ git count-objects -vH
```



```
count: 0
size: 0 bytes
in-pack: 600
packs: 1
size-pack: 100.87 KiB
$ du -sh .git/objects
140K    .git/objects
```

- Read those numbers again. 21.42 MiB of content is now in one packfile of 85,423 bytes, with a 17,872-byte index beside it.

Stage	Bytes	Disk
Raw object content	22,463,780	n/a
Loose, zlib only	4,601,531	7.2 MB
Packed with deltas	85,423	140 KB

- That is a reduction of about 263 to 1 against the raw content, and about 54 to 1 against the already-compressed loose form.
- Where did it come from? Look inside the pack:

```
5087b1a6... blob 111810 15280 30437
23841aba... blob 78 93 45717 1 5087b1a6...
6578910f... blob 38 52 45810 2 23841aba...
2048ca7c... blob 44 58 45862 3 6578910f...
```

- The first blob is stored whole: 111,810 bytes, compressed to 15,280. Every following version is stored as a delta of 38 to 78 bytes.
- Of the 200 blobs, 198 are stored as deltas and only 2 are stored whole.

■ 38.11.4 PLAIN - what is really happening inside

- A packfile begins with the four ASCII bytes PACK, then a 4-byte version number, then a 4-byte object count. Here is the real header of the pack above:

```
| 00000000 50 41 43 4b 00 00 00 02 00 00 02 58 93 0e 78 9c
```

- 50 41 43 4b is PACK. 00 00 00 02 is version 2. 00 00 02 58 is 600 in hexadecimal, exactly the object count git reported.
- The last 20 bytes of the file are a SHA-1 checksum over everything before them, and that checksum is what the file is named after. The pack above is pack-700da35f...b936f4f9ea0477, and the file's final bytes are b9 36 f4 f9 ea 04 77.
- Beside it sits a .idx file starting with the magic bytes ff 74 4f 63 and version 2. It is a sorted table letting git find any object's offset in the pack without scanning it.
- Delta selection works like this. Git sorts objects by type, then by a path hint, then by size descending. It then slides a window over that order and, for each object, tries to express it as a difference against the others in the window.
- Two limits control the result. The window size, default 10, is how many candidates are considered. The depth, default 50, is how many hops a chain may have.
- The sandbox pack hit that limit exactly:

```
chain length = 48: 14 objects
chain length = 49: 24 objects
chain length = 50: 27 objects
```

8. Nothing has depth 51, because 50 is the ceiling. Deeper chains would compress better but cost more to read, because reading one object means applying every delta along the chain.
9. Note the direction. Git usually stores newer objects as deltas against older ones or the reverse, depending on size; the ordering heuristic prefers larger objects as bases. There is no rule that the base is the previous version in time.
10. `git gc` runs several jobs: repack loose objects into a pack, pack the refs into packed-refs, expire old reflog entries, prune unreachable objects older than the prune window, and update helper indexes such as the commit-graph.

■ 38.11.5 TECHNICAL – the engineer’s version

1. Packfile format version 2 has been the on-disk format since 2005 and is stable. It is also the wire format: what travels during clone, fetch and push is a packfile.
2. Two delta encodings exist inside a pack. `OBJ_OFS_DELTA` refers to its base by a relative offset within the same pack, and `OBJ_REF_DELTA` refers to it by full object id. Offsets are smaller, so modern git prefers them locally and uses reference deltas for thin packs on the wire.
3. A **thin pack** omits bases the receiver is known to have. The receiving side completes it with `git index-pack --fix-thin`. This is why a push of one small change is measured in kilobytes even in a large repository.
4. Measured configuration and results from the sandbox:

Item	Value
<code>pack.window default</code>	10 candidates
<code>pack.depth default</code>	50
Deltified blobs observed	198 of 200
Longest chain observed	50
Pack file / index size	85,423 B / 17,872 B

5. Delta compression is completely orthogonal to git’s data model. Every object in a pack still has the same object id it had when loose, still hashes to the same value once reconstructed, and `git cat-file -p` gives identical output. You cannot tell from the model whether an object is deltified.
6. Consequences that do show through, and they matter in real work:
 1. Binary files that do not delta well, such as compressed images, video and compiled artefacts, cost their full size in every version forever.
 2. `core.bigFileThreshold`, default 512 MiB, makes git skip delta attempts and store such files whole.
 3. Repacking a very large repository is memory hungry. `pack.windowMemory` and `pack.threads` exist to control that.
7. `git gc --aggressive` re-computes deltas with a much larger window, by default 250. It can take hours on a large repository and typically yields a modest further reduction. It is rarely the right answer.
8. Modern maintenance uses `git maintenance run` with incremental repacking rather than periodic full `git gc`. That is a change of practice within the last several years, not a change to the format.
9. Historical note: packfiles were not in the first version of git. Linus Torvalds began the project on 3 April 2005 and committed the first version on 7 April 2005, with the message Initial revision of "git", the information manager from hell. Loose objects came first, and packing was added later in 2005 when the loose-object-only design proved too slow for the Linux kernel history. Junio Hamano took over as maintainer in July 2005 and released version 1.0 on 21 December 2005.

■ 38.11.6 WORDS – remember these

1. Loose object — one object in one file — zlib-compressed file under `.git/objects/xx/`.
2. Packfile — many objects in one file — the `.pack` format, version 2, with a `.idx` lookup table beside it.
3. Delta — an object stored as a difference from another — `OBJ_OFS_DELTA` or `OBJ_REF_DELTA` inside a pack, purely a storage encoding.
4. Delta chain depth — how many hops to reach a whole object — controlled by `pack.depth`, default 50.
5. Thin pack — a pack that assumes you already have the bases — used on the wire, completed by `git index-pack --fix-thin`.
6. Garbage collection — tidying the object store — `git gc`, which repacks, packs refs, expires reflogs and prunes unreachable objects.

38.12 Reading the reader's own situation

■ 38.12.1 PLAIN – in simple words

1. During the network outage described in this book, the reader's machine reported `origin/main` as `0a95cc8`.
2. Three separate facts are hidden in that one short line, and telling them apart is the whole skill.
3. First, `0a95cc8` is an abbreviated object id. It is the first 7 of the 40 hexadecimal characters of a commit's hash.
4. Second, `origin/main` is a remote-tracking ref. It is a small local file holding that hash.
5. Third, and this is the part people get wrong, that file was written the last time the reader successfully talked to GitHub. It was not read from GitHub at the moment it was displayed.
6. During an outage, nothing on your machine can tell you what the server currently holds. Your machine can only tell you what it remembers.
7. The reader also rebased a local branch onto a moved `main`, and the branch's hash changed afterwards. Given everything in this chapter, that was unavoidable.

■ 38.12.2 PLAIN – a picture in your head

1. Imagine you write a friend's address in your notebook after visiting them.
2. Months later somebody asks where your friend lives. You read out the notebook.
3. You are not reporting where they live. You are reporting where they lived when you last checked.
4. If the phone lines are down, you cannot check. The notebook is all you have, and it is still the same notebook whether they moved yesterday or not.
5. `origin/main` is that notebook page. `git ls-remote` is the phone call.

Where this comparison breaks:

6. A person might tell you they moved. A git server never volunteers anything. It answers only when asked, and only over the network.
7. And your notebook page can be updated by two events, not one: a fetch, and also a successful push, because after a push you know what you just put there.

■ 38.12.3 PLAIN – a worked example

1. Start with the string itself. Seven characters, `0a95cc8`. What can you learn?

```
$ git cat-file -t 0a95cc8  
commit
```

```
$ git cat-file -s 0a95cc8
$ git cat-file -p 0a95cc8
```

- Those three commands tell you the type, the size and the full contents: the tree, the parent, the author, the committer and the message.
- If the object is not present locally, git says so plainly. Run against the sandbox repository, which has never seen that commit:

```
$ git cat-file -t 0a95cc8
fatal: Not a valid object name 0a95cc8
$ git rev-parse 0a95cc8
fatal: ambiguous argument '0a95cc8': unknown revision or path
not in the working tree.
```

- That message is worth reading carefully. Git cannot tell the difference between “no such commit” and “a commit I have not fetched”. Both look identical from here.
- To expand the abbreviation into the full 40 characters:

```
$ git rev-parse 0a95cc8
$ git rev-parse --short=12 origin/main
```

- To see what the remote-tracking ref currently says, and where that value physically lives:

```
$ git rev-parse origin/main
b53a9f46d432aa53e687328841c6dd2b89963809
$ cat .git/refs/remotes/origin/main
b53a9f46d432aa53e687328841c6dd2b89963809
```

- And to ask the server, right now, over the network:

```
$ git ls-remote origin refs/heads/main
b53a9f46d432aa53e687328841c6dd2b89963809    refs/heads/main
```

- Compare those two values. If they differ, your cached ref is stale. If `git ls-remote` fails, you have learned about the network and nothing at all about the server’s contents.

■ 38.12.4 PLAIN – what is really happening inside

- Why a rebase changes the hash. Recall the rule from section 38.3: a commit’s name is the hash of its bytes, and its bytes include the tree, the parent, both people, both timestamps and the message.
- A rebase does not move a commit. It cannot. It builds a new commit with the same message and the same changes but a different parent.
- Here is exactly that, done in the sandbox. A branch `fix` was created off `main`, then `main` moved on, then `fix` was rebased.
- Before the rebase:

```
tree 8376d5522743401a5e02cb99e72435cb6930e5a6
parent 7fc6cf85594a850b727418867d4054af04412e88
author KedByte Reader <reader@example.com> 1767760200 +0530
committer KedByte Reader <reader@example.com> 1767760200 +0530

Fix: add docs/fix.md
```

Its hash was `26c500778e51ea3a240b916e1ca97c33f2c37f55`.

- After the rebase:

```
tree 353596612537fa8a5e94b31dfa28bd33108ad139
parent 39bf0c8354d383aaf21ff05eaff4304acc24f346
author KedByte Reader <reader@example.com> 1767760200 +0530
committer KedByte Reader <reader@example.com> 1767767400 +0530

Fix: add docs/fix.md
```

Its hash is now 70a5d6220cc7944c8b3824701b4952027bc539af.

6. Compare the two blocks line by line. The message is identical. The author line is identical, including the original author timestamp.
7. Three things changed: the parent, because it is now on top of main; the tree, because the snapshot now also contains main's newer file; and the committer timestamp, because the commit was recorded again just now.
8. Any one of those three would have been enough to change the hash. All three changed.
9. The old commit still exists. Nothing was destroyed:

```
$ git reflog fix
70a5d62 fix@{0}: rebase (finish): refs/heads/fix onto 39bf0c8
26c5007 fix@{1}: commit: Fix: add docs/fix.md
7fc6cf8 fix@{2}: branch: Created from main
$ git cat-file -t 26c5007
commit
```

10. So "the SHA changed after a rebase" is not really true. A different commit now exists, and the branch file was rewritten to point at it. Both commits are on disk.

■ 38.12.5 TECHNICAL - the engineer's version

1. Abbreviation length. Git's default is `core.abbrev=auto`, which picks a length long enough that a collision among the objects present is unlikely, with a floor of 7 characters. Small repositories therefore show 7.
2. A 7-character prefix is 28 bits, about 268 million values. By the birthday bound, ambiguity becomes likely once a repository holds a few tens of thousands of objects, which is why large repositories abbreviate to 10, 12 or more.
3. An abbreviation is only unique within one repository at one moment. Adding objects can make a previously unique prefix ambiguous. Never store an abbreviated hash as a durable identifier; store all 40 characters.
4. What actually updates `refs/remotes/origin/*`:

Event	Updates origin/main?
git fetch	yes
git pull	yes, it fetches first
Successful git push	yes
git ls-remote	no, reads only
Someone else pushing	no
Time passing	no

5. The session evidence, applied. During the outage the reader had cached `origin/main` as `0a95cc8`. That was a true statement about the past and carried no information about the present.
6. Two pushes in that session failed with `send-pack: unexpected disconnect while reading sideband packet`. The write may or may not have landed; a disconnect while reading the response

tells you nothing about the request. The correct next step is not to guess but to re-query with `git ls-remote origin refs/heads/<branch>` and compare hashes.

7. This is why verifying by hash beats verifying by name. A branch name is a pointer that anyone can move. A commit hash is the content itself. In the same session, continuous integration was checked by run identifier against the exact head SHA rather than by trusting a coloured tick next to a branch name, and that is the stronger check for exactly this reason.
8. Practical checklist for “is my local view of the remote correct”:

```
git rev-parse origin/main          # what I remember
git ls-remote origin refs/heads/main # what is true now
git fetch origin && git rev-parse origin/main
git log --oneline origin/main..main # mine, not theirs
git log --oneline main..origin/main # theirs, not mine
```

9. And after a rebase, the correct push is `git push --force-with-lease`, not `--force`. The lease form refuses if the remote moved since your last fetch, which converts a silent overwrite of someone else’s work into an error you can read.

■ 38.12.6 WORDS – remember these

1. Abbreviated object id — the first few characters of a hash — a unique prefix within one repository, length chosen by `core.abbrev`.
2. Remote-tracking ref — your cached copy of a remote branch pointer — a ref under `refs/remotes/`, updated only by fetch and push.
3. `ls-remote` — ask the server what it holds right now — a network query that lists the remote’s refs without changing anything locally.
4. Rebase — replay commits onto a new base — creates new commit objects with new parents, therefore new object ids.
5. `--force-with-lease` — force push, but only if nothing moved — a compare-and- swap on the remote ref against your last known value.

38.13 Why this design is good, and what it costs

■ 38.13.1 PLAIN – in simple words

1. The benefits all fall out of one decision: name things by their content.
2. **Deduplication across all of history.** The same file content, in any commit, on any branch, from any author, is stored once.
3. **Integrity for free.** Every read can be checked against the name. Damage and tampering are both detectable without any extra machinery.
4. **Cheap branching.** A branch is 41 bytes, so having twenty of them costs less than one photograph.
5. **Distributed operation with no coordination.** Two people can create commits at the same time, offline, with no server involved, and the results will never clash, because names are computed from content and not assigned by an authority.
6. That last point is why the reader could keep committing all through a network outage. Nothing in `git commit` needs a server.
7. The costs are real too.
8. Large binary files are handled badly. Rewriting history is confusing. And the whole model takes genuine effort to learn, which is why this chapter exists.

■ 38.13.2 PLAIN – a picture in your head

1. Think of two libraries in two cities that never phone each other.
2. Each catalogues books by a code derived from the book's exact text.
3. When they finally exchange catalogues, matching codes mean identical books, with no argument and no committee.
4. Different codes mean different books, even if the titles match.
5. Neither library needed permission from the other to catalogue anything.

Where this comparison breaks:

6. Real libraries have to decide which copy is the official one. Git also has to decide, and it does not decide by content. A human decides, by choosing what main points at on the server.
7. Content addressing removes the need to coordinate naming. It does not remove the need to agree on what is authoritative.

■ 38.13.3 PLAIN – a worked example

1. Deduplication, measured. In the sandbox, four files with three identical contents produced two blobs, not four.
2. Integrity, measured. A single bit was flipped in one loose object file:

```
$ git fsck
error: inflate: data stream error (incorrect data check)
error: unable to unpack header of .git/objects/6f/e3e873...
error: 6fe3e873...: object corrupt or missing
missing blob 6fe3e87371433ed417646e3cb2354cfb9cff08d3
```

3. That one was caught by zlib's own checksum. So a more careful attack was tried: a perfectly well-formed object, holding different content, written under the old name.

```
$ git cat-file -p 6fe3e87
The EVIL three words.

$ git fsck
error: 8d71e45d309cb1328988a64591163f5c4f8f43f1: hash-path
mismatch, found at: .git/objects/6f/e3e873...
missing blob 6fe3e87371433ed417646e3cb2354cfb9cff08d3
```

4. Read that carefully, because it contains both the strength and an important limitation.
5. The strength: `git fsck` computed the content's true name, `8d71e45...`, found it stored under `6fe3e87...`, and reported the mismatch. The tamper is caught with certainty.
6. The limitation, and it is honest: `git cat-file -p` handed back the evil content without complaint. Git does not re-verify every loose object on every ordinary read. Verification happens on transfer, when configured, and when you ask with `fsck`.
7. Cheap branching, measured: 1,000 branches created in 2.28 seconds, using 41,000 bytes of content.
8. Distributed operation: two commits created independently, offline, will have different hashes unless they are byte-for-byte identical, in which case they are the same commit and it does not matter.

■ 38.13.4 PLAIN – what is really happening inside

1. Now the costs, honestly.
2. **Large binary files.** Git stores whole content per version. A 50 MB video edited ten times is ten blobs. Compressed formats do not delta, so the packfile saves almost nothing.

3. Worse, a clone fetches all of history by default. The 500 MB stays in every clone forever, even after the file is deleted, because the old commits still reference it.
4. The workarounds are all partial: `git lfs`, which stores a pointer file in git and the bytes elsewhere; shallow clones with `--depth`; partial clones with `--filter=blob:none`; and rewriting history with `git filter-repo`, which changes every hash from the rewrite point onwards.
5. **History rewriting is confusing** precisely because it is not rewriting. It is creating parallel objects and moving pointers. The old objects linger, other people still have the old hashes, and any branch built on the old ones must itself be rebuilt.
6. **The model takes effort.** Most tools let you learn the commands without the data model. Git punishes that. Almost every confusing git error makes immediate sense once you know that commits are immutable snapshots and branches are movable pointers, and almost none of them make sense otherwise.
7. There are smaller costs too. Git tracks content, not files, so it has no record of a rename; it infers renames when displaying a diff, and the inference can be wrong. It records only the executable bit, so full permissions and file ownership need a separate tool. And empty directories cannot be represented at all, because a tree with no entries in it is simply not referenced.

■ 38.13.5 TECHNICAL – the engineer’s version

1. Benefits and their mechanisms, stated precisely:

Benefit	Mechanism
Global deduplication	key = hash of content
Integrity verification	re-hash and compare
O(1) branch creation	write one 41-byte ref
Offline commit	ids need no authority

2. The distributed property is worth stating exactly. Git needs no central allocator of identifiers because the identifier space is a hash of content, and independent parties producing identical content produce identical ids by construction. This is the same property that lets content-addressed systems such as IPFS and container image registries work.
3. Costs, with the mitigations and their prices:

Cost	Mitigation	Price of the mitigation
Big binaries	Git LFS	extra server, extra tooling
Deep history size	<code>--depth</code> clone	limited history locally
Blob bloat	<code>--filter=blob:none</code>	fetches on demand
Bad past commits	<code>filter-repo</code>	every later hash changes

4. `core.bigFileThreshold`, default 512 MiB, controls when git stops attempting delta compression. Below it, git still tries, and for compressed media it still fails to save anything.
5. Rename detection is a display-time heuristic controlled by `diff.renameLimit` and a similarity threshold, by default 50 percent. It is not stored, it is recomputed, and on very large changesets git may give up entirely and report the rename as a delete plus an add.
6. Where experts disagree: whether git’s monorepo behaviour at very large scale is a flaw in the model or an implementation gap. Facebook chose Mercurial in 2013 partly over this, and Microsoft built a virtual filesystem layer for the Windows repository, later replaced by scalar and partial

clone in git itself. Both camps agree the model is sound; they disagree about how far it scales without special tooling.

7. Established fact: the object model has not changed since 2005, other than the addition of the optional SHA-256 format. Active work: the SHA-256 transition and the refable ref back-end. Marketing claim territory: any statement that one hosting provider makes git itself faster. Hosting affects the network path, not the object model.

■ 38.13.6 WORDS - remember these

1. Monorepo — one repository holding many projects — a scaling pattern that stresses git's whole-history clone model.
2. Git LFS — a way to keep big files out of git — Large File Storage, which commits a small pointer file and stores the bytes on a separate server.
3. Partial clone — a clone that skips file contents until needed — enabled with `--filter=blob:none`, fetching blobs on demand.
4. Shallow clone — a clone with truncated history — `--depth <n>`, which omits older commits and their objects.
5. Rename detection — noticing a file moved — a similarity heuristic computed at diff time, never stored in any object.

38.98 Common wrong ideas

1. Wrong: a commit stores a diff, the changes you made. Right: a commit names one tree, which is a complete snapshot of every file. The second version of `docs/intro.md` in this chapter is stored as all 38 bytes of the new file, not as “add one line”.
2. Wrong: a branch contains commits. Right: a branch is a 41-byte file holding one commit's hash. The commits you see are the ones reachable by walking parent links backwards from there, and they may be reachable from twenty other branches too.
3. Wrong: git compresses by storing diffs between versions of a file. Right: git compresses in two independent ways. zlib compresses each object separately, and packfiles store some objects as deltas against any similar object, chosen by a size and window heuristic, not by version order. Neither is part of the history model.
4. Wrong: deleting a branch deletes its commits. Right: it deletes one small file. The commits stay until nothing references them, their reflog entries expire, and garbage collection runs, which by default is at least 30 days.
5. Wrong: `git commit --amend` edits the last commit. Right: it builds a new commit object with a new hash and moves the branch to it. The original is still on disk and is listed in the reflog.
6. Wrong: detached HEAD is an error state. Right: it is a normal state meaning HEAD names a commit instead of a branch. Every CI checkout runs this way. The only risk is losing sight of commits you make there.
7. Wrong: `origin/main` shows what is on the server. Right: it shows what was on the server the last time you fetched or pushed. Only `git ls-remote` or a fresh `git fetch` asks the server.
8. Wrong: rebasing moves commits onto a new base. Right: it creates new commit objects with new parents, new trees and new committer timestamps, therefore new hashes. The originals remain until they are collected.
9. Wrong: because git uses SHA-1, and SHA-1 is broken, git repositories can be forged. Right: SHA-1 collision resistance is broken in general, but since Git 2.13 in May 2017 git uses a collision-detecting SHA-1 that refuses known attack patterns, and a useful attack must also survive review and transport. The risk is real but it is not “anyone can forge a commit”.

10. Wrong: a tag is just a branch that does not move. Right: a lightweight tag is close to that, but an annotated tag is a separate stored object with its own hash, tagger, date, message and optional signature, and `git describe` treats the two differently.

38.99 Chapter summary in 20 lines

1. Git names every stored thing by hashing the thing itself. That single decision explains almost everything else.
2. The name is SHA-1 of `<type> <size>\0<content>`, printed as 40 hexadecimal characters, and you can reproduce it with `printf` and `sha1sum`.
3. SHA-1's collision resistance was publicly broken by CWI Amsterdam and Google on 23 February 2017, at about $2^{63.1}$ evaluations.
4. Git responded in version 2.13, May 2017, by making the collision-detecting SHA-1 of Marc Stevens and Dan Shumow the default; a SHA-256 object format exists but is still not the default in 2026.
5. There are four object types. A blob is file content with no name. A tree is one directory listing with modes. A commit names one tree and zero or more parents. A tag object is an annotated pointer.
6. Tree entries store the mode in ASCII, the name, a zero byte, and 20 raw bytes of hash. Only five modes are legal, including 120000 for symlinks and 160000 for submodules.
7. Changing one file creates one new blob, one new tree per directory level up to the root, and one new commit. Everything else is pointed at again.
8. In the worked repository, editing `docs/intro.md` created 4 new objects and reused 4 existing ones.
9. A commit's hash covers its parent's hash, so one tip hash seals the entire history behind it. History is immutable because names are derived from content.
10. History is a directed acyclic graph, not a line. Merge commits have two or more parents, and `^1` and `^2` select between them.
11. A branch is a file of 40 hexadecimal characters and a newline. Creating one writes 41 bytes; 1,000 branches took 2.28 seconds and 41,000 bytes.
12. Refs may be packed into `.git/packed-refs`, in which case the individual files vanish. A loose file always wins over the packed entry.
13. `HEAD` normally holds `ref: refs/heads/main`. When it holds a raw hash you are detached, which is normal, not an error, and recoverable.
14. Other refs live under `refs/tags`, `refs/remotes` and `refs/notes`, plus pseudo-refs such as `ORIG_HEAD` and `FETCH_HEAD`.
15. The reflog records every ref movement locally, keeps reachable entries 90 days and unreachable ones 30, and turns a catastrophic `git reset --hard` into a one-line recovery with `git reset --hard main@{1}`.
16. Plumbing commands show the machinery: `cat-file`, `rev-parse`, `hash-object`, `ls-tree`, `rev-list`, `count-objects`, `verify-pack`. Porcelain commands are for people and their output may change.
17. Storage is loose objects first, then packfiles. In the worked example 21.42 MiB of object content became one packfile of 85,423 bytes, with 198 of 200 blobs stored as deltas up to depth 50.
18. Deltas are storage only. Git's model is still whole snapshots, and the object ids are identical either way.
19. The reader's `origin/main` of `0a95cc8` was a cached local memory, not a live fact; only `git ls-remote` or a fresh fetch asks the server.
20. The rebased branch had to get a new hash, because its parent, its tree and its committer timestamp all changed, and all three go into the hash.

The Index, Push, Fetch, Merge and Rebase

39.0 What this chapter gives you

1. You will be able to name the three places a file can live in git, working directory, index and object database, and say which command moves content between which pair.
2. You will be able to open `.git/index`, read its header by hand, and explain why it is a full snapshot with stat data and not a list of differences.
3. You will be able to prove that `git add` writes a real object to disk immediately, before any commit exists, and find that object yourself.
4. You will be able to describe the three steps `git commit` performs, write a tree, make a commit, move the branch ref, and do all three by hand with plumbing commands.
5. You will be able to choose correctly between `git restore`, `git reset` in its three modes, `git revert`, `git clean` and `git stash`, and say for each one whether it can lose work.
6. You will be able to describe exactly what crosses the network on a push: capability advertisement, have and want negotiation, one packfile, pkt-line framing, and three sideband channels.
7. You will be able to explain the reader's own failure, `send-pack: unexpected disconnect while reading sideband packet`, say which stage it happened at, and say why it tells you nothing about whether the write landed.
8. You will be able to state the rule for non-idempotent operations over unreliable transports, and apply it beyond git.
9. You will be able to explain merge base, three-way merge, fast-forward, conflict markers, and why a clean text merge can still produce broken code.
10. You will be able to explain why rebase changes a branch's SHA even though the changes are identical, and decide when rebasing is safe.

39.1 The three states and three places

■ 39.1.1 PLAIN – in simple words

1. A file that git is tracking exists in three different places at once.
2. The first place is the **working directory**. That is the ordinary folder on your disk, holding the files you open and edit.
3. The second place is the **index**, also called the staging area. It holds the exact version of every file that your next commit will contain.
4. The third place is the **object database**, inside `.git/objects`. That is the permanent store of every version git has ever been told to keep.
5. A fourth thing sits alongside these three, called **HEAD**. It is a pointer saying which commit you currently have checked out.

6. The three copies of a file are often identical. When they are, git says the working tree is clean and prints nothing interesting.
7. When they differ, git tells you which pair differs. That is the whole of `git status`.
8. "Changes to be committed" means index differs from HEAD.
9. "Changes not staged for commit" means working directory differs from index.
10. "Untracked files" means the file is in the working directory and in neither of the other two places.

■ 39.1.2 PLAIN – a picture in your head

1. Think of posting a parcel at a counter.
2. Your desk at home is the working directory. Things lie around, half sorted, in whatever state you left them.
3. The counter at the post office is the index. You put on the counter only the items you actually want to send.
4. You can put one item on the counter, go home, change your mind about a second item, and come back. The counter keeps what you already placed.
5. The sealed and stamped parcel is the commit. Once sealed, its contents are fixed forever.
6. The archive room behind the counter is the object database. Every parcel ever sealed is stored there.
7. The clerk's note saying which parcel you are currently working from is HEAD.

Where this comparison breaks:

8. At a real post office the item leaves your desk when you put it on the counter. In git, staging a file copies it. Your working file stays put.
9. A real counter holds items. The git index holds a complete list of every tracked file, not only the changed ones.
10. A real parcel is opened by the receiver. A git commit is never opened and changed. It is replaced by a new one.

■ 39.1.3 PLAIN – a worked example

1. Here is a real repository with one change of every kind in it, and its real `git status` output.

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   TODO.md
    modified:   calc.py
    deleted:    notes.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes)
    modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    out.log
```

2. Map every line onto the three places.
3. `new file: TODO.md`. Not in HEAD. Present in the index. So index differs from HEAD, and the difference is an addition.

- Nothing in this picture ever edits a stored object. Objects are written once and never modified.

■ 39.1.5 TECHNICAL – the engineer’s version

- The three states are formally: modified, staged, committed. The three storage locations are the working tree, the index, and the object store.
- The index is a single binary file at \$GIT_DIR/index, by default .git/index. Its location can be overridden by GIT_INDEX_FILE.
- git status is implemented as two diffs: diff-index --cached HEAD for the staged column and diff-files for the unstaged column, plus a directory walk for untracked paths.
- The machine-readable form is git status --porcelain=v2, stable across versions and intended for scripts. Here is the same repository state.

```
$ git status --porcelain=v2 --branch
# branch.oid 66b84b4f79e5cce670a6a2c0f739d5433403423e
# branch.head main
1 .M N... 100644 100644 100644 1af06666... 1af06666... README.md
1 A. N... 000000 100644 100644 00000000... 1333ed77... TODO.md
1 M. N... 100644 100644 100644 4693ad3c... 6cddd28b... calc.py
1 D. N... 100644 000000 000000 fb188b9e... 00000000... notes.txt
? out.log
```

- Columns after the XY code are: submodule state, mode in HEAD, mode in index, mode in worktree, object name in HEAD, object name in index, path. Hashes are abbreviated here to fit the page; the real output prints all 40 digits.
- Note TODO.md has HEAD mode 000000 and HEAD hash all zeros. That is how git encodes “not present in HEAD”.

Status word	Which two places differ	Direction
to be committed	index vs HEAD	staged
not staged	worktree vs index	unstaged
untracked	in worktree only	unknown to git
ignored	worktree, matched gitignore	suppressed

- Commands that observe this: git status, git diff (worktree vs index), git diff --cached (index vs HEAD), git diff HEAD (worktree vs HEAD), git ls-files -s, git diff-index, git diff-files.

■ 39.1.6 WORDS – remember these

- Working directory — the files you edit — the checked-out worktree on the filesystem, outside .git.
- Index — the counter where you put things to send — the binary staging file .git/index holding the next commit’s full snapshot.
- Object database — the archive of everything kept — .git/objects, holding loose and packed blob, tree, commit and tag objects.
- HEAD — the note saying where you are — a symbolic ref, usually pointing at a branch ref under refs/heads/.
- Staged — placed on the counter — present in the index with content differing from HEAD.
- Untracked — git has never been told about it — a path in the worktree with no index entry and no HEAD entry.

39.2 What the index actually is

■ 39.2.1 PLAIN – in simple words

1. The index is not an idea. It is a real file on your disk, and you can look at it.
2. It is at `.git/index`. It is binary, so opening it in a text editor shows rubbish, but its structure is documented and simple.
3. It holds one entry per tracked file. Not one entry per changed file. Every single tracked file, always.
4. Each entry stores the file's path, the name of its stored content, its permission bits, and a copy of what the operating system said about the file the last time git looked.
5. That last part is the clever bit. Git remembers the file's size, its last modified time, and its position on disk.
6. Next time you run `git status`, git asks the operating system for those same facts again. If they match, git assumes the file did not change and does not read it at all.
7. That is why `git status` is fast even in a project with tens of thousands of files. Most files are never opened.
8. The index stores whole snapshots, not differences. There is no "diff" anywhere in it.

■ 39.2.2 PLAIN – a picture in your head

1. Think of a hotel cloakroom with a big register book.
2. Every guest's coat has a ticket. The register lists, for each coat, the ticket number, the peg it hangs on, its colour, and its weight.
3. When you come back, the clerk does not take every coat down and inspect it.
4. The clerk glances at the register and at the pegs. If the weights and colours still match, nothing has changed and the clerk says so instantly.
5. Only if something looks different does the clerk take a coat down and check it properly.
6. The register is the index. The weights and colours are the stat data. The coats themselves are the stored objects.

Where this comparison breaks:

7. A clerk could be fooled by two coats of identical weight and colour. Git can be fooled too, and this is real: if a file changes within the same second and keeps the same size, git can miss it. Git has a "racy timestamp" rule to handle that case, described later in this section.
8. A cloakroom register lists only coats currently held. The git index lists every tracked file, even ones you have not touched in years.

■ 39.2.3 PLAIN – a worked example

1. Here is a small repository with five tracked files. First its size, then its contents.

```
$ ls -l .git/index
-rw-r--r-- 1 root root 479 .git/index

$ git ls-files -s
100644 1af06666...9965 0    README.md
100644 1333ed77...ee69 0    TODO.md
100644 6cddd28b...7159 0    calc.py
100644 b04e7afa...7b19 0    docs/guide.md
100644 11b15b1a...a0d5 0    src/main.py
```

2. The columns are: file mode, object name of the content, stage number, path. Stage 0 means "not conflicted". Hashes shortened here to fit the page.

3. Note docs/guide.md and src/main.py. The index has no directories. It has flat paths with slashes in them.
4. Now the stat cache. The --debug flag prints it.

```
$ git ls-files --debug | head -7
README.md
  ctime: 1786606646:156685960
  mtime: 1786606646:156685960
  dev: 65024    ino: 1049252
  uid: 0      gid: 0
  size: 13    flags: 0
TODO.md
```

5. mtime is when the content last changed. ctime is when the file's metadata last changed. ino is the inode number, the file's identity on the filesystem. size is 13 bytes.
6. Every one of those numbers came from a single stat call to the operating system, which is far cheaper than reading and hashing the file.
7. Here is the header of that binary file, byte by byte.

```
$ head -c 16 .git/index | od -A d -t x1z
00000000 44 49 52 43 00 00 00 02 00 00 00 05 >DIRC.....<
```

8. 44 49 52 43 is the ASCII letters DIRC. Then 00 00 00 02 is version 2. Then 00 00 00 05 is five entries.

■ 39.2.4 PLAIN – what is really happening inside

1. The file has three parts: a 12-byte header, then the entries, then optional extra sections, then a checksum at the end.
2. Each entry begins with 40 bytes of numbers copied from the operating system's stat call: two timestamps with nanoseconds, device, inode, mode, user, group and size.
3. Then 20 bytes holding the content's SHA-1 name. Then 2 bytes of flags, including the path length and the merge stage.
4. Then the path itself, followed by at least one zero byte, padded so the whole entry length is a multiple of eight.
5. The entries are sorted by path, as raw bytes. That sorting is what lets git find an entry by binary search instead of scanning.
6. Let us add it up for the five-file repository above. Fixed part is 62 bytes. README.md is 9 characters, so 62 plus 9 is 71, padded up to 72.
7. Doing that for all five paths gives 72, 72, 72, 80 and 80, which is 376.
8. Add the 12-byte header and the 20-byte trailing checksum and we have 408. The file is 479. The missing 71 bytes are one extension section.
9. That section is the cached tree, signature TREE, 63 bytes of data plus its own 8-byte header. It remembers tree objects already computed, so a commit does not have to rebuild trees for unchanged directories.
10. When git status runs, for each entry git calls lstat on the path. If size, mtime, ctime and inode all match, git marks the file unchanged without reading a single byte of it.

■ 39.2.5 TECHNICAL – the engineer’s version

1. The format is specified in the git documentation page `git format-index`. The signature is the four bytes DIRC, standing for “directory cache”.
2. Supported versions are 2, 3 and 4. Version 3 adds a second flags word for skip-worktree and intent-to-add. Version 4 prefix-compresses path names.
3. All multi-byte numbers are big-endian (“network byte order”). The trailing checksum is a hash over everything before it.

Field group	Bytes	Contents
header	12	DIRC, version, entry count
entry, fixed part	62	stat data, OID, flags
entry, variable	var	path, NUL, pad to 8
trailer	20	SHA-1 of the whole file

4. Here is that exact five-entry index parsed by a short script, showing the real numbers.

```
total size: 479
signature: DIRC version: 2 entries: 5
  README.md    mode=100644 size= 13 stage=0 bytes=72
  TODO.md      mode=100644 size=  5 stage=0 bytes=72
  calc.py       mode=100644 size= 65 stage=0 bytes=72
  docs/guide.md mode=100644 size= 12 stage=0 bytes=80
  src/main.py   mode=100644 size= 15 stage=0 bytes=80
offset after entries: 388
extension: TREE size: 63
```

5. Only two file modes are legal for regular files, 100644 and 100755. 120000 is a symbolic link and 160000 is a gitlink (a submodule).
6. The stat cache is a cache, so it can be wrong. The known hazard is the “racy index” problem: a file written in the same second as the index itself may have a matching mtime yet different content. Git handles this by treating any entry whose mtime is not strictly older than the index’s own mtime as suspect, and re-reading it.
7. Real measurements, taken on a Linux sandbox with git 2.43.0, on a repository with 10,000 tracked files totalling 2,850,000 bytes:

Situation	git status time
stat cache valid	20 to 25 ms
every mtime invalidated	89 to 103 ms
hashing all 10,000 files	77 ms

8. The index for those 10,000 entries was 801,570 bytes, about 80 bytes per entry. The honest version: this repository is tiny, so the gap is only about four times. On a repository with large files the gap is far larger, because the cost of re-reading grows with total bytes while the stat calls do not.
9. Since git 2.36, released 18 April 2022, git ships a built-in filesystem monitor daemon, enabled with `git config core.fsmonitor true`. It replaces the per-file `lstat` scan with change notifications from the operating system, and matters most on macOS and Windows.
10. Sparse-checkout with a sparse index, `extensions.sparseIndex`, allows directory entries in the index with mode `040000`, so that a huge repository does not need an index entry per file.

■ 39.2.6 WORDS – remember these

1. Index entry — one line in the register — a 62-byte fixed record plus a padded path, holding stat data, OID, mode and flags.
2. Stat data — what the filesystem says about a file — `st_mtime`, `st_ctime`, `st_dev`, `st_ino`, `st_size` and mode from `lstat(2)`.
3. Stat cache — remembering that so you need not re-read — the mechanism that makes `git status` skip unchanged files.
4. DIRC — the four letters at the start — the index file signature, short for “directory cache”, present since git’s first commit in April 2005.
5. Stage number — which side of a conflict — 0 for normal, 1 for merge base, 2 for ours, 3 for theirs.
6. Cached tree — the remembered directory hashes — the TREE index extension that lets `git write-tree` skip unchanged directories.

39.3 What `git add` really does

■ 39.3.1 PLAIN – in simple words

1. Most people believe `git add` marks a file for later. It does much more than that, and it does it at once.
2. When you run `git add`, git reads the file, compresses its content, works out its name from the content, and writes that content into `.git/objects` straight away.
3. Then it writes an entry in the index recording that name against the path.
4. So the content is already safely stored before you type `git commit`. The commit does not copy your files; the copy has already happened.
5. This is why staging a file and then editing it again gives you three different versions at once: the committed one, the staged one, and the one on disk.
6. It also means you can stage part of a file. Git can build a version that exists nowhere on disk, store it, and put that in the index.
7. That partial staging is done with `git add -p`, which walks you through each chunk of change and asks yes or no.

■ 39.3.2 PLAIN – a picture in your head

1. Imagine a photocopier next to your desk that also files things.
2. When you say “add this page”, the machine does not put a sticky note on your page. It photocopies it immediately and files the copy in a cabinet.
3. It then writes in a ledger: “for the page called `report.txt`, use copy number `eb2fc3ca`”.
4. You can now scribble all over your original. The copy in the cabinet is untouched.
5. If you later say “add this page” again, a new copy is made and the ledger line is updated to point at the new copy. The old copy stays in the cabinet, unreferenced.

Where this comparison breaks:

6. A photocopier makes a new copy every time. Git checks first: if a copy with that exact content already exists, it stores nothing and reuses it.
7. The ledger in this picture holds one line per page you added. The real index holds one line per tracked file, added or not.

■ 39.3.3 PLAIN – a worked example

1. Start a completely empty repository and prove that `git add` writes to `.git/objects` before any commit exists.

```
$ git init -q .
$ find .git/objects -type f
(nothing)

$ printf 'hello index\n' > greeting.txt
$ git hash-object greeting.txt
eb2fc3ca2f129a710df1a6c0fd5ebfd088a10bfd

$ git add greeting.txt

$ find .git/objects -type f
.git/objects/eb/2fc3ca2f129a710df1a6c0fd5ebfd088a10bfd

$ git log --oneline
fatal: your current branch 'main' does not have any commits yet

$ git ls-files -s
100644 eb2fc3ca2f...bfd 0    greeting.txt
```

2. Read that carefully. There is no commit. There is no branch tip. Yet the content is already a permanent object in the database.
3. `git hash-object` printed the name before the add, without writing anything. It is the same name, because the name is computed from the content.
4. Now the three-version demonstration. Take a file with two separate changes and stage only the first.

```
$ git add -p
@@ -1,6 +1,6 @@
    import os

    -PORT = 8080
    +PORT = 9090
    DEBUG = False

(1/2) Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]? y
@@ -8,3 +8,4 @@ def start():
    def stop():
        print("stopping")
+    print("goodbye")
(2/2) Stage this hunk [y,n,q,a,d,K,g,/,e,?]? n
```

5. Now ask git for the name of `server.py` in all three places.

```
committed : 6802ebd3762732a47322b13ed2d9510a9145a23f
index      : 345790221885e55f73d7e6c21d54fffc51998330
working dir: 154a7a3ee775a1746b842017a179c42cee649a30
```

6. Three different names, so three different contents. The middle one exists nowhere on your disk as a file. It is a real object in `.git/objects`.

■ 39.3.4 PLAIN – what is really happening inside

1. `git add path` performs these steps, in order, for each path.
2. Call `lstat` on the path to get size, mode and timestamps.

3. Read the file's bytes. Apply any content filters that are configured, for example line-ending conversion or a clean filter.
4. Build the object header: the word `blob`, a space, the byte count in decimal, and one zero byte. Put the content after it.
5. Compute SHA-1 over header plus content. That 40-character result is the object name.
6. Check whether an object with that name already exists. If it does, do nothing more; the content is already stored.
7. If it does not, compress the header-plus-content with `zlib` and write it to `.git/objects/ab/cdef...`, where `ab` is the first two characters.
8. Write or update the index entry: path, the new object name, the mode, and the freshly captured stat data.
9. Write the whole index back out, atomically, by writing `.git/index.lock` and renaming it over `.git/index`.
10. For `git add -p`, `git` first computes the diff, splits it into hunks, and for each accepted hunk applies it to a copy of the staged version. The result of applying the accepted subset is what gets hashed and stored.

■ 39.3.5 TECHNICAL - the engineer's version

1. The plumbing equivalents are `git update-index --add --cacheinfo` and `git hash-object -w`. The porcelain `git add` is a wrapper over both.
2. `git hash-object` without `-w` computes and prints the OID without writing. With `-w` it writes the loose object.
3. Loose objects are `zlib`-compressed with the default compression level set by `core.compression`, default `-1` meaning `zlib`'s own default of 6.
4. Useful `git add` variants:

Command	Effect
<code>git add -p</code>	interactive per-hunk staging
<code>git add -u</code>	stage changes to tracked files only
<code>git add -A</code>	stage everything, including deletions
<code>git add -N</code>	record intent to add, no content
<code>git add -e</code>	edit the diff by hand before staging

5. `git add -N` sets the intent-to-add flag, which requires index version 3. It makes an untracked file appear in `git diff` output without staging content.
6. In `git add -p`, the keys are: `y` stage, `n` skip, `q` quit, `a` stage this and all remaining in the file, `d` skip this and all remaining in the file, `s` split the hunk, `e` edit the hunk, `j/k` move without deciding.
7. `git add --interactive` was written by Junio C Hamano and merged on 18 December 2006; it shipped in `git 1.5.0` on 14 February 2007.
8. Filters that change content on the way in are set by `gitattributes`: `text` and `eol` for line endings, and `filter.<name>.clean` for custom transforms such as Git LFS. The honest version: with a clean filter configured, the blob stored is not the bytes on your disk.
9. To see what would be staged without staging it, use `git diff` for unstaged changes and `git add --dry-run -A`.

■ 39.3.6 WORDS – remember these

1. Blob — the stored content of one file — a git object of type blob, containing raw bytes with no filename.
2. Hunks — one chunk of change in a diff — a @@ section with its context lines, the unit `git add -p` operates on.
3. Loose object — one file per object — an object stored individually under `.git/objects/xx/`, as opposed to inside a packfile.
4. Intent to add — “this file is coming” — the index flag set by `git add -N`, registering a path with no content.
5. Clean filter — a transform applied on the way in — `filter.<name>.clean` from `gitattributes`, run before hashing.
6. Atomic index write — no half-written index — the lock-file-and-rename sequence git uses for `.git/index`.

39.4 What `git commit` really does

■ 39.4.1 PLAIN – in simple words

1. `git commit` does exactly three things, in this order.
2. First, it turns the index into tree objects. One tree for the top folder, and one for each subfolder, each listing names, modes and object names.
3. Second, it creates one commit object. That object holds the name of the top tree, the name of the commit you were on, who you are, when it is, and your message.
4. Third, it changes one small text file, the branch file, so it holds the new commit’s name.
5. That is all. No files are copied. The blobs were already written when you ran `git add`.
6. Because the branch file is the last thing to change, the commit is either fully there or not there at all.
7. If a step fails halfway, the objects written so far are simply unreferenced and will be cleaned up later. Nothing is corrupted.

■ 39.4.2 PLAIN – a picture in your head

1. Think of publishing an edition of a newspaper.
2. All the articles are already typeset and stored, one file each. That happened when you staged them.
3. Making the edition means writing a contents page listing which article file goes on which page. That is the tree.
4. Then you write a cover sheet: edition number, date, editor’s name, a sentence about what is in it, and the number of the previous edition. That is the commit.
5. Then you update the single sign on the wall that says “current edition”. That is the branch file.

Where this comparison breaks:

6. A newspaper’s contents page is written fresh each time. Git reuses tree objects for any folder whose contents did not change at all.
7. A newspaper edition can be recalled and reprinted. A commit object can never be edited. `git commit --amend` makes a completely new one.

■ 39.4.3 PLAIN – a worked example

1. Here is a commit done by hand, one plumbing command per step, so you can see the three stages separately.

```
$ git add greeting.txt

$ git write-tree
14a7508225642503557d7347c657e485225656a1

$ git commit-tree 14a75082... -p HEAD -m "second line"
d5d96f0f76b0571dd21802be5196808779e4b9ba

$ git update-ref refs/heads/main d5d96f0f...

$ git log --oneline
d5d96f0 second line
8c96e11 add greeting
```

2. Step one made a tree from the index. Step two made a commit pointing at that tree and at the old HEAD. Step three moved the branch.
3. `git commit` is those three commands with a message editor bolted on.
4. Now watch the three objects appear for a first commit in an empty repository. Before the commit there is one object, the blob.

```
before commit:
  eb2fc3ca... (blob)

after commit:
  856b2713... (tree)
  8c96e114... (commit)
  eb2fc3ca... (blob)
```

5. The commit object itself is plain text. Here it is.

```
$ git cat-file -p HEAD
tree 856b271361271d5c57e0ac557a10cc6961ea6806
author Reader <reader@example.com> 1772429400 +0530
committer Reader <reader@example.com> 1772429400 +0530

add greeting
```

6. And the branch file went from not existing at all to holding 40 characters plus a newline.

■ 39.4.4 PLAIN – what is really happening inside

1. Step one, `write-tree`, walks the index in sorted path order.
2. Whenever the path prefix changes, it knows a directory ended, so it builds the tree object for that directory and adds it as an entry in its parent.
3. If the cached tree extension says a directory is unchanged, git skips all of that and reuses the remembered tree name.
4. Step two builds the commit's bytes: the word `tree` and the tree name, then one parent line per parent, then author, then committer, then a blank line, then the message.
5. That text is hashed the same way a blob is, with a header saying `commit` and the byte count, and stored.
6. Step three writes the new commit's name into `.git/refs/heads/<branch>` and appends a line to `.git/logs/HEAD` and `.git/logs/refs/heads/<branch>`. Those log files are the reflog.

7. If HEAD is detached, meaning it holds a commit name directly rather than a branch name, step three writes to `.git/HEAD` instead.
8. Why does the index exist at all? Because it lets you decide what goes in a commit separately from what is on your disk.
9. Without it, either every commit contains everything you have changed, or the tool must ask you a list of files every single time.
10. The honest counter-argument: other systems manage without it. Mercurial has no staging area by default and uses `hg commit --interactive` for the same job. Subversion and Perforce have change lists instead. Fossil and Darcs ask about patches at commit time. So the index is a design choice, not a necessity, and reasonable people call it an unnecessary extra concept.

■ 39.4.5 TECHNICAL – the engineer’s version

1. The three plumbing commands are `git write-tree`, `git commit-tree` and `git update-ref`. Every porcelain commit path goes through equivalents.
2. `git commit --amend` does not modify the current commit. It builds a new commit with the same parents and moves the branch. The old commit remains reachable only through the reflog.
3. The commit object’s canonical byte layout, in order: `tree <oid>`, zero or more parent `<oid>`, author `<name> <email> <unix-ts> <tz>`, committer ..., optional headers such as `gpgsig`, a blank line, message.
4. Author and committer differ whenever a commit is replayed. `Rebase`, `cherry pick` and `git am` all keep the author and rewrite the committer.
5. Ref updates go through the ref transaction machinery. With the files backend this is a `.lock` file plus rename per ref. An alternative `reftable` backend was integrated in git 2.45, released April 2024, and is selected at creation time with `git init --ref-format=reftable`.
6. Commit-related knobs worth knowing:

Setting or flag	Effect
<code>commit.gpgSign</code>	sign every commit
<code>git commit -a</code>	stage tracked modifications first
<code>git commit --amend</code>	replace the tip commit
<code>git commit --fixup X</code>	message that autosquash recognises
<code>core.hooksPath</code>	where pre-commit hooks live

7. Hooks that run: `pre-commit`, `prepare-commit-msg`, `commit-msg`, `post-commit`. A non-zero exit from the first three aborts the commit.
8. Historical note: the index was in git’s very first commit. On 7 April 2005 Linus Torvalds committed `cache.h`, containing `#define CACHE_SIGNATURE 0x44495243 /* "DIRC" */`, and the object store then lived in `.dircache/objects`. The tools were called `update-cache`, `write-tree` and `commit-tree`. The names changed; the design did not.

■ 39.4.6 WORDS – remember these

1. Tree object — a folder listing — a git object mapping names to modes and object IDs, one per directory.
2. Commit object — the cover sheet — a git object naming one tree, zero or more parents, author, committer and message.
3. Ref — the sign on the wall — a file under `.git/refs` (or a line in `packed-refs`) holding an object name.

4. Amend — replace the top edition — create a new commit with the old commit's parents and move the branch to it.
5. Committer versus author — who applied it versus who wrote it — two separate identity and timestamp fields in every commit.
6. Reflog — the local diary of ref movements — append-only files under `.git/logs`, local only, never pushed.

39.5 Undoing things, laid out clearly

■ 39.5.1 PLAIN - in simple words

1. Git has many undo commands because there are many different things to undo.
2. The trick is to ask: which of the three places do I want to change?
3. `git restore <file>` changes the working directory only. It copies the staged version over your file, throwing your edits away.
4. `git restore --staged <file>` changes the index only. It copies the HEAD version into the index, so the file is no longer staged.
5. `git reset --soft <commit>` moves HEAD and the branch only. Index and working directory are untouched.
6. `git reset --mixed <commit>`, which is the default, moves HEAD and the branch and resets the index. Working directory untouched.
7. `git reset --hard <commit>` moves all three. Your uncommitted work is gone.
8. `git revert <commit>` changes nothing that already exists. It makes a new commit that undoes an old one. This is the safe undo for shared history.
9. `git clean` deletes untracked files. Nothing about them was ever stored, so they cannot be recovered.
10. `git stash` puts your uncommitted changes aside as hidden commits and gives you a clean tree.

■ 39.5.2 PLAIN - a picture in your head

1. Think of a kitchen with a chopping board, a tray, and a fridge.
2. The chopping board is the working directory. The tray is the index. The fridge is the object database.
3. `git restore` wipes the board and re-copies from the tray.
4. `git restore --staged` empties the tray and refills it from the fridge.
5. `git reset --soft` only moves the label saying which meal you are making. Board and tray keep everything.
6. `git reset --mixed` moves the label and clears the tray.
7. `git reset --hard` moves the label, clears the tray, and scrapes the board into the bin.
8. `git revert` does not touch anything. It cooks a new dish that cancels out an old one, and puts that in the fridge too.

Where this comparison breaks:

9. Scraping the board into a bin is final. `git reset --hard` losing a commit is not final, because the reflog remembers where the branch used to point for 90 days. Losing uncommitted edits, though, really is final.

■ 39.5.3 PLAIN – a worked example

1. Three commits, c1, c2, c3, where the file goes A, then A B, then A B C. Now watch each reset mode from the same starting point.

```

BASELINE (at c3)
  HEAD    = c3    index = "A B C"    workdir = "A B C"    status: clean

git reset --soft HEAD~1
  HEAD    = c2    index = "A B C"    workdir = "A B C"
  status: M f.txt      (staged modification)

git reset --mixed HEAD~1
  HEAD    = c2    index = "A B"      workdir = "A B C"
  status: M f.txt      (unstaged modification)

git reset --hard HEAD~1
  HEAD    = c2    index = "A B"      workdir = "A B"
  status: clean        (the "C" line is gone)

```

2. Read the three status lines. Soft leaves the change staged. Mixed leaves it unstaged. Hard destroys it.
3. Now the other four, in one real session.

```

$ git status --short
A  staged.txt
M  work.txt
?? build.tmp

$ git restore --staged staged.txt
M  work.txt
?? build.tmp
?? staged.txt

$ git restore work.txt
?? build.tmp
?? staged.txt

$ git clean -n
Would remove build.tmp
Would remove staged.txt

$ git clean -f
Removing build.tmp
Removing staged.txt

```

4. Note the -n dry run. Always run that first. `git clean -f` deletes files that git has never stored, so there is nothing to recover from.

■ 39.5.4 PLAIN – what is really happening inside

1. Here is the table. Read the middle column as “which of the three places does this touch”.

Command	What it changes	Can lose work
<code>git restore <f></code>	working dir only	yes, edits
<code>git restore --staged</code>	index only	no
<code>git reset --soft</code>	HEAD and branch	no

Command	What it changes	Can lose work
<code>git reset --mixed</code>	HEAD, branch, index	staged only
<code>git reset --hard</code>	HEAD, branch, index, wd	yes, badly
<code>git reset <f></code>	index entry for one path	no
<code>git revert <c></code>	adds a new commit	no
<code>git clean -f</code>	deletes untracked files	yes, forever
<code>git clean -fdx</code>	deletes ignored files too	yes, forever
<code>git stash</code>	wd and index into a stash	rarely
<code>git checkout <c></code>	HEAD only (detaches)	no

- Only two rows in that table can destroy something that git has never stored: `git reset --hard` and `git clean`. Everything else is recoverable.
- `git stash` is not magic. It makes real commits. One commit records the index, another records the working tree, and a ref called `refs/stash` points at the second one.

```
$ git stash list
stash@{0}: On main: half finished idea

$ git log --oneline --graph refs/stash
*   69aa036 On main: half finished idea
|\
| * 77c7641 index on main: b068de2 three hours of important work
|/
* b068de2 three hours of important work
```

- Now the recovery guide. Suppose you ran `git reset --hard HEAD~1` and threw away a commit.

```
$ git log --oneline
f74729c base                                <- the good commit is gone

$ git reflog
f74729c HEAD@{0}: reset: moving to HEAD~1
b068de2 HEAD@{1}: commit: three hours of important work
f74729c HEAD@{2}: commit (initial): base

$ git reset --hard 'HEAD@{1}'

$ git log --oneline
b068de2 three hours of important work
f74729c base
```

- The reflog is a local diary of every value HEAD and each branch has held. It is written on every commit, checkout, reset, merge, rebase and pull.
- Because the commit object was never deleted, only unreferenced, pointing a ref at it again brings it straight back.

■ 39.5.5 TECHNICAL - the engineer's version

- `git reset` has a fourth and fifth mode. `--merge` resets the index and updates worktree files that differ between HEAD and target while keeping unstaged changes. `--keep` is similar but aborts on conflict.
- `git reset <paths>` never moves HEAD. Path form and commit form are different operations sharing a name, which is a known wart. `git restore` and `git switch`, introduced in git 2.23 on

16 August 2019, exist to split those roles. They were marked experimental at introduction and remain so in the documentation as of git 2.43.

3. Reflog retention defaults: `gc.reflogExpire` is 90 days for reachable entries, `gc.reflogExpireUnreachable` is 30 days. `gc.pruneExpire` is 2 weeks, so unreachable objects survive at least that long before deletion.
4. Recovery ladder, in the order to try:

Situation	Command
branch moved wrongly	<code>git reflog then reset</code>
commit lost, not in reflog	<code>git fsck --lost-found</code>
dropped a stash	<code>git fsck --unreachable</code>
bad rebase, want the old branch	<code>git reset --hard ORIG_HEAD</code>
bad merge, not committed	<code>git merge --abort</code>
bad rebase, mid-flight	<code>git rebase --abort</code>

5. `ORIG_HEAD` is written by merge, rebase, reset and pull before they move HEAD. `git reset --hard ORIG_HEAD` is the fastest undo for those four.
6. Since git 2.36 (April 2022), `git branch --recurse-submodules` and related plumbing exist, but stash and submodules still interact badly; `git stash` does not stash submodule changes by default.
7. `git stash push --staged`, added in git 2.35 (24 January 2022), stashes only what is in the index. `git stash -u` includes untracked files; `git stash -a` includes ignored files.
8. `git stash` was added to git on 30 June 2007 and shipped in git 1.5.3 on 2 September 2007.

■ 39.5.6 WORDS – remember these

1. Soft reset — move the label only — move HEAD and branch, leave index and worktree.
2. Mixed reset — move the label and clear the tray — move HEAD and branch and reset the index; the default mode.
3. Hard reset — move everything and discard — overwrite index and worktree from the target commit.
4. Revert — undo by adding, not removing — create a new commit whose diff is the inverse of an existing commit.
5. `ORIG_HEAD` — where I was before that big operation — a ref written by merge, rebase, reset and pull.
6. Reflog — the local diary — per-ref append-only history of previous values, kept 90 days by default and never transferred over the network.

39.6 What actually crosses the wire on a push

■ 39.6.1 PLAIN – in simple words

1. When you push, git does not send files. It sends objects, and it sends only the ones the other side does not already have.
2. The first thing that happens is a conversation. Your git starts a program on the server called `receive-pack`.
3. The server speaks first. It lists every branch and tag it has, with the object name each one points at, and a list of features it supports.
4. Your git compares that list with what you are pushing. It works out which objects the server is missing.

5. Your git bundles those objects into one compressed file called a packfile, and sends it, together with a line saying “move branch main from this old name to this new name”.
6. The server reads the packfile, checks that nothing is missing or corrupt, then updates the branch.
7. Then the server sends a short report: whether the pack unpacked, and whether each branch update succeeded.
8. All of that is one connection and, for the object transfer, one round trip. Twenty commits cost the same number of round trips as one.

■ 39.6.2 PLAIN - a picture in your head

1. Think of sending documents to a records office by courier.
2. You phone first. The clerk reads out an index of everything already filed, and says which formats they accept.
3. You compare against your own shelf and put only the missing documents into one box. You do not send twenty separate envelopes.
4. On the box you write one instruction slip: “file these, then change the pointer for folder main from document 1c8a3b1 to document 9978003”.
5. The courier carries the box. The clerk unpacks it, checks nothing is missing, files everything, updates the pointer, and phones you back.
6. That phone call back is the report. It is a separate thing from the box.

Where this comparison breaks:

7. A courier can tell you the box arrived. Over a network there is no such guarantee. If the return phone call is cut off, you have no idea whether the clerk filed the documents. That is the whole of section 39.7.

■ 39.6.3 PLAIN - a worked example

1. This is a real push, traced with `GIT_TRACE_PACKET=1`. Lines are trimmed to fit the page; `>` means sent and `<` means received.

```

receive-pack> 9978003f... refs/heads/main\0report-status
                report-status-v2 delete-refs side-band-64k quiet
                atomic ofs-delta object-format=sha1 agent=git/2.43.0
receive-pack> 0000
push>          9978003f... e60afb23... refs/heads/main\0
                report-status-v2 side-band-64k quiet
                object-format=sha1 agent=git/2.43.0
push>          0000
                [ packfile bytes travel here ]
receive-pack> unpack ok
receive-pack> ok refs/heads/main
receive-pack> 0000
sideband<      \1000eunpack ok0017ok refs/heads/main0000

```

2. Line one is the advertisement. The server says what it has and what it can do. `0000` ends that list.
3. Line five is the command. Old name, new name, ref name, then the capabilities the client accepts.
4. Then the packfile is streamed.
5. Then the report: `unpack ok` and `ok refs/heads/main`.
6. Now look at the last line very carefully, because it shows the raw framing.
7. `\1` is the sideband channel number: channel 1, meaning real data.
8. `000e` is hexadecimal 14: this packet is 14 bytes long including its own 4-byte length. So 10 bytes of payload: `unpack ok` plus a newline.

9. 0017 is hexadecimal 23: 19 bytes of payload, ok refs/heads/main plus a newline.
10. 0000 is the flush packet, meaning “end of this stream”.
11. Now the round-trip claim, measured. Twenty commits, each changing one file.

```
$ git rev-list --count origin/main..main
20
$ git rev-list --objects origin/main..main | wc -l
60
$ git push --progress origin main
Enumerating objects: 62, done.
Counting objects: 100% (62/62), done.
Delta compression using up to 2 threads
Compressing objects: 100% (20/20), done.
Writing objects: 100% (60/60), 3.95 KiB | 674.00 KiB/s, done.
Total 60 (delta 0), reused 0 (delta 0), pack-reused 0
To file:///tmp/gw/srv.git
1c8a3b1..9978003 main -> main
```

12. Twenty commits became sixty objects, twenty commits plus twenty trees plus twenty blobs, in one packfile of 3.95 KiB, in one exchange.

■ 39.6.4 PLAIN – what is really happening inside

1. Everything on the wire is wrapped in a frame called a **pkt-line**.
2. A pkt-line is four hexadecimal characters giving the total length, then that many bytes minus four of payload.
3. So 0009hello is a complete packet: length 9, payload hello which is 5 bytes, plus the 4 length characters.
4. Three lengths are special. 0000 means flush, end of a section. 0001 is a delimiter used in the newer protocol. 0002 marks the end of a response.
5. On top of pkt-line sits the **sideband**. When both ends agree to it, the first byte of each packet’s payload is a channel number.
6. Channel 1 is real data, the packfile or the report. Channel 2 is progress text, which your git prints with the prefix remote:. Channel 3 is a fatal error, after which the stream stops.
7. This is why, on a real clone, you see lines beginning with remote:. Those came over channel 2.

```
$ git clone https://github.com/octocat/Hello-World.git
Cloning into 'Hello-World'...
remote: Enumerating objects: 13, done.
remote: Total 13 (delta 0), reused 0, pack-reused 13 (from 1)
Receiving objects: 100% (13/13), done.
```

8. remote: Enumerating objects is the server talking on channel 2. Receiving objects is your own git counting what it reads from channel 1.
9. For fetching, the two sides negotiate. The client says want <oid> for what it needs and have <oid> for tips it already holds. The server replies ACK for the ones it recognises.
10. That is how the server learns the common ancestor without either side sending the whole history.

■ 39.6.5 TECHNICAL – the engineer’s version

1. Transports: SSH runs git-upload-pack or git-receive-pack as a remote command on port 22. Smart HTTP uses two requests on port 443. The git:// daemon protocol uses port 9418 and has no authentication or encryption.
2. Real smart-HTTP trace against github.com, taken with GIT_TRACE_CURL=1, trimmed to the git-relevant headers.

```
=> GET /octocat/Hello-World.git/info/refs?service=git-upload-pack
=> User-Agent: git/2.43.0
=> Git-Protocol: version=2
<= HTTP/1.1 200 OK
<= Content-Type: application/x-git-upload-pack-advertisement

=> POST /octocat/Hello-World.git/git-upload-pack
=> Content-Type: application/x-git-upload-pack-request
=> Git-Protocol: version=2
<= HTTP/1.1 200 OK
<= Content-Type: application/x-git-upload-pack-result
<= Transfer-Encoding: chunked
```

3. The GET fetches the advertisement, the POST carries the negotiation and receives the packfile. Push uses the same shape with `git-receive-pack`.
4. Real negotiation for a fetch, over protocol v2, trimmed:

```
fetch> command=fetch
fetch> thin-pack
fetch> no-progress
fetch> ofs-delta
fetch> want fec0e27c2650e36592a05049f34cae60fc2966c0
fetch> have 9f1a39e109aca3fc1cce3033a70a43553ebd2bb1
fetch> have f27fb125fe41194710c653bb5b43870a725be980
fetch> 0000
fetch< acknowledgments
fetch< ready
fetch< 0001
fetch< packfile
```

5. Exact framing limits, from the `gitprotocol-common` specification: maximum pkt-line data is 65,516 bytes; maximum total packet length is 65,520 bytes.
6. Sideband limits, from `gitprotocol-capabilities`: side-band allows up to 1,000 bytes per packet, 999 of payload plus one band byte. side-band-64k allows up to 65,520 bytes, 65,519 of payload plus one band byte.

Capability	Meaning
report-status	send a per-ref result after push
report-status-v2	adds proc-receive rewrite reporting
side-band-64k	multiplex data, progress and errors
ofs-delta	deltas may reference by pack offset
atomic	all refs update or none do
delete-refs	zero OID means delete this ref
quiet	suppress server-side progress

7. Protocol v2 was introduced in git 2.18 on 21 June 2018, made default in git 2.26 on 22 March 2020, demoted again in 2.27 after bugs, and made default once more in git 2.29 on 19 October 2020.
8. Push still uses the v0-style advertisement in the trace above, taken with git 2.43.0. Since git 2.32 (6 June 2021), `push.negotiate=true` lets push use v2 negotiation to avoid sending objects the server already has.
9. Smart HTTP itself dates from git 1.6.6, released 23 December 2009, written largely by Shawn O. Pearce. Before that, HTTP fetch was a “dumb” walker that downloaded loose objects one at a time.

10. Observation tools: `GIT_TRACE_PACKET=1`, `GIT_TRACE=1`, `GIT_TRACE_CURL=1` with `GIT_TRACE_CURL_NO_DATA=1`, `GIT_SSH_COMMAND` with `ssh -v`, and `git ls-remote` for a pure read of the server's refs.
11. Be clear about what kind of rule each of these is. The pkt-line framing, the three sideband channels, the capability names and the smart-HTTP URL paths are a **standard**: they are written down in git's protocol documentation, and other implementations follow them.
12. The remote: prefix on band 2 output is an **implementation detail** of the git client, defined by `DISPLAY_PREFIX` in `sideband.c`. Nothing on the wire contains those eight characters.
13. Naming your main remote origin and your default branch main is a **convention**. Git does not care; the tooling around it often does.

■ 39.6.6 WORDS – remember these

1. pkt-line — a length-prefixed frame — 4 hex digits of total length followed by payload; 0000 is flush.
2. Sideband — three streams down one pipe — band 1 data, band 2 progress printed as `remote:`, band 3 fatal error.
3. Packfile — one compressed bundle of objects — the `.pack` format with deltas and an accompanying `.idx`.
4. receive-pack — the server side of push — the process that reads the pack and updates refs.
5. upload-pack — the server side of fetch — the process that negotiates and builds a pack for the client.
6. Capability advertisement — what each end can do — the space-separated list sent after a NUL byte on the first ref line.

39.7 The reader's own failure, analysed properly

■ 39.7.1 PLAIN – in simple words

1. Twice in one session, the reader's push stopped with this message.


```
send-pack: unexpected disconnect while reading sideband packet
fatal: the remote end hung up unexpectedly
```
2. Here is what that sentence means, word by word.
3. `send-pack` is the program on your own machine that does the pushing. So the complaint is coming from your side, not the server's.
4. `reading sideband packet` means your git had finished sending and had switched to listening for the answer.
5. `unexpected disconnect` means the connection went silent while it was listening. Not an error message. Silence.
6. So the stage is exactly this: the pack was already sent, and your git was waiting for the report that says whether it worked.
7. And now the crucial point. Your git never got the report. That is the only fact you have.
8. It does not tell you the push failed. It does not tell you the push succeeded. It tells you that the answer did not arrive.
9. The server may have taken the pack, checked it, moved the branch, and then died before it could say so.
10. Or the server may have died before doing any of that.
11. From your side those two look identical. The only way to find out is to ask the server again.

■ 39.7.2 PLAIN – a picture in your head

1. You post a signed contract by registered mail and wait for the receipt.
2. The receipt never comes.
3. Did the office receive and file your contract, and then lose the receipt in the post? Or did the envelope never arrive at all?
4. You genuinely cannot tell from your end. Both stories produce exactly the same experience: you sent something and heard nothing.
5. The only sane move is to phone the office and ask what they have on file.
6. Note what you must not do. You must not assume it failed and post a second contract, because if the first one landed you now have two.
7. And you must not assume it succeeded and move on, because if it did not you have lost the work.

Where this comparison breaks:

8. With a contract, a duplicate is a real problem. With a git push, re-pushing the identical commits is harmless, because the ref update either has already happened or will happen. That is a lucky property of git, not a general one.

■ 39.7.3 PLAIN – a worked example

1. This can be reproduced exactly. Here is a server that behaves normally, accepts the pack, updates the branch, and then drops the connection instead of delivering its report.

```
$ git ls-remote origin
(nothing: the server has no refs yet)

$ git push origin main
send-pack: unexpected disconnect while reading sideband packet
fatal: the remote end hung up unexpectedly
$ echo $?
128
```

2. That looks like a failure. Now ask the server.

```
$ git ls-remote origin
36e4614a1df6e4ff7839130240981cabb5b970ca    HEAD
36e4614a1df6e4ff7839130240981cabb5b970ca    refs/heads/main

$ git status -sb
## main...origin/main [gone]
```

3. The push worked. The branch is on the server. The error was about the reply, not about the write.
4. Also note origin/main [gone]. Your local copy of the server's state was never updated, because the report never arrived. Your machine still thinks nothing is there.
5. Now the second server, which dies before applying anything. Same client, same commit, same command.

```
$ git push origin main
send-pack: unexpected disconnect while reading sideband packet
fatal: the remote end hung up unexpectedly
$ echo $?
128

$ git ls-remote origin
(nothing)
```

6. Identical message. Identical exit code. Opposite outcome.
7. That is the whole lesson in one pair of transcripts. The error text carries no information about whether the write landed.
8. In the reader's own session, both failures had NOT applied. That was discovered by running `git ls-remote` and looking, not by reasoning from the error message. If it had been deduced from the message, the conclusion would have been luck, not knowledge.

■ 39.7.4 PLAIN – what is really happening inside

1. Sequence of events on a push, with the failure point marked.

```

1. client starts receive-pack on the server
2. server sends ref advertisement + capabilities -> flush
3. client sends "old new refname" -> flush
4. client streams the packfile
5. server reads pack, runs index-pack, verifies objects
6. server runs pre-receive hook
7. server updates refs/heads/main
8. server runs post-receive hook
9. server sends "unpack ok" and "ok refs/heads/main"
   <-- the connection dies anywhere from step 5 to step 9
10. client prints the report, updates origin/main

```

2. The client's error is raised in step 9. Everything from step 5 to step 8 may or may not have completed.
3. Concretely: in git's source, `send-pack.c` calls `recv_sideband("send-pack", ...)`. Inside `sideband.c`, when the packet reader returns end-of-file, it produces exactly the message the reader saw.
4. The word `send-pack` in the message is simply the name the caller passed in. It is a label, not a diagnosis.
5. Why does the connection die? Any of: an idle timeout on a middlebox, a carrier-grade NAT table entry expiring, a proxy closing an idle connection, the server process being killed, a load balancer draining, or a genuinely dropped path like the one in the reader's traceroute.
6. Note that step 4, streaming the pack, can take a long time with nothing coming back. That is exactly when idle-connection timers fire.
7. The general shape of the problem: a **non-idempotent** operation, meaning one where doing it twice is not the same as doing it once, sent over a transport that can fail after the request but before the reply.
8. Chapter 36 made the same point about HTTP APIs: `POST` is not idempotent, so a client that times out cannot safely retry, and that is precisely why idempotency keys exist. This is the same problem in a different suit.
9. Git is kinder than most, because pushing the same commits twice is harmless. But you still cannot know the state without asking.

■ 39.7.5 TECHNICAL – the engineer's version

1. The rule, stated properly: **a failure while reading a response tells you nothing about whether the request was applied**. Only the outcome of a fresh query does.
2. This is a two-generals situation. No number of extra acknowledgements makes the final one guaranteed to arrive.
3. Correct procedure after any interrupted push:

```

# what does the server actually hold right now?
git ls-remote origin refs/heads/<branch>

```

```
# what did I try to send?
git rev-parse HEAD

# refresh the remote-tracking refs, then read them
git fetch origin
git status -sb
```

- Do not use `git status` alone. Before a fetch, `origin/main` is a cached value written by the last successful exchange, not a live fact.
- Techniques that make this class of failure less likely or less painful:

Technique	What it buys you
<code>git push --atomic</code>	all-or-nothing across several refs
<code>--force-with-lease</code>	fails if the server moved unnoticed
<code>http.postBuffer</code> raise	fewer chunked-upload edge cases
<code>SSH ServerAliveInterval</code>	keeps NAT and proxy timers alive
<code>GIT_TRACE_PACKET=1</code>	shows the exact stage of failure
smaller pushes	shorter windows for a timeout

- `git push --atomic` uses the atomic capability so several ref updates land as one transaction. It does not solve the reporting problem; it only removes the partial-ref-update case.
- `--force-with-lease` is the strongest safety net here, because it encodes an expectation about the server's current value and fails if reality differs.
- Generalisation beyond git, worth memorising:

Operation type	Safe to blind-retry?	What to do instead
read (GET, fetch)	yes	just retry
idempotent write	yes	just retry
create (POST)	no	re-query, or use key
git push	in practice yes	re-query first

- The honest version: `git push` is described as “in practice safe to retry” because the second push either fast-forwards to the same value or reports “Everything up-to-date”. That is a property of the ref-update rule, not a guarantee written in a specification.
- In the reader's session, the diagnosis of the wider outage came from the same discipline: `curl -v https://github.com` printed `Trying 20.207.73.82:443...` and then timed out after 15 seconds with no reply of any kind, while the same request succeeded over mobile data. What was proven was “no response on this path”. What was not proven was who dropped it.

■ 39.7.6 WORDS – remember these

- Non-idempotent — doing it twice is not the same as once — an operation whose repetition changes the result.
- Two generals problem — you can never be sure the last message arrived — the classic impossibility result for agreement over a lossy channel.
- Re-query — go and look rather than guess — issue a fresh read to establish the true state after an ambiguous failure.
- Stale remote-tracking ref — your cached idea of the server — `origin/main` as written by the last successful fetch or push.

5. Atomic push — all refs or none — the `--atomic` flag, backed by the server's atomic capability.
6. Lease — an assertion about the server's current value — the expected old OID carried by `--force-with-lease`.

39.8 Fetch versus pull

■ 39.8.1 PLAIN – in simple words

1. `git fetch` downloads. That is all it does.
2. It brings any objects you are missing into `.git/objects`, and it updates your record of where the server's branches are.
3. It does not touch your files. It does not touch your branch. It does not touch your index. After a fetch, your working directory is byte for byte what it was before.
4. `git pull` is two commands in one: a fetch, then an integration step that combines the fetched work into your current branch.
5. That second step is where the surprises live, because it changes your files and can create commits or conflicts.
6. So `git fetch` is always safe to run. `git pull` is not always safe to run.
7. The usual advice is: fetch, look at what arrived, then decide how to integrate it.

■ 39.8.2 PLAIN – a picture in your head

1. Think of a shared noticeboard in an office, and your own copy of it in a folder.
2. `git fetch` is walking over, photographing the board, and filing the photo. Your desk is untouched. You now know what the board says.
3. `git pull` is walking over, photographing the board, and then immediately rewriting your own working notes to match it.
4. If your notes and the board disagree, that second act is where the mess happens, right there on your desk, without warning.

Where this comparison breaks:

5. A photograph is passive. A fetched object is not: it is a real object in your database that other commands can now reach and use.
6. Copying the board means overwriting. Git never overwrites your commits; it merges or replays. But it does overwrite your working files.

■ 39.8.3 PLAIN – a worked example

1. Here is a fetch, with the state of everything printed before and after.

```
before: HEAD=7f77069  origin/main=1303ba7
before: file f contains "A" and "from x1"

$ git fetch origin
From file:///tmp/gw/ff
 1303ba7..2c979a9  main      -> origin/main

after : HEAD=7f77069  origin/main=2c979a9
after : file f contains "A" and "from x1"

$ git status -sb
## main...origin/main [ahead 1, behind 1]
```


- Only one thing changed: origin/main. HEAD did not move. The file did not change. And now `git status` can tell you the truth: one commit each way.
- Now a pull in the same situation, on a modern git with nothing configured.

```
$ git pull
hint: You have divergent branches and need to specify how to
hint: reconcile them.
hint:
hint:   git config pull.rebase false  # merge
hint:   git config pull.rebase true   # rebase
hint:   git config pull.ff only       # fast-forward only
fatal: Need to specify how to reconcile divergent branches.
```

- That refusal is a good thing, and it is recent. Older git versions silently merged, which is how so many repositories acquired hundreds of commits reading “Merge branch ‘main’ of ...”.

■ 39.8.4 PLAIN – what is really happening inside

- `git fetch origin` does five things.
- Connect and get the ref advertisement.
- Work out which objects are missing, using want and have lines.
- Receive one packfile and store it under `.git/objects/pack/`.
- Update remote-tracking refs, that is `refs/remotes/origin/*`, to the server’s values.
- Write `FETCH_HEAD` and append to the reflog. Nothing else moves.
- `git pull` runs that, then runs either `git merge FETCH_HEAD` or `git rebase FETCH_HEAD`, depending on configuration.
- The surprise has three separate causes.
- First, the integration step is invisible in the command name. Nothing in the word “pull” says “and then merge”.
- Second, which integration happens depends on configuration that may differ between your machine and a colleague’s.
- Third, it acts on your working directory, so a bad outcome is immediately in your face, mid-task, possibly with conflicts.

■ 39.8.5 TECHNICAL – the engineer’s version

- `git pull` is documented as `git fetch` followed by either `git merge` or `git rebase`. The choice is made by `pull.rebase`, then `branch.<name>`. `rebase`, then the command-line flags.
- Recommended settings, with what each buys you:

Setting	Effect
<code>pull.ff only</code>	refuse to auto-merge
<code>pull.rebase true</code>	replay your work on top
<code>fetch.prune true</code>	delete gone remote-tracking
<code>push.default simple</code>	push only the current branch
<code>push.autoSetupRemote true</code>	first push sets upstream
<code>rebase.autoStash true</code>	stash and restore around it

- `push.default = simple` became the default in git 2.0, released 28 May 2014. `push.autoSetupRemote` arrived in git 2.37, released 27 June 2022.
- `git fetch --prune` deletes remote-tracking refs whose branches no longer exist upstream. Without it, `origin/old-feature` lingers forever.

5. `git fetch --all` fetches from every remote. `git remote update` is the older spelling of the same thing.
6. `FETCH_HEAD` is a file listing everything the last fetch brought, with the ones marked “not-for-merge” excluded from the integration step.
7. The `refspec` controls what is fetched. The default for a clone is `+refs/heads/*:refs/remotes/origin/*`. The leading `+` means “allow non-fast-forward updates to my remote-tracking refs”, which is why a force-push upstream does not break your fetch.
8. Shallow and partial fetches change the object set: `--depth=N` truncates history, `--filter=blob:none` omits file contents until needed. Partial clone requires the server capability `filter`, which GitHub’s advertisement in the earlier trace does list.
9. The honest version: experts disagree about `pull.rebase`. `Rebase-by-default` gives a clean line but rewrites your local commits every pull, which is confusing if you have already shared the branch. `Merge-by-default` is truthful but noisy. `pull.ff` only sidesteps the argument by making you choose each time, and is the safest default to teach.

■ 39.8.6 WORDS – remember these

1. Fetch — download and record — transfer missing objects and update `refs/remotes/*` only.
2. Pull — fetch plus integrate — fetch followed by merge or rebase into the current branch.
3. Remote-tracking ref — your note of where the server was — a ref under `refs/remotes/<remote>/`, updated only by fetch, push or remote commands.
4. Refspec — a mapping of their refs to mine — `+<src>:<dst>`, where `+` allows non-fast-forward updates.
5. `FETCH_HEAD` — what the last fetch brought — a file listing fetched tips, some marked not-for-merge.
6. Prune — forget branches that no longer exist — `git fetch --prune` or `fetch.prune true`.

39.9 Merge

■ 39.9.1 PLAIN – in simple words

1. Merging combines two lines of work into one.
2. Git starts by finding where the two lines split apart. That shared ancestor is called the **merge base**.
3. It then has three versions of every file: the base version, your version, and their version.
4. For each file it asks: did I change it, did they change it, or both?
5. If only one side changed a file, take that side. If neither changed it, keep the base. If both changed it, look closer.
6. If both changed different parts of the file, take both changes.
7. If both changed the same part, git stops and asks you. That is a conflict.
8. The result is stored as a new commit with two parents, one for each line that was joined.
9. There is one special case. If your branch has not moved at all since the split, there is nothing to combine. Git just moves your branch forward. That is called a **fast-forward**, and it creates no commit.

■ 39.9.2 PLAIN – a picture in your head

1. Two people are given identical copies of a report and go away to edit it.
2. When they come back, an editor puts three copies on the desk: the original, copy A and copy B.
3. For each paragraph the editor compares all three. If only A touched it, take A’s. If only B touched it, take B’s.

4. If both touched the same paragraph in different ways, the editor cannot decide and leaves both versions marked up for a human.
5. If B never actually changed anything, the editor simply hands over A's copy and calls it done. That is fast-forward.

Where this comparison breaks:

6. A human editor reads for meaning. Git compares lines of text and nothing else. Two changes that read fine together can still be wrong together, and git will never notice. Section 39.10 gives a real example.

■ 39.9.3 PLAIN – a worked example

1. Fast-forward first. Branch feature is ahead; main has not moved.

```
$ git log --oneline --graph --all
* 80b9042 C: feature file
* 3fc5061 B: on main
* 0a83b6e A: base

$ git merge-base main feature
3fc5061b9e3a37928186aab85f0c7ed2fc6b73c4

$ git merge-base --is-ancestor main feature && echo "ancestor"
ancestor

$ git merge feature
Updating 3fc5061..80b9042
Fast-forward
 feature.txt | 1 +
 1 file changed, 1 insertion(+)

$ git rev-list --merges --count HEAD
0
```

2. Zero merge commits were created, because main was already an ancestor of feature. Moving the label forward is enough.
3. Now a real three-way merge. Both branches changed the same file in different places.

```
BEFORE
* 1f96d1d B: rename header      (main)
| * 8681015 C: add contact line (feature)
|/
* 9db1830 A: base page

BASE (A)      OURS (main, B)  THEIRS (feature, C)
header        site header     header
body          body            body
footer        footer          footer
                              contact
```

4. Line 1 changed on our side only. Line 4 was added on their side only. No overlap, so git can take both.

```
$ git merge --no-edit feature
Auto-merging page.txt
Merge made by the 'ort' strategy.
page.txt | 1 +
```

```

AFTER
* 012c5e8 Merge branch 'feature'
|\
| * 8681015 C: add contact line
* | 1f96d1d B: rename header
|/
* 9db1830 A: base page

$ cat page.txt
site header
body
footer
contact

```

5. And the merge commit itself, which is the only commit in the repository with two parent lines.

```

$ git cat-file -p HEAD
tree ba2693f4d0086cd9a8eab7371433d6aa3acdd178
parent 1f96d1d9b440cb9f8b8d53ab39a8becb65a44eda
parent 86810156b0b336611ae572e45ad596dc623886a4
author Reader <reader@example.com> 1772772600 +0530
committer Reader <reader@example.com> 1772772600 +0530

Merge branch 'feature'

```

■ 39.9.4 PLAIN – what is really happening inside

1. Finding the merge base is a graph problem, not a guess. Git walks backwards from both tips, marking which commits are reachable from which side.
2. The merge base is a commit reachable from both tips with no descendant that is also reachable from both. `git merge-base A B` prints it.
3. Sometimes there is more than one such commit. Git's default strategy then merges the bases together first, recursively, and uses the result as the base. That is why the old strategy was called recursive.
4. Fast-forward is detected by a single question: is the current tip an ancestor of the target? `git merge-base --is-ancestor A B` answers it with an exit code.
5. If yes, there is nothing to combine, so git writes the target's tree into the index and worktree and moves the branch. No new object is created.
6. `--no-ff` forces a merge commit even when a fast-forward is possible. The reason to use it is that the merge commit is the only record that these commits were a group.

```

with --no-ff:
* a78fc04 Merge branch 'feature'
|\
| * 52ee392 C (feature)
| * 9aa4587 B (feature)
|/
* 7c451a3 A

$ git log --oneline --first-parent
a78fc04 Merge branch 'feature'
7c451a3 A

```

7. That last command is the payoff of `--no-ff`. `--first-parent` follows only the mainline, so a two-year history of merges reads as one line per feature.

8. The three-way merge itself is applied per file, comparing base, ours and theirs, then per region within the file.

■ 39.9.5 TECHNICAL - the engineer's version

1. Strategies: `ort` is the default, `recursive` is the old default, `resolve` handles a single base only, `octopus` merges more than two heads, `ours` discards the other side's content entirely, `subtree` shifts paths.
2. `ort`, short for "ostensibly recursive's twin", became the default in git 2.34, released 15 November 2021. GitHub reported it outperforming `recursive` by roughly 500 times on rename-heavy merges and by more than 9,000 times on some rebases, largely because it does not use the index as its working data structure.
3. Do not confuse the `ours` strategy with the `-X ours` option. The strategy throws away the other branch's changes. The option only decides conflicting hunks in your favour.

Thing	Effect
<code>-s ours</code>	keep our tree entirely
<code>-X ours</code>	our side wins conflicting hunks only
<code>-X theirs</code>	their side wins conflicting hunks
<code>-X ignore-all-space</code>	ignore whitespace when merging
<code>--no-ff</code>	always create a merge commit
<code>--ff-only</code>	refuse anything but a fast-forward
<code>--squash</code>	stage the result, make no merge link

4. Rename detection is on by default, controlled by `merge.renameLimit` and `diff.renames`. It is why moving a file on one side and editing it on the other usually merges correctly.
5. `git merge --abort` restores the pre-merge state using `MERGE_HEAD` and `ORIG_HEAD`. It is safe while a merge is in progress.
6. During a conflicted merge, `.git/MERGE_HEAD`, `.git/MERGE_MSG` and `.git/MERGE_MODE` exist. Their presence is how git knows a merge is unfinished and why `git status` prints "You have unmerged paths".
7. An octopus merge with more than two parents is legal and appears in the Linux kernel history. It refuses to run if any pair conflicts.

■ 39.9.6 WORDS - remember these

1. Merge base — where the two lines split — the best common ancestor of two commits in the DAG.
2. Three-way merge — compare base, ours, theirs — the algorithm that decides per region which side's change to keep.
3. Fast-forward — just move the label — an update where the old tip is an ancestor of the new one, so no merge commit is needed.
4. Merge commit — the join — a commit with two or more parents.
5. `ort` — the modern merge engine — the default merge strategy since git 2.34, November 2021.
6. First-parent history — the mainline only — `git log --first-parent`, which treats each merge as one entry.

39.10 Conflicts at the file level

■ 39.10.1 PLAIN – in simple words

1. A conflict is not git failing. It is git refusing to guess.
2. Git compares text line by line. It works on regions of lines, not on meaning.
3. If your change and their change are in different regions of the file, git takes both without asking.
4. If they overlap, git cannot know which one you want, so it writes both into the file surrounded by markers and stops.
5. The markers are ugly on purpose. They are meant to be impossible to miss and impossible to leave in by accident.
6. Your job is to edit the file until it is what you actually want, delete the markers, and tell git you are done with `git add`.
7. There is a second, nastier kind of conflict that git will never report. Two changes that merge perfectly as text, but break the program.
8. Git cannot see that, because git does not know what your code means.

■ 39.10.2 PLAIN – a picture in your head

1. Two proofreaders mark up the same page.
2. One changes a word in paragraph two. The other adds a sentence to paragraph nine. The typesetter applies both without a thought.
3. Both change the same sentence in paragraph five, differently. The typesetter stops, prints both versions one above the other with a big line between them, and puts the page in the “needs a decision” tray.
4. Now the nasty case. One proofreader renames a character from Anna to Anya throughout chapter one. The other adds a new scene in chapter nine mentioning Anna.
5. No paragraph overlaps. The typesetter merges happily. The book is now inconsistent, and nothing in the process caught it.

Where this comparison breaks:

6. A typesetter might notice the name change by reading. Git never reads. It will merge a file into nonsense with complete confidence, every time.

■ 39.10.3 PLAIN – a worked example

1. Both branches change the same line of a config file.

```

BASE           OURS (main)    THEIRS (feature)
PORT = 8080     PORT = 3000    PORT = 9090

$ git merge feature
Auto-merging config.py
CONFLICT (content): Merge conflict in config.py
Automatic merge failed; fix conflicts and then commit.

$ cat config.py
HOST = "localhost"
<<<<<<< HEAD
PORT = 3000
=====
PORT = 9090
>>>>>> feature
TIMEOUT = 30

```

2. Line by line. <<<<<<< HEAD opens the conflicted region and names your side. Everything until ===== is your version.
3. ===== is the divider. Everything after it until >>>>>>> is their version. >>>>>>> feature closes the region and names their side.
4. The unconflicted lines, HOST and TIMEOUT, are outside the markers and have already been merged.
5. The default marker style hides the base. Turning on diff3 shows it, which is much more useful.

```
$ git checkout --conflict=diff3 config.py
$ cat config.py
HOST = "localhost"
<<<<<<< ours
PORT = 3000
||||||| base
PORT = 8080
=====
PORT = 9090
>>>>>>> theirs
TIMEOUT = 30
```

6. Now you can see that the original was 8080, so both sides changed it, and neither side is a superset of the other.
7. Meanwhile, the index holds all three versions at once.

```
$ git ls-files -u
100644 5c03df4f...9181 1    config.py
100644 bd085f95...ca9b 2    config.py
100644 ee2a92c1...4f85 3    config.py
```

8. Stage 1 is the base, stage 2 is ours, stage 3 is theirs. Resolving means replacing those three entries with one stage-0 entry, which is exactly what `git add` does.

■ 39.10.4 PLAIN – what is really happening inside

1. Now the semantic conflict. This one is real, run end to end.
2. Start with two files. `lib.py` defines `get_user`. `app.py` calls it.
3. On a branch called `refactor`, rename the function to `fetch_user` and fix the one call site that exists.
4. On `main`, add a brand new file `report.py` that calls `get_user`.
5. Neither side touched a line the other touched. Watch `git`.

```
$ git merge refactor
Merge made by the 'ort' strategy.
 app.py | 4 +--
 lib.py | 2 +-
 2 files changed, 3 insertions(+), 3 deletions(-)

$ python3 -c "import report; report.report()"
File "/tmp/gw/sem/report.py", line 1, in <module>
    from lib import get_user
ImportError: cannot import name 'get_user' from 'lib'
```

6. Zero conflicts. Zero warnings. Broken program.
7. This is not a bug. Git did exactly what it promises: combine text changes that do not overlap. It never promised the result would compile.
8. The only defence against semantic conflicts is a test suite that runs on the merge result, not on either branch alone. That is precisely what continuous integration on the merged head is for.

9. Binary files are the other special case. Git cannot combine regions of a JPEG or a spreadsheet, so any change on both sides is a whole-file conflict.

```
$ git merge designer
warning: Cannot merge binary files: logo.png (HEAD vs. designer)
CONFLICT (content): Merge conflict in logo.png

$ git status --short
UU logo.png
```

10. There are no markers in the file. The working copy is left as one of the two versions. You choose a side with `git checkout --ours` or `--theirs`, then `git add`.

■ 39.10.5 TECHNICAL – the engineer’s version

1. Conflict styles, set by `merge.conflictStyle`:

Style	Shows	Since
merge	ours and theirs only	always (default)
diff3	ours, base, theirs	long-standing
zdiff3	diff3 with common lines out	git 2.35, 2022

2. `zdiff3` arrived in git 2.35, released 24 January 2022. It hoists lines common to both sides out of the conflict region, making the real disagreement smaller.
3. The resolution workflow, in full:

```
git status          # which paths are unmerged
git diff            # combined diff of conflicts only
git mergetool       # optional three-pane editor
<edit files, remove markers>
git add <path>      # collapses stages 1,2,3 into stage 0
git status          # confirm nothing is unmerged
git commit          # message is prefilled from MERGE_MSG
```

4. `git mergetool` drives an external tool named by `merge.tool`. It creates `*.BASE`, `*.LOCAL` and `*.REMOTE` temporary files from index stages 1, 2 and 3, and removes them afterwards unless `merge-tool.keepBackup` is set.
5. `git checkout --ours <path>` and `--theirs <path>` pull stage 2 or stage 3 out of the index. During a rebase these are reversed relative to intuition, because the commit being replayed is “theirs”.
6. **rerere** stands for “reuse recorded resolution”. Enable it with `git config rerere.enabled true`. Git then records the conflict text and your resolution, and replays it automatically next time the same conflict appears.

```
$ git config rerere.enabled true
$ git merge feature
Recorded preimage for 'cfg.py'
<resolve by hand, git add, git commit>
Recorded resolution for 'cfg.py'.

$ git reset --hard HEAD~1
$ git merge feature
Resolved 'cfg.py' using previous resolution.

$ cat cfg.py
PORT = 3000
```

7. The recordings live in `.git/rr-cache/<hash>/preimage` and `postimage`. The hash is computed from the normalized conflict text, so an identical conflict anywhere gets the same resolution.
8. `rerere` was written by Junio C Hamano and merged on 6 February 2006, with the commit message “git-rerere: reuse recorded resolve”. It is most valuable during long rebases, where the same conflict recurs on every replayed commit.
9. For binary formats, `.gitattributes` with `*.png binary` or a custom `merge=` driver is the practical answer. Git LFS stores a pointer file, so conflicts appear on the pointer rather than the payload, which is smaller but no easier to merge.

■ 39.10.6 WORDS – remember these

1. Conflict — git refusing to guess — overlapping changes recorded as index stages 1, 2 and 3 with markers in the worktree.
2. Conflict marker — the ugly fence — `<<<<<<, =====, >>>>>>`, plus `|||||` for the base in `diff3` style.
3. Semantic conflict — merges fine, breaks anyway — a logically incompatible pair of changes that do not overlap textually.
4. `rerere` — remember how I fixed this — recorded conflict resolutions replayed automatically.
5. Unmerged path — a file still in dispute — a path with index entries at stages other than 0.
6. Merge driver — custom combining logic for a file type — a program named in `gitattributes` via `merge=<name>`.

39.11 Rebase

■ 39.11.1 PLAIN – in simple words

1. Rebase means: take my commits, and pretend I had started from somewhere else.
2. Git works out which commits are mine, sets them aside, moves my branch to the new starting point, and applies each of my changes again, one at a time.
3. Each time it applies one, it makes a **new commit**. Not the old one moved. A new object, with a new name.
4. It must be a new object, because a commit records the name of its parent, and the parent is now different. Change any byte and the name changes.
5. So after a rebase your work looks the same but every commit has a different identifier.
6. The result is a straight line instead of a fork, which many teams prefer to read.
7. Rebase can also edit history as it replays: reorder commits, reword messages, squash several into one, or drop one entirely. That is interactive rebase.

■ 39.11.2 PLAIN – a picture in your head

1. You wrote three pages of notes on top of page 10 of a shared notebook.
2. Meanwhile the shared notebook grew to page 14.
3. Merging means gluing your three pages in as a side branch, with a note saying where they joined.
4. Rebasing means copying your three pages out by hand onto fresh sheets, placing them after page 14, and burning the originals.
5. The words on your new sheets are identical. The sheets are not the same sheets.

Where this comparison breaks:

6. Copying by hand can introduce mistakes. Rebase reapplies changes exactly, but it can hit conflicts, because the surrounding text is now different.

7. The originals are not really burned. They stay in the reflog for 90 days.

■ 39.11.3 PLAIN – a worked example

1. A branch with two commits, and main has moved on.

```
BEFORE
* a91e93d M1: main moved on          (main)
| * 9ca2c92 F2: extend feat.txt      (feature)
| * 26e79b3 F1: add feat.txt
|/
* c32f284 A: base

$ git switch feature
$ git rebase main
Successfully rebased and updated refs/heads/feature.

AFTER
* 6a7f276 F2: extend feat.txt        (feature)
* 003972b F1: add feat.txt
* a91e93d M1: main moved on          (main)
* c32f284 A: base
```

2. The hashes side by side.

Commit	Before rebase	After rebase
F1	26e79b3	003972b
F2	9ca2c92	6a7f276
tip	9ca2c92	6a7f276

3. Two commits went in, two came out, and not one identifier survived.

4. Interactive rebase on a messy branch. This is the todo list git presents.

```
$ git rebase -i HEAD~4
pick 7644785 add parser
pick bd687cc wip
pick 894e838 fix typo in parser
pick 0ec5c9b add tests
```

5. Edit it to this, and save.

```
pick    7644785 add parser
fixup   bd687cc wip
fixup   894e838 fix typo in parser
reword  0ec5c9b add tests
```

6. The result: five commits become three, the file content is byte for byte identical, and the last message was rewritten.

```
$ git log --oneline
c868f3c add unit tests for the parser
4dcdb2e add parser
292e19e base
```

■ 39.11.4 PLAIN – what is really happening inside

1. `git rebase <upstream>` runs these steps.
2. Work out the commit list: everything reachable from HEAD but not from upstream, oldest first. That is `git rev-list upstream..HEAD --reverse`.
3. Check out the upstream tip, with HEAD detached.
4. For each commit in the list, compute its change against its own parent and apply that change to the current state.
5. Create a new commit with the same message, the same author and author date, a **new committer date**, and the new parent.
6. When the list is finished, move the branch ref to the last new commit and re-attach HEAD to it.
7. Step 5 is why the hash changes: the parent line differs, the tree usually differs, and the committer timestamp differs. Chapter 38 showed that the hash is computed over exactly those bytes.
8. If applying a change fails, the rebase stops and asks you to fix it, then `git rebase --continue`. `git rebase --abort` restores everything.
9. Interactive rebase is the same loop, but the list is written to a todo file you may edit first.

Verb	What it does
pick	replay this commit unchanged
reword	replay it, open the editor for the message
edit	replay it, then stop so you can amend
squash	fold into the previous, combine both messages
fixup	fold into the previous, discard this message
drop	do not replay it at all
exec	run a shell command at this point
break	stop here, continue later

10. `git commit --fixup <sha>` writes a message starting `fixup!`. Running `git rebase -i --autosquash` then reorders that commit next to its target and marks it `fixup` for you, with no editing by hand.

■ 39.11.5 TECHNICAL – the engineer’s version

1. Two backends exist. The `apply` backend uses `format-patch` and `git am`. The `merge` backend replays with the merge machinery and is the default since git 2.26 (March 2020). `--empty`, `--keep-empty` and `--reapply-cherry-picks` control what happens to commits that become empty.
2. `git rebase --onto <newbase> <upstream> <branch>` moves only the commits that are on branch but not on upstream. It is the tool for detaching a topic built on another topic.

```
| $ git rebase --onto main topic1 topic2
```
3. `git rebase -i --autosquash` was added by Nanako Shiraishi on 6 January 2010 and shipped in git 1.7.0 on 12 February 2010. Set `rebase.autoSquash true` to make it automatic.
4. `git rebase --exec 'make test'` runs a command after each replayed commit, so you can prove every commit in the branch builds, not just the tip.
5. `rebase.autoStash true` stashes uncommitted work before the rebase and restores it afterwards.
6. History: `git-rebase-script` was added by Junio C Hamano on 25 June 2005, with the message “rebase local commits to new upstream head”. Interactive mode was added by Johannes Schindelin on 24 June 2007 and shipped in git 1.5.3 on 2 September 2007.

7. `git rebase` sets `ORIG_HEAD` to the pre-rebase tip, and writes reflog entries labelled `rebase (start)`, `rebase (pick)`, `rebase (fixup)` and `rebase (finish)`, so every intermediate state is recoverable.

■ 39.11.6 WORDS – remember these

1. Rebase — replay my work on a new base — create new commits with the same changes and different parents.
2. Upstream — the thing I am rebasing onto — the commit that becomes the new parent of the first replayed commit.
3. Interactive rebase — edit the plan before replaying — `git rebase -i` with a todo list of verbs.
4. Autosquash — fold marked fixes automatically — reorder and mark `fixup!` and `squash!` commits during `rebase -i`.
5. Committer date — when this object was made — rewritten on every replay, unlike the author date.
6. Todo list — the rebase plan — the editable file of `pick`, `squash`, `fixup` and other verbs.

39.12 The reader's rebase, explained

■ 39.12.1 PLAIN – in simple words

1. In the reader's session, a local branch was rebased onto a `main` that had moved, and the branch's SHA changed.
2. Nothing went wrong. That is the only possible outcome.
3. A commit's name is computed from its exact bytes. Those bytes include the name of its parent.
4. Rebasing gives every replayed commit a different parent. Different parent means different bytes, which means a different name.
5. So the changes are the same and the commits are not the same objects.
6. The old commits still exist. They are simply no longer pointed at by the branch.

■ 39.12.2 PLAIN – a picture in your head

1. Two identical recipes, each written on a card that also says which card comes before it in the box.
2. Move one recipe to a different position in the box and the “comes after” line must change.
3. The ingredients are identical. The card is not the same card, because the card includes that line.

Where this comparison breaks:

4. You could rub out the line on a real card. A git object's name is derived from its bytes, so editing the bytes produces a different object rather than an edited one.

■ 39.12.3 PLAIN – a worked example

1. The same commit, before and after rebase, printed in full.

```
OLD F1 (26e79b3)
tree    d18b4ea7a3728589bfef3b1836e869ac1db45faf
parent  c32f2845c02885738408ebd404b13b9f2e3c7c4e
author   Reader <reader@example.com> 1772941200 +0530
committer Reader <reader@example.com> 1772941200 +0530

NEW F1 (003972b)
tree    ed1a2fa48bd51d2a79edc8c6e3628c892e5d4c9b
parent  a91e93d9b813a53be8d8b7f51733815d0de72dea
author   Reader <reader@example.com> 1772941200 +0530
committer Reader <reader@example.com> 1772942400 +0530
```

2. Three of the four lines differ. The parent is a different commit. The tree is different because it now also contains main's new version of `core.txt`. The committer timestamp is when the replay happened.
3. Only the author line survived, which is why `git log` still shows your original date.
4. Now recompute the new name by hand, exactly as Chapter 38 did.

```
$ git cat-file commit 003972b > c.raw
$ wc -c < c.raw
219
$ { printf 'commit %d\0' 219; cat c.raw; } | shasum
003972bffe4b385f957000613aa02ea8e5208924
$ git rev-parse 003972b
003972bffe4b385f957000613aa02ea8e5208924
```

5. The hash is not stored anywhere. It is arithmetic over those 219 bytes, and anyone can repeat it.

■ 39.12.4 PLAIN - what is really happening inside

1. The change each commit introduces really is identical. Git can prove that with a patch identifier, a hash of the diff with line numbers and whitespace normalized away.

```
$ git show 26e79b3 | git patch-id --stable
3644a7626a72280f69af9830a10cf2f013748966 26e79b3d...
$ git show 003972b | git patch-id --stable
3644a7626a72280f69af9830a10cf2f013748966 003972bf...
```

2. Same patch identifier on the left. Different commit identifier on the right. That is the whole story in two lines.
3. The old commits are still in the object database, reachable through the reflog.

```
$ git reflog feature
6a7f276 feature@{0}: rebase (finish): onto a91e93d
9ca2c92 feature@{1}: commit: F2: extend feat.txt
26e79b3 feature@{2}: commit: F1: add feat.txt
c32f284 feature@{3}: branch: Created from HEAD
```

4. So the pre-rebase branch is one command away: `git reset --hard feature@{1}`.
5. Practical consequence for the reader: after that rebase, the branch on the server and the branch on the machine had no commits in common past the base. A plain push would be refused as non-fast-forward, and a forced push would be required. Section 39.13 covers when that is acceptable.

■ 39.12.5 TECHNICAL - the engineer's version

1. The invariant: SHA-1 over `commit <len>\0` plus the commit body. Any change to tree, parent, author, committer or message yields a different object ID.
2. `git patch-id --stable` produces a diff-content identity that survives rebase, cherry-pick and reordering. `git cherry` and `git rebase` use it to skip changes already present upstream.
3. `git range-diff <base>..<old> <base>..<new>` compares two versions of the same series and is the correct tool for reviewing a rebased branch. It was added in git 2.19, September 2018.
4. What survives and what does not:

Property	Survives a rebase?
author name and date	yes
commit message	yes, unless edited
the diff (patch-id)	yes

Property	Survives a rebase?
tree object ID	usually not
parent object ID	no
committer date	no
commit object ID	no
GPG signature	no, must re-sign

- Signed commits are invalidated by rebase, because the signature covers the commit bytes. `git rebase -S` re-signs each replayed commit.
- Because remote-tracking refs are updated only by network operations, a rebase makes `git status` report a large “diverged” count immediately. That is arithmetic on the graph, not a warning about anything being wrong.

■ 39.12.6 WORDS – remember these

- Patch identifier — the fingerprint of a change — a hash of the normalized diff, unchanged by rebase.
- Rewritten history — the same work, new objects — a series replaced by a replayed series with different IDs.
- Diverged — no longer a straight line — two refs whose merge base is behind both of them.
- range-diff — compare two versions of a series — `git range-diff`, added in git 2.19, September 2018.
- Re-signing — the signature does not follow — GPG signatures must be recreated after any rewrite, with `git rebase -S`.
- Reflog recovery — the pre-rebase tip is still there — `branch@{1}` names the value the branch held before the last update.

39.13 When rebasing is safe and when it destroys work

■ 39.13.1 PLAIN – in simple words

- There is one rule, and it is short. Do not rebase commits that other people already have.
- Rebasing your own branch, before anyone else has fetched it, is safe and good practice. It makes your work easy to read and easy to review.
- Once you have pushed a branch and a colleague has fetched it, their machine holds those exact commit objects.
- If you rebase and force-push, the server now has different objects with the same branch name.
- Your colleague’s git still has the originals, plus anything built on top of them. Their next pull tries to combine two versions of the same work.
- The result is duplicated commits, avoidable conflicts, and confusion about which version is real.
- Main, master, develop and release branches are shared by definition. Never rebase those.

■ 39.13.2 PLAIN – a picture in your head

- You hand out photocopies of a chapter to five reviewers.
- Then you rewrite the chapter and renumber every paragraph.
- Each reviewer’s comments now point at paragraph numbers that mean something different, or nothing at all.
- Nobody’s work is destroyed, but reconciling it is now manual and tedious for five people instead of one.

Where this comparison breaks:

- Reviewers can see that the chapter changed. Git cannot. It sees two unrelated sets of commits with the same branch name and tries to merge them, which is how duplicate commits appear in the log.

■ 39.13.3 PLAIN – a worked example

- Alice and Bob both have commit B. Bob builds C on top of it. Alice amends B, which is a one-commit rewrite, and force-pushes.

```
Alice: 2bd9139 B: with a better message    (rewritten)
        fac214d A

Bob:    4370404 C: bob's work on top of B
        426111e B                        (the original)
        fac214d A
```

- Bob fetches, and his repository now contains both versions.

```
* 2bd9139 B: with a better message    (origin/main)
| * 4370404 C: bob's work on top of B  (main)
| * 426111e B
|/
* fac214d A
```

- Bob pulls with the default merge behaviour, and gets a conflict in a file neither of them meant to fight over.

```
$ git pull
Auto-merging f
CONFLICT (content): Merge conflict in f
```

- The fix Bob needs is `git pull --rebase`, which replays only his own C onto the new B.

```
$ git pull --rebase
Successfully rebased and updated refs/heads/main.
$ git log --oneline
0b6514b C: bob's work on top of B
2bd9139 B: with a better message
fac214d A
```

- Note that Bob had to know what happened. Git gave him no message saying “the upstream branch was rewritten”.

■ 39.13.4 PLAIN – what is really happening inside

- A normal push is only allowed if the old value on the server is an ancestor of the new value. That is the fast-forward rule.

```
$ git push origin main
! [rejected]        main -> main (fetch first)
error: failed to push some refs
hint: Updates were rejected because the remote contains work
hint: that you do not have locally.
```

- `--force` switches that check off entirely. Whatever the server had is replaced, and any commit only reachable from the old value becomes garbage.

3. `--force-with-lease` replaces the check with a different one: the server's current value must equal what your remote-tracking ref says it is.
4. So `--force-with-lease` says "overwrite, but only if nobody has pushed since I last looked". Here is it refusing.

```
my origin/main says : 2bd9139
the server really has: 0b6514b

$ git push --force-with-lease origin main
! [rejected]        main -> main (stale info)
error: failed to push some refs

$ git push --force origin main
+ 0b6514b...a848939 main -> main (forced update)
```

5. The plain `--force` succeeded and Bob's commit C is now unreachable on the server. Only Bob's own clone still has it.
6. One honest warning: `--force-with-lease` compares against your remote-tracking ref, and a background `git fetch` updates that ref without you noticing. Then the lease is satisfied and the protection is gone. Use `--force-with-lease=<ref>:<expected-oid>` when it matters.

■ 39.13.5 TECHNICAL – the engineer's version

1. The decision table:

Situation	Rebase?	Then push with
local branch, never pushed	yes	plain push
pushed, nobody else uses it	yes	<code>--force-with-lease</code>
open PR, reviewers on it	prefer not	<code>--force-with-lease</code>
shared feature branch, 2+ people	no	merge instead
main, master, develop, release	never	never force
tags already published	never	never move

2. `--force-if-includes`, added in git 2.30 (December 2020), strengthens the lease by also requiring that your local history includes everything your remote-tracking ref has seen. Turn it on with `push.useForceIfIncludes true`.
3. Server-side defences: `receive.denyNonFastForwards`, and on hosted services, protected branches that reject force pushes outright.
4. Recovery when a force push destroys a colleague's commit: the commit is still in that colleague's clone, and on GitHub-style services it is often still reachable through the reflog on the server or through the events API.
5. `git rebase` on a branch that has an open pull request is common practice, and opinions differ. One camp rebases and force-pushes to keep the series clean; the other adds fixup commits during review and squashes at merge time so that review comments stay anchored. Both are defensible.
6. `git range-diff` makes the first camp's approach reviewable, because it shows what changed between the old series and the new one.

■ 39.13.6 WORDS – remember these

1. Published history — commits other people have — anything that has been pushed and fetched by someone else.

2. Fast-forward rule — you may only extend — the default push check that the old tip is an ancestor of the new one.
3. Force push — replace whatever is there — `--force`, which switches the ancestry check off.
4. Lease — only if it is still what I saw — `--force-with-lease`, which compares against your remote-tracking ref.
5. Protected branch — the server says no — a hosting-service rule rejecting force pushes and deletions.
6. Duplicate commits — the same work twice — the visible symptom of merging a rewritten branch with its original.

39.14 Merge versus rebase as a team policy

■ 39.14.1 PLAIN – in simple words

1. Both merge and rebase produce a repository containing the same final files. Neither is more correct than the other.
2. The argument is about what the history looks like afterwards, and how easy it is to work with.
3. Merging keeps a true record: this work happened on a side branch, and it was joined on this date.
4. Rebasing produces a straight line that reads like a story, but the story is tidied. It never happened quite that way.
5. Squash merging goes further: an entire branch becomes one commit on the mainline, and the individual steps are discarded.
6. Teams pick one, write it down, and enforce it with a setting on the hosting service. The worst outcome is a repository where everyone does something different.

■ 39.14.2 PLAIN – a picture in your head

1. Three ways to write up a two-week experiment.
2. Merge: the full lab notebook, every dead end included, with dates.
3. Rebase: the same experiment rewritten as a clean sequence of steps that leads to the result.
4. Squash: a one-paragraph abstract.
5. All three are honest about the result. They differ in what you can go back and inspect a year later.

Where this comparison breaks:

6. A lab notebook cannot be rewritten without fraud. Git history is a tool for communication, and tidying it is normal and legitimate.

■ 39.14.3 PLAIN – a worked example

1. The same feature, three policies, as `git log --oneline --graph`.

```

MERGE (--no-ff)
*   a78fc04 Merge branch 'feature'
| \
| * 52ee392 C (feature)
| * 9aa4587 B (feature)
| /
* 7c451a3 A

REBASE then fast-forward
* 6a7f276 C (feature)
* 003972b B (feature)
* 7c451a3 A

```

SQUASH

- * 400e7f2 Add the whole feature (squashed)
- * 040d29f A

2. One extra fact about squash merging that surprises people.

```
$ git merge-base --is-ancestor feature main && echo yes || echo no
no
```

3. After a squash merge, git does not know the branch was merged. Merging it again would try to replay B, C and D. Delete squashed branches immediately.

■ 39.14.4 PLAIN – what is really happening inside

1. The arguments for merge.
2. It records what actually happened, including when parallel work existed.
3. It never rewrites objects, so nothing anybody has fetched ever becomes invalid.
4. `git log --first-parent` collapses each feature to one line, giving a clean mainline view on demand.
5. Reverting a whole feature is one command, `git revert -m 1 <merge>`.
6. The arguments for rebase.
7. A linear history is easier to read, and `git log` needs no graph rendering.
8. Every commit on the mainline was tested against the code it actually sits on top of, which makes bisect more reliable.
9. There are no merge commits whose diff is confusing to review.
10. The arguments for squash.
11. One commit per feature is the simplest possible mainline.
12. Reviewers' intermediate "fix typo" commits never reach the main branch.
13. The cost: bisect lands on a large commit, and the intermediate steps are gone unless the branch is kept.
14. The honest summary: this is a readability and bisect-ability argument, not a correctness argument. Every option produces identical file contents. Experts disagree, and both large, successful projects and small teams run all three policies happily.

■ 39.14.5 TECHNICAL – the engineer's version

1. Comparison of the three:

Property	Merge	Rebase	Squash
rewrites objects	no	yes	yes
mainline is linear	no	yes	yes
keeps step-by-step	yes	yes	no
bisect granularity	fine	fine	coarse
revert whole feature	easy	manual	easy
force push needed	no	often	no

2. The Linux kernel merges: Linus Torvalds pulls from maintainers, producing a deeply merged history, and explicitly asks maintainers not to rebase published trees. Many web and application teams squash-merge every pull request.
3. `git config merge.ff false` on a specific branch forces merge commits. `git config pull.ff` only prevents accidental merges on pull.

4. On hosting services the equivalent settings are the allowed merge methods for pull requests: merge commit, squash and merge, rebase and merge.
5. Bisect interacts with policy. With merge-based history, `git bisect` can land on a commit inside a feature branch that was never tested alone. `git bisect --first-parent`, added in git 2.29 (October 2020), restricts the search to the mainline and avoids that.
6. Commit hygiene matters more than the policy. A branch of ten commits each of which builds and passes tests is valuable under any of the three; a branch of ten commits named “wip” is not.

■ 39.14.6 WORDS - remember these

1. Linear history — one line, no forks — a history where every commit has exactly one parent.
2. Squash merge — one commit per feature — combine a branch into a single commit with no merge link.
3. First-parent view — the mainline only — following only the first parent of each merge.
4. Bisect-ability — how precisely a search can land — a property of how large and how independently testable each commit is.
5. Merge method — the button the service offers — merge commit, squash and merge, or rebase and merge.
6. Commit hygiene — each commit is a whole, working step — the practice that makes any policy work.

39.15 Cherry-pick, revert, bisect and the rest

■ 39.15.1 PLAIN - in simple words

1. `git cherry-pick` copies one commit’s change onto your current branch. Use it to move a single fix without moving everything around it.
2. `git revert` creates a new commit that undoes an old one. Use it when the bad commit is already public.
3. `git bisect` finds which commit introduced a bug, by testing a shrinking range automatically.
4. `git blame` shows which commit last changed each line of a file.
5. `git log -S“text”` finds commits where a piece of text was added or removed.
6. `git worktree` gives you a second checked-out directory sharing one object database, so you can look at another branch without stashing.

■ 39.15.2 PLAIN - a picture in your head

1. Bisect is the game of guessing a number between 1 and 1000 with yes-or-no questions.
2. Ask “is it above 500?” and half the possibilities vanish, whatever the answer.
3. Ten questions cover a thousand numbers; twenty cover a million.
4. Git plays the same game over commits. Each test halves the range.

Where this comparison breaks:

5. The number game assumes one answer. Bisect assumes the bug appeared once and stayed. If it comes and goes, bisect will mislead you. Mark such commits `git bisect skip`.

■ 39.15.3 PLAIN – a worked example

1. Thirty-one commits. Somewhere in there, `average([1,2,4])` started returning 2.0 instead of 2.333.

```
$ ./test.sh
AssertionError: average([1,2,4]) gave 2.0
```

2. Give git a known-bad commit and a known-good commit, and a script that exits 0 for good and non-zero for bad.

```
$ git bisect start HEAD 8c84483
$ git bisect run ./check.sh
Bisecting: 14 revisions left to test (roughly 4 steps)
[e07d528] c015: routine change
running './check.sh'
Bisecting: 7 revisions left to test (roughly 3 steps)
[56821a0] c022: routine change
running './check.sh'
Bisecting: 3 revisions left to test (roughly 2 steps)
[bb1e2b9] c018: routine change
running './check.sh'
Bisecting: 0 revisions left to test (roughly 1 step)
[a83bd2e] c017: routine change
running './check.sh'
bb1e2b9ca8de5e3324381ae15dec599b8c409a9f is the first bad commit
```

3. Four tests over thirty commits, because log base 2 of 30 is about 4.9.
4. And the guilty change, which was hidden among unrelated edits.

```
- return sum(nums) / len(nums)
+ return sum(nums) // len(nums) + 0.0
```

5. Finish with `git bisect reset`, which puts you back where you started.

■ 39.15.4 PLAIN – what is really happening inside

1. Cherry-pick computes the diff of one commit against its parent, applies it here, and makes a new commit. The change is the same; the object is new.

```
original S    = 9e25370ab0a266f7b25787b2a08e719f8dd9d964
copy on main = d673ae20eb28f1b43254cad402c16571ec99e9b3

patch-id of both = c92b260713d372c3c1c2babe11f8384d5319ef45
```

2. `git cherry -v main hotfix` uses that patch identifier to say which commits on hotfix are already on main. A leading - means already there, + means not yet.
3. Revert computes the inverse diff and commits it. Reverting a merge needs `-m 1` to say which parent's line of history to keep.

```
$ git revert HEAD
error: commit 88e4ee8... is a merge but no -m option was given.

$ git revert -m 1 --no-edit HEAD
```

4. Bisect works on the graph. `git rev-list --bisect` picks the commit that splits the remaining candidates most evenly, so each answer removes about half regardless of which answer it is.
5. `git bisect run <cmd>` automates it. Exit code 0 means good, 1 to 124 and 126 to 127 mean bad, and exit code 125 means “cannot test, skip this one”.

■ 39.15.5 TECHNICAL – the engineer’s version

1. The wider toolbox:

Command	Use it when
<code>git cherry-pick -x</code>	you want a “cherry picked from”
<code>git cherry-pick -n</code>	stage without committing
<code>git revert -n</code>	revert several as one commit
<code>git bisect skip</code>	this revision cannot be tested
<code>git bisect log/replay</code>	save and rerun a bisect session
<code>git blame -C -M</code>	follow copies and moves
<code>git log -S<string></code>	find when a string appeared
<code>git log -G<regex></code>	find diffs matching a pattern
<code>git worktree add</code>	second checkout, one object store
<code>git reflog expire</code>	prune the local diary

2. History: `git rev-list --bisect` was added by Linus Torvalds on 17 June 2005, and `git-bisect-script` followed on 30 July 2005 with the commit message “Making it easier to find which change introduced a bug”.
3. `git bisect run` takes any command, so the test can be a unit test, a build, a grep on the output, or a shell one-liner. It is the highest-value five minutes in this chapter.
4. `git bisect --first-parent` (git 2.29, October 2020) confines the search to the mainline, which matters in merge-heavy repositories.
5. `git worktree` shares `.git/objects` and refs across checkouts, so a second worktree costs disk space for files only, not for history.
6. `git log -S` is a “pickaxe” search on the number of occurrences of a string; `git log -G` matches the diff text with a regular expression. They answer different questions, and `-S` is usually the one you want.

■ 39.15.6 WORDS – remember these

1. Cherry-pick — copy one change here — apply one commit’s diff and create a new commit for it.
2. Revert — undo by adding — commit the inverse of an earlier commit.
3. Bisect — binary search over history — halve the candidate range with each test until one commit remains.
4. Pickaxe — find when text appeared — `git log -S<string>`.
5. Worktree — a second checkout of one repository — an extra working directory sharing the same object database.
6. Skip — this one cannot be judged — `git bisect skip`, or exit code 125 from a `bisect run` script.

39.98 Common wrong ideas

1. Wrong: `git add` only marks a file for later. Right: it hashes the content and writes a real object into `.git/objects` immediately.
2. Wrong: the index stores differences. Right: it stores a complete snapshot, one entry per tracked file, with stat data and object IDs.
3. Wrong: `git commit` copies your files. Right: the blobs already exist; commit writes trees, one commit object, and moves a ref.
4. Wrong: a push failure means nothing was written. Right: a disconnect while reading the response says nothing about the write. Re-query with `git ls-remote`.

5. Wrong: `git status` shows the server's current state. Right: it compares against a cached remote-tracking ref that only a fetch or push updates.
6. Wrong: `git fetch` can mess up my work. Right: fetch changes only refs/remotes/* and the object database. `git pull` is the one to be careful with.
7. Wrong: rebase moves my commits. Right: it creates new commit objects with new identifiers, and leaves the old ones in the reflog.
8. Wrong: a clean merge means the code works. Right: `git` merges text, not meaning. Two non-overlapping changes can break the program completely.
9. Wrong: `git reset --hard` loses commits forever. Right: commits survive in the reflog for 90 days. Uncommitted edits, however, are gone.
10. Wrong: `--force-with-lease` is always safe. Right: it compares against your remote-tracking ref, which a background fetch can silently refresh.

39.99 Chapter summary in 20 lines

1. A tracked file exists in three places: working directory, index, object database. HEAD says which commit you are on.
2. `git status` is nothing more than two comparisons: index against HEAD, and working directory against index.
3. The index is a real binary file, `.git/index`, starting with the four bytes DIRC, present in `git`'s first commit on 7 April 2005.
4. It holds one entry per tracked file: path, object ID, mode, and cached stat data from the filesystem.
5. That stat cache is why `git status` is fast: most files are never opened, only asked about.
6. `git add` hashes the content and writes the object at once, then records its name in the index. `git add -p` can stage a version that exists nowhere.
7. `git commit` writes trees from the index, writes one commit object, and moves one ref. Three steps, in that order.
8. The index exists so you can choose what goes in a commit. Other systems manage without one; it is a design choice, not a necessity.
9. `git restore`, `git reset --soft/--mixed/--hard`, `git revert`, `git clean` and `git stash` differ only in which of the three places they touch.
10. Only `git reset --hard` and `git clean` can destroy work that `git` never stored. Everything else is recoverable through the reflog.
11. A push sends one packfile of only the missing objects, framed in pkt-lines, with three sideband channels: data, progress, fatal error.
12. Twenty commits is one round trip, not twenty. In the real measurement, sixty objects in 3.95 KiB.
13. `send-pack`: unexpected disconnect while reading sideband packet means the pack was sent and the reply never arrived.
14. That error is identical whether the server applied the write or not. Both cases were reproduced. Only `git ls-remote` can tell you which happened.
15. In the reader's session both pushes had not applied, and that was established by re-querying, not by reading the error.
16. `git fetch` changes only remote-tracking refs and the object store. `git pull` adds an integration step that touches your files.
17. Merge finds the common ancestor and does a three-way merge; fast-forward needs no commit because the old tip is already an ancestor.
18. Conflicts are `git` refusing to guess about overlapping text. Semantic conflicts merge cleanly and still break the code.

19. Rebase replays each commit onto a new parent, so every commit gets a new identifier even though the patch identifier is unchanged.
20. Rebase your own unpushed work freely; never rewrite history other people already hold, and prefer `--force-with-lease` when you must force.

Your Own Git Server, and What GitHub Adds

40.0 What this chapter gives you

1. You will be able to say, in one sentence, what a git server is, and prove it by building one in an empty folder in under a second.
2. You will be able to explain what `--bare` means, what a bare repository contains, what it deliberately lacks, and why pushing into a normal repository's checked-out branch is refused by default.
3. You will be able to run the whole loop yourself: create a server repository, push to it, clone it somewhere else, commit there, push back, and pull the change into the first copy, with real output at every step.
4. You will be able to do the same thing over SSH and see that a remote URL is nothing but a path plus a way of getting to it.
5. You will be able to read and write a refspec, and say exactly what `+refs/heads/*:refs/remotes/origin/*` does, character by character.
6. You will be able to write a server-side hook that deploys a website on every push, and say why a client-side hook is a helper and never a rule.
7. You will be able to state precisely what GitHub is: hosting for bare repositories, plus a web interface, plus identity and permissions, plus automation, founded in 2008 and bought by Microsoft in 2018 for 7.5 billion United States dollars.
8. You will be able to sort every feature you use into "this is git" or "this is GitHub", and defend each answer.
9. You will be able to explain the reader's own outage exactly: why git kept working while the GitHub API did not, using a three-column table of operations.
10. You will be able to choose between GitHub, a self-hosted Forgejo or GitLab, and a plain SSH remote with no web interface at all, and say what you gain and what you take on.
11. You will be able to mirror, migrate and back up a repository properly, and say what a backup of git does not cover.

40.1 A git server is just another repository

■ 40.1.1 PLAIN – in simple words

1. Here is the whole idea of this chapter in one line.
2. A git "server" is just another git repository that other machines can reach.
3. That is all. There is no server program that must be bought. There is no special file format. There is no central authority.
4. Every copy of a git repository holds the full history. Every copy can send history to another copy and receive history from another copy.

5. So a “server” is only a copy that everybody agrees to treat as the meeting point.
6. We call it the server because of where it sits and who can reach it, not because of what it is made of.
7. The one thing a meeting-point copy usually does differently is that it has no working files. Only the history. We will see why in the next section.
8. Everything else in this chapter is proof of that single claim.

■ 40.1.2 PLAIN – a picture in your head

1. Think of a study group of four students who each keep a notebook of the same lecture course.
2. Each notebook is complete. Any student can copy pages from any other student.
3. The group decides to leave one spare notebook in the department office, because the office is open to all four and nobody has to chase anybody.
4. The office notebook is not magic. It is the same kind of notebook. It has the same pages. It is special only because everyone can walk to it.
5. If the office burns down, the course is not lost. Four complete notebooks still exist in four bags.
6. That is exactly the relationship between your laptop’s repository and the one you call the server.

Where this comparison breaks:

7. Copying pages by hand is slow and error-prone. Git copies exactly, and it copies only the pages the other side is missing.
8. Two students writing on the same page at the same time makes a mess. Git refuses that case rather than making a mess: it rejects the push and makes you combine the work first.
9. And a notebook has one order of pages. Git history is a graph, so two students can genuinely have two different valid orders at once.

■ 40.1.3 PLAIN – a worked example

1. We will use three folders on one machine for the whole chapter. One machine is enough to prove the point, because git does not know or care whether the other repository is across the room or across the world.
2. The layout is this.

/tmp/lab/proj	your working copy, folder number one
/tmp/lab/srv/proj.git	the "server", a repository with no files
/tmp/lab/copy	a second working copy, folder number three

3. Here is a real repository being created and given two commits. Every block in this chapter is real output from a real run, not an illustration.

```
$ git init proj
Initialized empty Git repository in /tmp/lab/proj/.git/

$ git add README.md && git commit -m "First commit: add README"
[main (root-commit) 041c4ad] First commit: add README
1 file changed, 1 insertion(+)
create mode 100644 README.md

$ git add app.py && git commit -q -m 'Add app.py' && git log --oneline
08391eb Add app.py
041c4ad First commit: add README
```

4. Two commits exist. Nothing has touched a network. Nothing has touched a server, because there is not one yet.

5. In the next section we will make the server, and it will take one command and about ten milliseconds.

■ 40.1.4 PLAIN – what is really happening inside

1. When two git repositories talk, one of them runs a small program that reads history out, and the other runs a small program that writes history in.
2. The reading program is called `git-upload-pack`. It says “here are all the names I have and what they point to”, and then sends whatever objects the other side asks for.
3. The writing program is called `git-receive-pack`. It accepts a bundle of objects and a list of name changes, and applies them.
4. When you type `git fetch`, your git starts `git-upload-pack` on the other side and reads from it.
5. When you type `git push`, your git starts `git-receive-pack` on the other side and writes to it.
6. That is the entire client and server story. Two small programs and a way to connect their inputs and outputs.
7. If the other repository is a folder on the same disk, git simply runs the program directly. No network at all.
8. If it is on another machine, git needs some way to run that program over there and pipe bytes back and forth. That is what SSH does. That is what HTTPS does, with a small wrapper. That is what the git protocol does.
9. So the phrase “git server” really means “a machine willing to run `git-upload-pack` and `git-receive-pack` on a repository for you”.

■ 40.1.5 TECHNICAL – the engineer’s version

1. Git’s wire protocol is a framed byte stream called **pkt-line**. Each frame begins with four hexadecimal digits giving the frame’s total length, including those four digits. The value `0000` is a flush packet and marks the end of a section.
2. We can see the raw advertisement by running the server program by hand over SSH. This is the real output, trimmed on the right to fit the page.

```
$ ssh gitbox 'git upload-pack --advertise-refs \
/tmp/lab/srv/proj.git' | cat -A | head -3
010b64952410dab49907fec630138e55e7504834b082 HEAD^@multi_ack
thin-pack side-band side-band-64k ofs-delta shallow ...
symref=HEAD:refs/heads/main object-format=sha1
agent=git/2.43.0$
003d64952410dab49907fec630138e55e7504834b082 refs/heads/main$
0000
```

3. Read the first line. `010b` is hexadecimal for 267, the length of that frame. Then a 40-character SHA-1, a space, the ref name `HEAD`, a NUL byte (shown by `cat -A` as `^@`), and then the capability list.
4. The second frame is `003d`, hexadecimal for 61: four length digits, 40 hash characters, one space, `refs/heads/main` at 15 characters, and one newline. That is $4 + 40 + 1 + 15 + 1 = 61$. The framing is exact.
5. `0000` ends the advertisement.
6. Protocol versions: version 0 is the original, version 1 adds a version string, and **protocol v2** was introduced in Git 2.18 (June 2018) and made the default for fetch in Git 2.26 (March 2020). Version 2 lets the client ask for a filtered ref list instead of receiving every ref up front, which matters on repositories with tens of thousands of refs.
7. The four transports and how the server program is started:

Transport	How the program starts	Default port
local path	run directly by your git	none
ssh	remote shell command	22
https	git-http-backend via CGI	443
git	git-daemon	9418

- Port 9418 is registered with IANA for the git protocol. The git protocol has no authentication and no encryption, which is why GitHub permanently disabled unencrypted `git://` access on 15 March 2022.
- Commands that observe this layer: `git ls-remote`, `GIT_TRACE=1 git fetch`, `GIT_TRACE_PACKET=1 git fetch`, and `git daemon --verbose`.

■ 40.1.6 WORDS – remember these

- Repository — a project’s full history plus its settings — a directory containing an object database, a ref namespace and a configuration file.
- Server — the copy everybody can reach — any repository exposed through a transport that can start upload-pack or receive-pack.
- upload-pack — the program that sends history out — the server side of fetch and clone, which advertises refs and streams a packfile.
- receive-pack — the program that takes history in — the server side of push, which validates ref updates and runs the receive hooks.
- pkt-line — git’s way of cutting a stream into labelled chunks — a framing format with a four-hex-digit length prefix and 0000 as flush.
- Transport — the road the bytes travel on — the mechanism (file, ssh, http, git) that connects the two programs’ standard input and output.

40.2 Bare repositories

■ 40.2.1 PLAIN – in simple words

- A normal repository has two parts: the files you edit, and a hidden folder called `.git` that holds the history.
- A **bare repository** is the second part on its own. The history, with no files to edit.
- You make one with `git init --bare`. By convention its folder name ends in `.git`, for example `proj.git`.
- A bare repository is what you use for a server, because a server has nobody sitting at it editing files.
- It is missing two things on purpose: a working tree (the actual editable files) and an index (git’s staging list of what will go into the next commit).
- Those two things only make sense when a human is working. A server has no human working in it.
- You cannot run `git status` usefully in a bare repository, because there is no working tree to compare against.
- You can run `git log`, `git show`, `git cat-file` and everything else that only reads history, because all of the history is right there.

■ 40.2.2 PLAIN – a picture in your head

1. Think of a normal repository as a desk. On the desk are the pages you are writing. In a drawer under the desk is the complete archive of every version ever saved.
2. A bare repository is the drawer, on its own, in a storeroom. No desk.
3. You can pull any past version out of the drawer. You just cannot scribble on anything, because there is no desk surface.
4. When somebody pushes, they are handing new folders into the drawer and updating the labels on the front.

Where this comparison breaks:

5. A drawer takes more space as you add papers. Git's drawer stores each distinct version once and stores similar versions as small differences, so it grows far more slowly than the total of all versions.
6. And a real drawer does not check what you put in it. Git verifies every object against its hash on the way in and on the way out.

■ 40.2.3 PLAIN – a worked example

1. Here are the two kinds side by side, from a real run.

```
$ git init --bare srv/proj.git
Initialized empty Git repository in /tmp/lab/srv/proj.git/

$ ls -l /tmp/lab/proj/.git      $ ls -l /tmp/lab/srv/proj.git
COMMIT_EDITMSG                HEAD
HEAD                          branches
branches                      config
config                        description
description                   hooks
hooks                         info
index                         objects
info                          refs
logs
objects
```

2. Look at what matches. HEAD, branches, config, description, hooks, info, objects, refs are in both. That is the whole history machinery.
3. Look at what only the normal repository has: index, logs and COMMIT_EDITMSG. Those are all about a person doing work here.
4. And the normal repository has one more thing that the bare one does not: the actual files, one level up.

```
$ ls -l /tmp/lab/proj
README.md
app.py
```

5. Git records the difference in one configuration setting.

```
$ git -C /tmp/lab/proj config --get core.bare
false

$ git -C /tmp/lab/srv/proj.git config --get core.bare
true
```



```
$ cat /tmp/lab/srv/proj.git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = true
```

6. That is the entire difference at the configuration level. One boolean.

■ 40.2.4 PLAIN – what is really happening inside

1. Now the important part: why you cannot push into a normal repository's checked-out branch.
2. A checked-out branch means three things are supposed to agree: the branch pointer, the index, and the files on disk.
3. If somebody pushes new commits into that branch from outside, git would move the branch pointer but would not touch the index or the files.
4. The person sitting at that repository would then be told they have deleted everything that arrived, because their files no longer match their branch.
5. So git refuses. Here is the real refusal, with the long lines wrapped to fit this page and nothing else changed.

```
$ git push wt main
remote: error: refusing to update checked out branch:
refs/heads/main
remote: error: By default, updating the current branch in a
non-bare repository is denied, because it will make the
index and work tree inconsistent with what you pushed, and
will require 'git reset --hard' to match the work tree to
HEAD.
remote:
remote: You can set the 'receive.denyCurrentBranch'
configuration variable to 'ignore' or 'warn' in the remote
repository to allow pushing into its current branch;
however, this is not recommended unless you arranged to
update its work tree to match what you pushed in some
other way.
To /tmp/lab/wt
! [remote rejected] main -> main (branch is currently
checked out)
error: failed to push some refs to '/tmp/lab/wt'
```

6. Read the message closely. It does not say “you may not do this”. It says “this would make the index and work tree inconsistent”.
7. A bare repository has no index and no work tree, so there is nothing to make inconsistent, so the refusal never applies.
8. That is the real reason servers use bare repositories. Not policy. Arithmetic.

■ 40.2.5 TECHNICAL – the engineer's version

1. `git init --bare` sets `core.bare = true` and lays the repository contents directly in the given directory rather than in a `.git` subdirectory.
2. Contents of a bare repository, and what each part is:

Entry	What it holds
HEAD	symbolic ref, usually <code>refs/heads/main</code>
objects/	the object database

Entry	What it holds
refs/	loose refs, heads and tags
packed-refs	refs collapsed into one file
config	repository configuration
hooks/	executable hook scripts
info/	exclude file, alternates, http info

- Absent from a bare repository: index (the staging area, a binary file in the format documented as the git index format, version 2, 3 or 4), the working tree, logs/ unless `core.logAllRefUpdates` is turned on, and `COMMIT_EDITMSG`.
- The refusal above comes from `receive.denyCurrentBranch`, whose default is `refuse`. Related settings are `receive.denyDeleteCurrent`, default `refuse`, and `receive.denyNonFastForwards`, default `false`.
- Three ways to make a bare repository, all equivalent in result:


```
git init --bare proj.git
git clone --bare /path/to/proj proj.git
git clone --mirror /path/to/proj proj.git
```
- They differ in what refs come along. `--bare` from `init` starts empty. `clone --bare` copies branches and tags but sets no `fetch refspec`. `clone --mirror` copies every ref under `refs/*` and configures the repository to keep tracking them. We measure that difference in 40.10.
- There is a fourth arrangement worth knowing: a non-bare repository with `receive.denyCurrentBranch = updateInstead`. Then a push into the checked out branch also updates the working tree, provided the working tree is clean. This is useful for a small deploy target and is handled by the `push-to-checkout` hook.
- Observation commands: `git rev-parse --is-bare-repository`, `git rev-parse --git-dir`, `git count-objects -vH`.

■ 40.2.6 WORDS – remember these

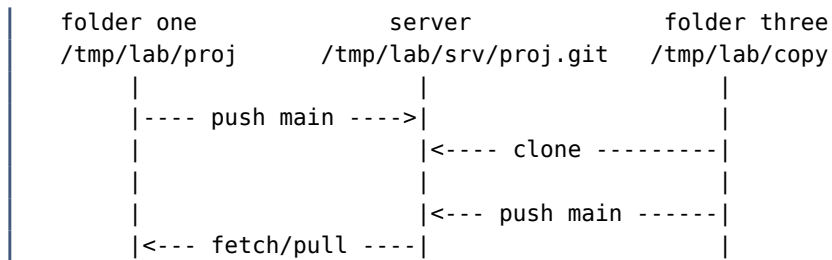
- Bare repository — the history with no editable files — a repository with `core.bare = true`, containing the object database and refs at top level, with no working tree and no index.
- Working tree — the files you actually edit — the checked-out state of one commit, materialized on the filesystem.
- Index — git's list of what goes in the next commit — a binary file mapping paths to blob hashes and stat data, also used during merges.
- Checked-out branch — the branch you are currently on — the branch named by the symbolic ref `HEAD`.
- `denyCurrentBranch` — the rule that stops pushes into a busy branch — the `receive.denyCurrentBranch` configuration key, default `refuse`.

40.3 Doing it end to end

■ 40.3.1 PLAIN – in simple words

- We will now do the whole loop, with nothing hidden.
- Make a server repository. Tell your project about it. Send your history.
- Copy the server repository into a fresh folder as if you were a second person on a second machine.
- Make a commit in that fresh folder. Send it to the server.

5. Go back to the first folder. Bring the change down. Now both folders and the server agree.
6. Then we do the same thing again over SSH, to show that the only thing that changed is the address.
7. Here is the shape of the whole thing.



■ 40.3.2 PLAIN – a picture in your head

1. Think of a shared drop box in a corridor between two rooms.
2. You put a copy of your work in the box. The person in the other room takes a copy out.
3. They add a page and put their copy back in. You take the new page.
4. Nobody ever has to be in the corridor at the same time. The box holds the agreed state.
5. Git's box is smarter than a real one, because it never loses the older versions when a new one arrives. It stacks them all, in order, with each one pointing at the one before.

Where this comparison breaks:

6. A drop box has no opinion. Git's box refuses anything that would erase history it already holds, unless you force it explicitly.
7. And a drop box holds one thing at a time. Git's box holds many named lines of work at once, one name per branch.

■ 40.3.3 PLAIN – a worked example

1. Step one. Point the project at the server and push.

```

$ git remote add origin /tmp/lab/srv/proj.git

$ git remote -v
origin /tmp/lab/srv/proj.git (fetch)
origin /tmp/lab/srv/proj.git (push)

$ git push -u origin main
To /tmp/lab/srv/proj.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
  
```

2. The server now really has the history. Look inside it directly.

```

$ ls -l /tmp/lab/srv/proj.git/refs/heads
main

$ cat /tmp/lab/srv/proj.git/refs/heads/main
fa776d1882d36bdc0af68926f71ff710cdea281a

$ git -C /tmp/lab/srv/proj.git log --oneline
fa776d1 Change app.py
08391eb Add app.py
041c4ad First commit: add README
  
```

3. A branch on the server is one file containing one 40-character hash. That is all a branch has ever been.
4. Step two. Clone it into a third folder, as a second person would.

```
$ git clone /tmp/lab/srv/proj.git /tmp/lab/copy
Cloning into '/tmp/lab/copy'...
done.

$ git -C /tmp/lab/copy log --oneline
fa776d1 Change app.py
08391eb Add app.py
041c4ad First commit: add README

$ git -C /tmp/lab/copy branch -a
* main
remotes/origin/HEAD -> origin/main
remotes/origin/main
```

5. Step three. Commit in the copy and push it back. This is the terminal view, where the percentage counters overwrite themselves in place.

```
$ git add NOTES.md && git commit -q -m 'Add NOTES.md'

$ git push --progress origin main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 2 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 328 bytes | 328.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To /tmp/lab/srv/proj.git
fa776d1..249b3ed main -> main
```

6. Three objects and 328 bytes went across: one commit, one tree, one blob. Nothing else was sent, because the server already had everything else.
7. Step four. Back in folder one, fetch first so you can see the two halves of a pull separately.

```
$ git status -sb
## main...origin/main

$ git fetch origin
From /tmp/lab/srv/proj
fa776d1..249b3ed main -> origin/main

$ git status -sb
## main...origin/main [behind 1]

$ ls
README.md
app.py
```

8. Read that carefully. After fetch, git knows about the new commit and says you are behind by one. But ls shows no new file. Fetch changed nothing in your working tree. It only downloaded history.
9. Now pull, which is fetch plus merge.

```
$ git pull
Updating fa776d1..249b3ed
```

```
Fast-forward
NOTES.md | 1 +
1 file changed, 1 insertion(+)
create mode 100644 NOTES.md

$ ls
NOTES.md
README.md
app.py
```

10. The loop is closed. Three repositories, one history, no service, no account, no web page.

■ 40.3.4 PLAIN - what is really happening inside

1. Now the same thing over SSH, to prove the point of this chapter.
2. A remote URL has two parts: how to get there, and where it is once you arrive. Change the first part and nothing else changes.

```
$ git remote add ssh-origin \
    ssh://root@127.0.0.1:2222/tmp/lab/srv/proj.git

$ git ls-remote ssh-origin
249b3edc818ac4a38fc39b982e19a880675326de    HEAD
249b3edc818ac4a38fc39b982e19a880675326de    refs/heads/main
```

3. Exactly the same repository, reached a different way. Now push to it.

```
$ git commit -q -am 'Document how to run app'

$ git push ssh-origin main
To ssh://127.0.0.1:2222/tmp/lab/srv/proj.git
    249b3ed..6495241  main -> main

$ git -C /tmp/lab/srv/proj.git log --oneline -1
6495241 Document how to run app
```

4. And clone over SSH using the short form, which needs an entry in your SSH configuration file because the short form cannot carry a port number.

```
$ cat /root/.ssh/config
Host gitbox
    HostName 127.0.0.1
    Port 2222
    User root
    IdentityFile /root/.ssh/id_ed25519

$ git clone gitbox:/tmp/lab/srv/proj.git /tmp/lab/viassh
Cloning into '/tmp/lab/viassh'...

$ git -C /tmp/lab/viassh log --oneline -2
6495241 Document how to run app
249b3ed Add NOTES.md
```

5. What SSH actually did was log in and run one command. Here is git's own verbose trace of that decision, reproduced by running the command by hand.

```
$ ssh gitbox 'git upload-pack --advertise-refs \
    /tmp/lab/srv/proj.git' | head -c 60
010b64952410dab49907fec630138e55e7504834b082 HEAD
```

6. So SSH is not doing anything clever. It is a remote login that runs `git-upload-pack` or `git-receive-pack` and connects the pipes.
7. That is why an SSH remote needs no git-specific server software at all. If the account can log in and git is installed, you have a git server.

■ 40.3.5 TECHNICAL – the engineer’s version

1. The git protocol also works, which shows the third transport. Here is a real git daemon serving the same bare repository.

```
$ git daemon --reuseaddr --base-path=/tmp/lab/srv \
  --export-all --verbose

$ git ls-remote git://127.0.0.1/proj.git
a2648f3588da557802930bc984745c8a574b7a38      HEAD
8a65572b3a58cded170998be063e26b8cbea76e6      refs/heads/demo
0327736b4327edde443c3dfa9456555015d0b3f4      refs/heads/feature/login
a2648f3588da557802930bc984745c8a574b7a38      refs/heads/main
c541c6934d9bd45d02e86d902253422366134e87      refs/notes/commits
9ee3b0bda37deeffc6317b5dcda3c483895f490c      refs/tags/v1.0
```

2. The daemon log confirms protocol v2 and which program was requested.

```
[17254] Ready to rumble
[17260] Connection from 127.0.0.1:42896
[17260] Extended attribute "host": 127.0.0.1
[17260] Extended attribute "protocol": version=2
[17260] Request upload-pack for '/proj.git'
[17254] [17260] Disconnected
```

3. And the git protocol is read-only unless you go out of your way, because it has no authentication:

```
$ git push origin main
fatal: remote error: access denied or repository not exported:
/proj.git
```

4. Timings and sizes from this run, on a small repository:

Operation	Objects	Bytes on wire
push of one commit	3	328
bare repo, loose	41	164.00 KiB
bundle of all refs	all	3.1 KiB

5. Server-side authorization for SSH remotes is normally done by the account, not by git. The standard pattern on a self-hosted box is a single git user whose `~/.ssh/authorized_keys` has one line per developer key, each line prefixed with `command="git-shell -c \"$SSH_ORIGINAL_COMMAND\""` plus `no-port-forwarding,no-agent-forwarding,no-pty`.
6. `git-shell` is a restricted login shell shipped with git that only permits `git-upload-pack`, `git-upload-archive` and `git-receive-pack`. It has existed since the early git releases and is the simplest real access control you can deploy.
7. Useful environment variables for debugging transports: `GIT_TRACE=1`, `GIT_TRACE_PACKET=1`, `GIT_SSH_COMMAND`, `GIT_CURL_VERBOSE=1`.

■ 40.3.6 WORDS – remember these

1. Remote — a saved nickname for another repository — a named set of configuration keys holding one or more URLs and one or more refsspecs.
2. origin — the usual nickname for where you cloned from — a convention, set automatically by `git clone`, changeable with `git clone -o name`.
3. Fetch — download history without touching your files — updating remote-tracking refs and the object database only.
4. Pull — fetch and then combine — `git fetch` followed by `git merge` or `git rebase`, depending on `pull.rebase`.
5. Fast-forward — moving a branch pointer along a straight line — an update where the old commit is an ancestor of the new one, so no merge is needed.
6. git-shell — a login shell that only allows git — a restricted shell that rejects every command except the three git transport programs.

40.4 Remotes in depth

■ 40.4.1 PLAIN – in simple words

1. A remote is a saved nickname for another repository.
2. Instead of typing a long address every time, you say `origin` and git looks up the address.
3. You can have as many remotes as you like. They are just entries in a text file inside your repository.
4. The address can take four shapes: a plain folder path, an SSH address, an HTTPS address, or a `git://` address.
5. You can add a remote, rename it, change its address and delete it, and none of that touches your history at all. It is only a phone book entry.
6. A remote also carries a rule that says which of the other side's names get copied into which of your names. That rule is called a **refspec**.
7. The refspec is the part that most people never read, and it is the part that explains almost every confusing thing about push and fetch.

■ 40.4.2 PLAIN – a picture in your head

1. Think of the contacts list in a phone.
2. `origin` is a contact name. Behind it sits a number.
3. Changing the number does not change any of your past conversations. It only changes where the next call goes.
4. Deleting the contact does not delete the conversations either.
5. Now imagine each contact also has a filing rule attached: “anything this person sends me goes in the folder named after them”. That is the refspec.
6. So messages from Anita land in a folder called Anita, and messages from Bala land in a folder called Bala, and they never overwrite each other even if both send a file with the same name.

Where this comparison breaks:

7. A phone contact has one number. A git remote can have one address for reading and several different addresses for writing.
8. And a filing rule in a phone is your private business. A refspec is negotiated with the other side, because the other side must agree that the names you ask for exist.

■ 40.4.3 PLAIN – a worked example

1. Everything about a remote lives in the repository's configuration file. Here is the real file after adding two remotes.

```
$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = /tmp/lab/srv/proj.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
    remote = origin
    merge = refs/heads/main
[remote "ssh-origin"]
    url = ssh://root@127.0.0.1:2222/tmp/lab/srv/proj.git
    fetch = +refs/heads/*:refs/remotes/ssh-origin/*
```

2. Now the refspec, character by character. The value is:

```
| +refs/heads/*:refs/remotes/origin/*
```

3. It splits at the colon into a source and a destination.

Part	Meaning
+	allow a non-fast-forward update
refs/heads/*	source: every branch over there
:	separator
refs/remotes/origin/*	destination: my copy of them

4. In plain words: “take every branch on the other side, and store it under a name of mine that begins with refs/remotes/origin/, and let it move backwards or sideways if it has to.”
5. So the other side's main becomes your refs/remotes/origin/main, which you normally type as origin/main.
6. This is why origin/main is not a branch you can commit on. It is a read-only copy of a name that lives somewhere else.
7. Watch the destination folder appear on disk after fetching from two different remotes.

```
$ ls -R .git/refs
.git/refs:
heads
remotes
tags

.git/refs/heads:
main

.git/refs/remotes:
origin
ssh-origin

.git/refs/remotes/origin:
main
```

```
.git/refs/remotes/ssh-origin:
main
```

- Now push with an explicit refspec, which lets you send a local branch to a differently named remote branch.

```
$ git push origin HEAD:refs/heads/wip-login
To /tmp/lab/srv/proj.git
 * [new branch]      HEAD -> wip-login

$ git ls-remote origin
64952410dab49907fec630138e55e7504834b082    HEAD
64952410dab49907fec630138e55e7504834b082    refs/heads/main
0327736b4327edde443c3dfa9456555015d0b3f4    refs/heads/wip-login
```

- And an empty source means “delete the destination”. That is the whole trick behind deleting a remote branch.

```
$ git push origin :refs/heads/wip-login
To /tmp/lab/srv/proj.git
 - [deleted]         wip-login

$ git ls-remote origin
64952410dab49907fec630138e55e7504834b082    HEAD
64952410dab49907fec630138e55e7504834b082    refs/heads/main
```

- `git push origin --delete wip-login` is a friendlier spelling of exactly the same refspec.

■ 40.4.4 PLAIN - what is really happening inside

- The leading `+` deserves its own demonstration, because it is the difference between “I refuse to lose history” and “overwrite it”.
- We put a commit on a branch, copy it with a refspec that has no `+`, then rewrite the commit on the server, then fetch again.

```
$ git fetch -q origin \
    '+refs/heads/demo:refs/remotes/nf/demo'
$ git rev-parse --short refs/remotes/nf/demo
4f4b9de
```

- Now the branch is rewritten on the server, with `git commit --amend` followed by a force push. The old commit `4f4b9de` becomes `8a65572`, and the two are not ancestors of each other.

```
$ git push --force origin demo
To /tmp/lab/srv/proj.git
 + 4f4b9de...8a65572 demo -> demo (forced update)
```

- Fetch with a refspec that has no `+`, and git refuses. Fetch with `+`, and git obeys. The same command, run twice, differing by one character.

```
$ git fetch origin refs/heads/demo:refs/remotes/nf/demo
From /tmp/lab/srv/proj
 ! [rejected]         demo -> nf/demo (non-fast-forward)
 + 4f4b9de...8a65572 demo -> origin/demo (forced update)

$ git fetch origin '+refs/heads/demo:refs/remotes/nf/demo'
From /tmp/lab/srv/proj
 + 4f4b9de...8a65572 demo -> nf/demo (forced update)
```

```
$ git rev-parse --short refs/remotes/nf/demo
8a65572
```

5. Notice something in the middle of the first result. `nf/demo` was rejected, and in the very same command `origin/demo` was force-updated.
6. That is because the default refspec for `origin` has the `+`, and the one we typed by hand did not. Two rules, one fetch, two different outcomes.
7. That single output block explains the whole meaning of the plus sign better than any paragraph could.

■ 40.4.5 TECHNICAL - the engineer's version

1. URL forms, all four, with real examples from this chapter's run:

Form	Example
local path	<code>/tmp/lab/srv/proj.git</code>
ssh, full	<code>ssh://root@host:2222/tmp/proj.git</code>
ssh, short	<code>gitbox:/tmp/lab/srv/proj.git</code>
https	<code>https://github.com/git/git</code>
git	<code>git://127.0.0.1/proj.git</code>

2. The short SSH form, `user@host:path`, is called the scp-like syntax. It cannot express a port number. If you need a non-default port you must use the `ssh://` form or a `Host` block in `~/.ssh/config`. That is a real limitation, not a style preference.
3. In the `ssh://` form the path is absolute. In the scp-like form the path is relative to the login user's home directory unless it starts with a slash. This trips people up constantly.
4. Remote management commands:

```
git remote add <name> <url>
git remote rename <old> <new>
git remote remove <name>
git remote set-url <name> <newurl>
git remote set-url --push <name> <pushurl>
git remote set-url --add --push <name> <another>
git remote show <name>
git remote prune <name>
```

5. Different fetch and push URLs are genuinely useful. Here we set one remote that reads from one place and writes to two, so a single push updates both a primary and a backup.

```
$ git remote -v
both    /tmp/lab/srv/proj.git (fetch)
both    /tmp/lab/srv/proj.git (push)
both    /tmp/lab/srv/mirror.git (push)

$ git push both main
Everything up-to-date
To /tmp/lab/srv/mirror.git
* [new branch]      main -> main
```

6. Common reasons to have several remotes:

Situation	Typical remotes
you forked a project	origin, upstream
migrating platforms	origin, newhost
offsite backup	origin, backup
internal plus public	origin, public

7. Refspec grammar, stated properly. A refspec is `[+]<src>[:<dst>]`. For fetch, `src` is a ref on the remote and `dst` is a ref in your repository. For push, `src` is a local ref or any expression that resolves to a commit, and `dst` is a ref on the remote. The `+` sets the force flag for that one mapping. An empty `src` on push means delete `dst`. An omitted `dst` on fetch means “download it but only record it in `FETCH_HEAD`”.
8. A remote may carry several fetch refs. Adding one is how you fetch GitHub pull-request refs:

```
git config --add remote.origin.fetch \
    '+refs/pull/*/head:refs/remotes/origin/pr/*'
```

9. URL rewriting is a useful production trick. `insteadOf` rewrites a prefix for all commands:

```
$ git config --global \
    url.'ssh://root@127.0.0.1:2222/tmp/lab/srv/'.insteadOf 'kb:'

$ git ls-remote kb:proj.git | head -1
a2648f3588da557802930bc984745c8a574b7a38    HEAD
```

10. `pushInsteadOf` does the same but only for pushes, which is the standard way to clone over HTTPS in continuous integration while pushing over SSH from a developer machine using the same recorded URL.

■ 40.4.6 WORDS – remember these

1. Refspec — the rule that maps their names onto your names — a string of the form `[+]<src>[:<dst>]` used by fetch and push.
2. Remote-tracking ref — your read-only copy of their branch — a ref under `refs/remotes/<remote>/`, updated only by fetch, push and remote prune.
3. Force flag — permission to lose history on one ref — the `+` prefix in a refspec, or `--force` on the command line.
4. Scp-like syntax — the short SSH address form — `user@host:path`, which cannot carry a port number.
5. `insteadOf` — a find-and-replace for remote addresses — configuration keys `url.<base>.insteadOf` and `url.<base>.pushInsteadOf`.
6. `FETCH_HEAD` — a scratch note of what the last fetch brought — a file listing the refs and hashes downloaded by the most recent fetch.

40.5 Hooks: server-side and client-side

■ 40.5.1 PLAIN – in simple words

1. A **hook** is a small program that git runs automatically at a certain moment.
2. Hooks live in the `hooks` directory of a repository. They are ordinary executable files. Shell, Python, anything the machine can run.
3. Git ships them as `.sample` files, which are inert. Rename one to remove the `.sample` and it becomes live.

4. There are hooks that run on your own machine, and hooks that run on the server when a push arrives.
5. The three important server hooks are pre-receive, update and post-receive.
6. pre-receive runs once for the whole push, before anything is saved. If it exits with a non-zero status, the entire push is rejected.
7. update runs once per branch or tag being changed. It can reject one branch and let the others through.
8. post-receive runs once after everything is saved. It cannot reject anything. It is for doing things afterwards: deploying, notifying, triggering a build.
9. Client hooks such as pre-commit and commit-msg run on your own machine. They are helpful, and they are not enforcement, because anybody can skip them with one flag.

■ 40.5.2 PLAIN – a picture in your head

1. Think of a building with a security desk at the entrance and a noticeboard inside.
2. pre-receive is the guard who checks the whole delivery at the door. If any item on the trolley is wrong, the whole trolley goes back.
3. update is a second guard who checks each box on the trolley separately. Some boxes may pass while others are turned away.
4. post-receive is the person who, once the delivery is inside, updates the noticeboard and phones the department that ordered it. That person cannot send the delivery back. It is already in.
5. Client hooks are the checklist taped to your own desk before you leave the house. Useful. Easy to ignore.

Where this comparison breaks:

6. A guard can be persuaded. A hook is a program and does exactly what it says, every time, with no judgement.
7. And a real guard sees the whole delivery. A hook sees only three pieces of text per ref: the old hash, the new hash and the ref name. Everything else it must look up for itself.

■ 40.5.3 PLAIN – a worked example

1. Here is a real, working post-receive hook that deploys a website. It is in the server repository at hooks/post-receive and is executable.

```
#!/bin/sh
# Deploy the site when main is updated.
DEPLOY_DIR=/tmp/lab/www
BRANCH=refs/heads/main

while read old new ref
do
    if [ "$ref" = "$BRANCH" ]; then
        echo "post-receive: deploying $ref"
        echo "post-receive: old=$(git rev-parse --short $old)"
        echo "post-receive: new=$(git rev-parse --short $new)"
        git --work-tree="$DEPLOY_DIR" --git-dir=. checkout -f main
        echo "post-receive: deployed to $DEPLOY_DIR"
    else
        echo "post-receive: ignoring $ref"
    fi
done
```

2. Line by line, because every line earns its place.
3. `#!/bin/sh` is the shebang. It tells the kernel which interpreter to run. Without it, git will try to execute the file directly and fail.
4. `DEPLOY_DIR` is where the website files should end up. In production this would be the directory your web server serves.
5. `BRANCH` is the only branch we deploy. Pushing a feature branch must not change the live site.
6. `while read old new ref` is the key line. Git feeds `post-receive` one line per updated ref on standard input, and each line has exactly three fields: the hash before, the hash after, and the full ref name.
7. The `if` compares the ref name to `refs/heads/main`. Note it is the full ref name, not the short name. A hook never sees `main` on its own.
8. `git rev-parse --short` turns a 40-character hash into a 7-character one for the log line. For a brand new branch the old hash is forty zeros.
9. The deploy line is the interesting one. `--work-tree` tells git to put the files somewhere other than the repository, and `--git-dir=.` says the history is right here. `checkout -f main` then writes every file of `main` into the deploy directory, overwriting whatever is there.
10. `echo` output from a server hook is sent back to the person pushing, prefixed with `remote:.` That is how you talk to your users.
11. Now the real push, with the hook live. The deploy directory was empty before this command.

```
$ git add index.html && git commit -q -m 'Add home page'

$ git push origin main
remote: post-receive: deploying refs/heads/main
remote: post-receive: old=6495241
remote: post-receive: new=adb4a16
remote: Already on 'main'
remote: post-receive: deployed to /tmp/lab/www
To /tmp/lab/srv/proj.git
    6495241..adb4a16  main -> main

$ ls -la /tmp/lab/www
-rw-r--r-- 1 root root  21 Aug 13 03:12 NOTES.md
-rw-r--r-- 1 root root  47 Aug 13 03:12 README.md
-rw-r--r-- 1 root root  18 Aug 13 03:12 app.py
-rw-r--r-- 1 root root  17 Aug 13 03:12 index.html

$ cat /tmp/lab/www/index.html
<h1>KedByte</h1>
```

12. That is a complete deployment system in fourteen lines of shell. No service, no agent, no vendor.

■ 40.5.4 PLAIN - what is really happening inside

1. Now the two hooks that can say no. First `pre-receive`, enforcing a branch naming policy.

```
#!/bin/sh
fail=0
while read old new ref
do
    case "$ref" in
        refs/heads/main)      ok=yes ;;
        refs/heads/feature/*) ok=yes ;;
        refs/tags/*)          ok=yes ;;
        refs/notes/*)         ok=yes ;;
        *)                    ok=no  ;;
    esac
done
```

```

esac
if [ "$ok" = "no" ]; then
    echo "policy: bad branch name: $ref"
    fail=1
fi
done
if [ "$fail" = "1" ]; then
    echo "policy: whole push rejected"
    exit 1
fi
exit 0

```


8. Here is a real force push being stopped by that rule.

```
$ git commit -q --amend -m 'Add about page (message fixed)'

$ git push --force origin main
remote: policy: main is fast-forward only
remote: error: hook declined to update refs/heads/main
To /tmp/lab/srv/proj.git
 ! [remote rejected] main -> main (hook declined)
error: failed to push some refs to
      '/tmp/lab/srv/proj.git'

$ git ls-remote origin refs/heads/main
a07dcbcd1fbb04efa6dbb8f6272a2aed8c361556      refs/heads/main
```

9. That is branch protection, built by hand, in twenty lines. GitHub's branch protection is the same idea with a web form in front of it.
10. One honest note from the same run. When we tried `git push origin :main` to delete the branch, git's own built-in `receive.denyDeleteCurrent` check fired before our hook did, because `main` is the branch that the bare repository's `HEAD` points at. Our hook line is still worth having, for every branch that is not the current one.

■ 40.5.5 TECHNICAL – the engineer's version

1. Server-side hooks, in the exact order git runs them on a push:

Hook	Runs	Can reject	Input
pre-receive	once per push	yes, all refs	stdin
update	once per ref	yes, that ref	argv
post-receive	once per push	no	stdin
post-update	once per push	no	argv

2. `post-update` is the oldest of these and its main modern use is running `git update-server-info`, which is only needed for the “dumb” HTTP transport, where a plain web server serves the repository files with no git-aware program behind it.
3. Client-side hooks that matter day to day:

Hook	When	Typical use
pre-commit	before message editor	linting, secret scanning
prepare-commit-msg	before editor opens	insert a template
commit-msg	after message written	enforce message format
pre-push	before push starts	run tests
pre-rebase	before a rebase	protect published branches

4. Here are two real client hooks and the real refusals they produce. First `pre-commit`, refusing a marker string:

```
#!/bin/sh
if git diff --cached -U0 | grep -q 'DO-NOT-COMMIT'; then
    echo "pre-commit: found DO-NOT-COMMIT marker"
    exit 1
fi
exit 0
```

```
$ git add secrets.py && git commit -m 'KB-42: add config'
pre-commit: found DO-NOT-COMMIT marker
```

5. Then `commit-msg`, enforcing a ticket prefix. Git passes the path of the file holding the message as the first argument.

```
#!/bin/sh
first=$(head -n 1 "$1")
if ! echo "$first" | grep -Eq '^KB-[0-9]+: .+'; then
    echo "commit-msg: need a KB-<number>: prefix"
    echo "commit-msg: you wrote: $first"
    exit 1
fi
exit 0

$ git commit -m 'add contact page'
commit-msg: need a KB-<number>: prefix
commit-msg: you wrote: add contact page

$ git commit -m 'KB-43: add contact page'
[main 40048a8] KB-43: add contact page
1 file changed, 1 insertion(+)
```

6. And now the honest part, demonstrated rather than asserted. One flag skips both hooks, and the resulting commit pushes perfectly happily.

```
$ git commit --no-verify -m 'whatever, no ticket, no checks'
[main a2648f3] whatever, no ticket, no checks
1 file changed, 1 insertion(+)
create mode 100644 leak.py

$ git push -q origin main && git ls-remote origin main
a2648f3588da557802930bc984745c8a574b7a38 refs/heads/main
```

7. The secret marker went in. The ticket prefix was skipped. Both hooks were bypassed by a single documented flag.
8. **Client hooks are not enforcement.** They are a convenience that catches honest mistakes. Three independent reasons: `--no-verify` exists, hooks are not committed to the repository so a fresh clone has none, and a hook file that is not executable is silently ignored.
9. Because hooks are not versioned, teams use a manager that installs them from a committed configuration. The common ones are `pre-commit` (a Python tool, first released in 2014), `Husky` for Node.js projects, and setting `core.hooksPath` to a directory that is committed. `core.hooksPath` arrived in Git 2.9 (June 2016) and is the plainest option.
10. Real enforcement lives on the server: `pre-receive` and `update` hooks on a self-hosted server, or branch protection plus required status checks on a hosted platform. Everything else is advice.

■ 40.5.6 WORDS – remember these

1. Hook — a program git runs at a set moment — an executable in the `hooks` directory or in `core.hooksPath`, invoked with a defined argument or `stdin` contract and judged by its exit status.
2. `pre-receive` — the all-or-nothing gate on a push — a server hook reading `old new ref` lines on `stdin`, whose non-zero exit rejects every ref.
3. `update` — the per-branch gate — a server hook called once per ref with three arguments, whose non-zero exit rejects that ref only.

4. post-receive — the after-the-fact action — a server hook that runs once the refs are updated and cannot reject anything.
5. no-verify — the flag that skips client hooks — `--no-verify` on commit and push, which is why client hooks are advice and not policy.
6. core.hooksPath — a way to share hooks with a team — a configuration key pointing at a directory of hooks, which can be committed to the repository.

40.6 What GitHub actually is

■ 40.6.1 PLAIN – in simple words

1. GitHub is four things stacked on top of each other.
2. One: hosting for bare repositories. Machines that hold your repositories and run `git-upload-pack` and `git-receive-pack` for you.
3. Two: a web interface. Pages that let you read code, history and differences in a browser, and comment on them.
4. Three: an identity and permission system. Accounts, organizations, teams, tokens, and rules about who may do what to which repository.
5. Four: automation. Servers that run your tests and builds when something happens, and store the results.
6. GitHub was founded in 2008 by Tom Preston-Werner, Chris Wanstrath and P. J. Hyett. The site opened to the public in April 2008.
7. Microsoft announced it would buy GitHub on 4 June 2018 for 7.5 billion United States dollars in Microsoft stock. The purchase completed on 26 October 2018.
8. Here is the sentence that matters most in this chapter: the git part of GitHub is not the valuable part.
9. Anybody can host bare repositories. It took us one command and ten milliseconds. What is hard to copy is the other three layers, and above all the fact that millions of developers already have an account there.

■ 40.6.2 PLAIN – a picture in your head

1. Think of a public library that also happens to have meeting rooms, a membership desk and a printing service.
2. Holding books is the easy part. Any building with shelves can hold books.
3. What makes the library useful is that you know where it is, everyone else uses it too, there is a catalogue, there is a librarian who checks membership cards, and there is a room you can book.
4. GitHub is that. The shelves are cheap. The catalogue, the membership system and the fact that everyone is already a member are the product.
5. That is also why leaving is harder than it looks. Your books come with you. Your reading room bookings, your notes in the margins and your library card do not.

Where this comparison breaks:

6. A library lends you one copy and keeps the original. Git gives you the whole collection every time, so the “original” has no special status.
7. And a library card is issued by the library. Your git history is not issued by GitHub. It is yours, complete, on your own disk.

■ 40.6.3 PLAIN – a worked example

1. Here is proof that GitHub's git layer is ordinary git, taken live during the writing of this chapter.

```
$ git ls-remote https://github.com/git/git | head -5
745601a9a94110d74769ab605ccd4f61339758d2    HEAD
165e5ad3169d0fd26637da3383a4514f1a9d1e72    refs/heads/bisect
426ec014a6630fb78c5a7334461bc33568f07743    refs/heads/jch
e9019fcafe0040228b8631c30f97aeladb61bcddc    refs/heads/maint
745601a9a94110d74769ab605ccd4f61339758d2    refs/heads/master

$ git ls-remote https://github.com/git/git | wc -l
5297
```

2. That is exactly the same command, producing exactly the same shape of output, as the one we ran against a folder in /tmp.
3. The git source repository on GitHub advertises 5,297 refs. Our little bare repository advertised seven. Same protocol, same program, different scale.
4. Now here is proof of the second layer, the part that is not git. GitHub stores pull requests as extra refs that plain git can see but that plain git never creates.

```
$ git ls-remote https://github.com/cli/cli 'refs/pull/*' \
| head -4
e9a3253762e768badaa1d4a5b3d267416d1e42f4    refs/pull/1/head
9966ca8bdb06ff86febbb49ff08480ef3fb4f790    refs/pull/10/head
24b83202679eeddb72bf9eb35752bf3f5ea737    refs/pull/100/head
b4b8e8a13bb99bae479929048ea5f64808d0bc32    refs/pull/10002/head

$ git ls-remote https://github.com/cli/cli 'refs/pull/*' \
| wc -l
5147
```

5. Five thousand one hundred and forty-seven pull-request refs. Git can list them, because they are refs. Git cannot create one, cannot comment on one, cannot approve one and does not know what “pull request” means.
6. That single distinction is the subject of the next two sections.

■ 40.6.4 PLAIN – what is really happening inside

1. When you `git push` to GitHub, this is the sequence.
2. Your git resolves `github.com` to an address. In the reader's own session it resolved to `20.207.73.82`, which is in a Microsoft-owned range, because GitHub is a Microsoft company and fronts traffic through Microsoft's network edge.
3. Your git opens a connection, either TCP port 22 for SSH or TCP port 443 for HTTPS.
4. It authenticates. Over SSH that is your key pair. Over HTTPS that is a personal access token or a credential helper.
5. GitHub's front end checks who you are and whether you may write to that repository. This check is GitHub's, not git's.
6. If allowed, GitHub runs `git-receive-pack` against the bare repository it holds for you. From this point the bytes are ordinary git.
7. GitHub's own pre-receive hooks then run. Branch protection, required signatures, push rules and secret scanning all live here.
8. If nothing objects, the refs update and GitHub's equivalent of post-receive fires. That is what creates the timeline entry, sends the notification, updates the pull request, and starts an Actions workflow.

9. Steps 4, 5, 7 and 8 are GitHub. Steps 3 and 6 are git. That is the whole boundary.

■ 40.6.5 TECHNICAL – the engineer’s version

1. Dates and figures worth committing to memory:

Event	Date
Git first released by Linux Torvalds	April 2005
GitHub launched publicly	April 2008
GitHub introduces Git LFS	April 2015
Microsoft announces acquisition	4 June 2018
Acquisition completes	26 Oct 2018
Password auth for git removed	13 Aug 2021
Unencrypted git:// disabled	15 Mar 2022

2. GitHub removed password authentication for git operations over HTTPS on 13 August 2021. Since then HTTPS pushes require a personal access token, an OAuth token, a GitHub App installation token, or SSH keys instead.
3. On 24 March 2023 GitHub replaced its RSA SSH host key after the private key was briefly exposed in a public repository. Every user with the old key in `known_hosts` saw a host key verification failure. This is a good reminder that host key pinning is real security with real operational cost.
4. GitHub’s git hosting uses a custom system for replication and routing. Publicly described pieces include Spokes, formerly called DGit, for three-way replication of repositories across servers, and a proxy layer in front of the git transports. The important engineering point is that nothing about this changes the wire protocol. Your git client cannot tell.
5. The scale of GitHub’s non-git surface is what makes it expensive to run: the REST API, the GraphQL API, webhooks, Actions runners, package registries, Pages hosting and code search. None of those are git.
6. Estimated user counts published by GitHub have grown from roughly 100 million developers reported in early 2023 to figures above that since. Treat any single number as approximate and dated, because the company publishes it as a marketing figure rather than an audited one.

■ 40.6.6 WORDS – remember these

1. Forge — a website that hosts repositories and adds collaboration — the general term covering GitHub, GitLab, Gitea, Forgejo and Bitbucket.
2. Hosting — somebody else runs the bare repository — a service that exposes upload-pack and receive-pack over authenticated transports.
3. Personal access token — a password made for programs — a revocable credential with a fixed set of scopes, used in place of a password.
4. Scope — the list of things a token is allowed to do — a named permission such as `repo`, `workflow` or `read:org` attached to a token.
5. `refs/pull` — GitHub’s storage for pull request heads — read-only refs created by GitHub, visible to `git ls-remote`, not creatable by git.

40.7 What GitHub adds that plain git does not have

■ 40.7.1 PLAIN – in simple words

1. Git has exactly one job: move history between repositories, exactly.
2. It has no idea what a pull request is. No idea what a review is. No idea what an issue, a label, a release, a package or a build is.
3. Every one of those is added by the website you happen to be using.
4. Here is the whole list, in plain words, one item per line.
5. Pull requests. A written request saying “please take my branch into yours”, with a conversation attached to it.
6. Code review. Named people reading the change and marking it approved, or asking for changes, or just commenting.
7. Branch protection. A rule saying “nobody may push straight into main”.
8. Required checks. A rule saying “the tests must pass before the merge button is allowed to work”.
9. CODEOWNERS. A file listing which people must review which folders.
10. Issues and projects. A shared to-do list, and a board of cards.
11. Releases. A named point in history with files attached, such as a compiled program somebody can download.
12. Packages. A place to publish libraries that other projects install.
13. Actions. Computers that run your tests and builds when something happens, and keep the logs.
14. Two programming interfaces, REST and GraphQL. Ways for programs to ask GitHub questions and give it orders.
15. Permissions. Accounts, organizations, teams, roles and tokens.
16. Not one of those is part of git. Git ships none of them, and would work exactly the same if GitHub had never existed.

■ 40.7.2 PLAIN – a picture in your head

1. Think of a shared workshop where people build furniture.
2. Git is one machine in the corner. It copies a design exactly, down to the last measurement, and hands the copy to somebody else.
3. That machine is superb at that job and does nothing else. It has no opinion about who you are or whether the design is any good.
4. The workshop around the machine is everything else. A pass reader on the door. A noticeboard of jobs to do. A sign-off sheet outside the paint room saying two people must approve the colour. An overnight test rig. A shelf of finished pieces with labels on them.
5. You can take the copying machine home and still build furniture.
6. You cannot take the noticeboard, the sign-off sheets, the pass reader or the test rig. They belong to the building.
7. GitHub is the building. Git is the machine.

Where this comparison breaks:

8. The building also stores the designs, so day to day the two feel like one thing. The split is in the protocol, not in the floor plan.
9. And the sign-off sheet is not decoration. On GitHub, branch protection really does stop the push at the door. It is a lock, not a note.

■ 40.7.3 PLAIN – a worked example

1. A pull request has two halves. One half is refs, which git can see. The other half is a database row, which git cannot see.

2. Here is the visible half, read live with plain git, no account, no token.

```
$ git ls-remote https://github.com/cli/cli 'refs/pull/10002/*'
b4b8e8a13bb99bae479929048ea5f64808d0bc32 refs/pull/10002/head

$ git ls-remote https://github.com/cli/cli 'refs/pull/*/head' \
  | wc -l
4981

$ git ls-remote https://github.com/cli/cli 'refs/pull/*/merge' \
  | wc -l
166
```

3. Read those two numbers. 4,981 pull requests have a head ref, which is the commit the author is proposing.
4. Only 166 have a merge ref. That is GitHub's pre-computed trial merge, kept only while a pull request is open and can be merged cleanly.
5. Now the invisible half. Git cannot answer any of these, at any price: who opened it, what the title says, who approved it, which comment sits on which line, whether the tests passed, whether the merge button works.
6. Proof by absence. Ask git for every command it has that mentions requests or reviews.

```
$ git help -a | tr ' ' '\n' | grep -iE 'request|review|issue'
request-pull
```

7. One result, and it is not what you think. `git request-pull` has existed since 2005 and prints an email for you to send to a maintainer by hand.

```
$ git request-pull main /tmp/mg/base feature/login
The following changes since commit cd18e975:

    Base commit (2026-08-13 07:43:39 +0000)

are available in the Git repository at:

    /tmp/mg/base feature/login

for you to fetch changes up to 00ee7d7c:

    Fix login typo (2026-08-13 07:43:39 +0000)

Reader (2):
    Start login
    Fix login typo

login.py | 2 ++
1 file changed, 2 insertions(+)
```

8. That is the true ancestor of the pull request. It produces text for a mailing list. It has no state, no approval, no button and no thread.
9. GitHub's version added the state, the approval, the button and the thread. That addition is the product.

■ 40.7.4 PLAIN – what is really happening inside

1. A pull request is three things glued together.
2. One: two ref names. A base, meaning where you want the work to land, and a head, meaning the work. Both are ordinary git refs.
3. Two: a row in a database, holding a number, a title, a body, a state (open, closed or merged) and an author.
4. Three: a conversation. Comments, review states, requested reviewers, labels, and links to issues.
5. When you press Merge, GitHub runs an ordinary git merge on its own server, into the base branch, using whichever strategy you picked.
6. Review states are values on that database row: APPROVED, CHANGES_REQUESTED, COMMENTED, DISMISSED, PENDING.
7. Branch protection is enforced by GitHub's own pre-receive hook. Earlier in this chapter we wrote a pre-receive hook by hand that rejected a push. GitHub runs a much larger version of the same thing.
8. A required status check is a record attached to a **commit SHA**, not to a branch. Something posts "the tests passed for this SHA" through the API.
9. The protection rule then asks one question: does the head SHA of this pull request carry a passing record with the required name?
10. That is exactly why, in the reader's own session, checking CI by run ID against the exact head SHA was stronger than trusting a green tick beside a branch name. The tick is drawn next to a name, and names move. The record is attached to a SHA, and a SHA never moves.
11. CODEOWNERS is the one hybrid. The file is a real blob inside your repository, so git stores and versions it. The rule "these owners must approve" is GitHub's, and git ignores the file completely.
12. Actions is a scheduler plus a fleet of machines. A push fires an internal event, a workflow file in the repository says what to run, and a runner picks up the job.
13. Releases and packages are file storage with a database index in front. The attached files are not git objects and are not in the history.
14. So the rule is short: refs and objects are git; anything with an opinion is GitHub.

■ 40.7.5 TECHNICAL – the engineer's version

1. The classification. "Truth lives in" means where the authoritative copy is.

Feature	git or GitHub	Truth lives in
Commit, tree, blob, tag	git	object database
Branch and tag refs	git	refs, packed-refs
Merge, rebase, cherry-pick	git	local repository
Push, fetch, clone	git	wire protocol
Server-side hooks	git	hooks directory
Submodules	git	gitlink plus file
Pull request	GitHub	GitHub database
Review state	GitHub	GitHub database
Branch protection rule	GitHub	GitHub database
Required status check	GitHub	GitHub database
CODEOWNERS file	git	a blob in the tree
CODEOWNERS behaviour	GitHub	rule engine
Issue, label, milestone	GitHub	GitHub database
Projects board	GitHub	GitHub database

Feature	git or GitHub	Truth lives in
Release and its assets	GitHub	object storage
Packages registry	GitHub	object storage
Actions workflow run	GitHub	runners plus DB
REST and GraphQL API	GitHub	GitHub servers
Teams, roles, tokens	GitHub	GitHub database
Fork, star, watch	GitHub	GitHub database

2. Dates, so you can see that every GitHub feature has a birthday and git has none of them.

GitHub feature	Date
Pull Requests 2.0 post	31 Aug 2010
Releases introduced	2 Jul 2013
Protected branches, checks	3 Sep 2015
GraphQL API announced	14 Sep 2016
CODEOWNERS introduced	6 Jul 2017
Actions announced	16 Oct 2018
Actions generally available	13 Nov 2019
Merge queue GA	12 Jul 2023

3. Rate limits, from GitHub's published documentation, current as of 2026. REST, unauthenticated: 60 requests per hour per IP address. REST, with a personal access token: 5,000 per hour. GitHub Apps and OAuth apps owned by a GitHub Enterprise Cloud organization: 15,000 per hour. GraphQL is budgeted in points, 5,000 points per hour for a user token.
4. The API keeps its own books, and will tell you so:

```
$ curl -s https://api.github.com/rate_limit
{ "resources": {
  "core": { "limit": 15000, "remaining": 15000 },
  "search": { "limit": 30, "remaining": 30 },
  "graphql": { "limit": 5000, "remaining": 5000 } } }
```

5. Honest note on that output: the request above ran through a proxy that attached an application credential, which is why core shows 15,000 and not 60. The number you see depends on how you authenticated.
6. Git has no equivalent of any of that, because git has no accounts. There is nothing in the git protocol to rate-limit against.
7. The permission model, as of 2026. Repository roles: Read, Triage, Write, Maintain, Admin. Organization roles: Member, Moderator, Billing manager, Owner, plus custom roles on paid plans. Teams can nest, and effective permission is the union of every grant a user receives.
8. Token types. Classic personal access tokens carry coarse scopes such as `repo`, `workflow`, `read:org` and `admin:repo_hook`. Fine-grained personal access tokens, generally available since 2022, are limited to chosen repositories, carry per-resource read or write permissions, and must have an expiry date.
9. The reader's own session shows why the scope list is not decoration. One push was refused because the token lacked the `workflow` scope and the push modified a CI workflow file. Git had already packed and sent the objects. GitHub's server-side rule refused the ref update.

10. GraphQL exposes fields with no git equivalent at all. The reader saw `mergeStateStatus` values `CLEAN`, `BLOCKED` and `UNKNOWN`; the full set also includes `BEHIND`, `DIRTY`, `DRAFT`, `HAS_HOOKS` and `UNSTABLE`.
11. Git cannot compute any of those values, because each depends on a rule git does not know exists.
12. Scale figures published by GitHub are marketing figures, not audited ones. Roughly 100 million developers were reported in early 2023, about 150 million for May 2025, and the repository count was reported to pass one billion in June 2025. Treat all three as approximate and dated.

■ 40.7.6 WORDS - remember these

1. Pull request — a request to merge one branch into another, with a discussion — a GitHub database object referencing a base ref and a head ref, exposed to git only as `refs/pull/N/head`.
2. Review state — the verdict a reviewer left — one of `APPROVED`, `CHANGES_REQUESTED`, `COMMENTED`, `DISMISSED`, `PENDING`.
3. Branch protection — a rule stopping direct pushes to a branch — a server-side policy implemented in GitHub's pre-receive path.
4. Required status check — the tests must pass before merging — a named check record that must exist and be successful for the head SHA.
5. Check run — one reported result for one commit — an API object attached to a SHA with a name, status, conclusion and output.
6. CODEOWNERS — a file saying who owns which folders — a versioned path pattern file whose enforcement is entirely the forge's, not git's.
7. Fork — your own server-side copy of somebody's repository — a GitHub database relationship plus a repository that may share object storage.
8. GraphQL API — one endpoint you send a shaped query to — GitHub's typed schema at a single endpoint, budgeted in points rather than requests.
9. `mergeStateStatus` — GitHub's summary of whether merging is allowed — a GraphQL enum with values such as `CLEAN`, `BLOCKED` and `BEHIND`.
10. Merge queue — a line that tests changes in order before merging — a GitHub service that builds candidate merges and merges only what passes.

40.8 The exact boundary, using the reader's own outage as evidence

■ 40.8.1 PLAIN - in simple words

1. During the reader's outage one sentence describes everything: git kept working, and the GitHub API did not.
2. Committing worked. Branching worked. Merging worked. Rebasing worked. Reading old history worked. Tagging worked.
3. Pushing failed. Fetching failed. Looking at a pull request failed. Checking a CI run failed.
4. That is not luck and not a coincidence. It follows from two facts the book has already given you.
5. Fact one: a git command needs the network if and only if it must read from, or write to, another repository. Everything else is reading and writing files inside the `.git` folder, and files need no network.
6. Fact two: every GitHub-specific feature is a network service. There is no local copy of a pull request, an issue, a review or a build result.
7. Put the two together and the boundary is exact. The local half of git kept working. The remote half of git, and the whole of GitHub, did not.

- Nothing here is special to GitHub. Point your remote at a self-hosted Forgejo and the same line falls in the same place.

■ 40.8.2 PLAIN – a picture in your head

- Imagine you keep a paper diary at home, and your office keeps a shared wall planner that everyone updates.
- The phone line to the office is cut.
- You can still write in your diary. Every page, any amount. The cut line does not touch your pen.
- You can still read every old page, because they are in your hands.
- You cannot see today's wall planner, and you cannot change it.
- Here is the sharp part. You can still look at the photo of the wall planner you took last Tuesday. It will show you Tuesday's planner, confidently, and it will be out of date.
- That photo is `origin/main`. During the outage the reader's machine reported `origin/main` as `0a95cc8`. That was a cached value from the last successful fetch, not a live fact about the server.

Where this comparison breaks:

- A photo obviously looks like a photo. A remote-tracking ref does not. `git log origin/main` prints normally, with no warning and no note saying how old the information is.
- And a wall planner suggests one true copy. Git has no true copy. During the outage `origin/main` was stale, but the reader's own `main` was not somehow less real.

■ 40.8.3 PLAIN – a worked example

- Here is the boundary as a table. Read the third column as the reason, not as an excuse.

Operation	Works offline	Why
<code>git commit</code>	yes	writes <code>.git</code> only
<code>git branch</code> , <code>git switch</code>	yes	writes a ref file
<code>git merge</code> , <code>git rebase</code>	yes	objects already local
<code>git log</code> , <code>git diff</code> , <code>blame</code>	yes	reads local objects
<code>git tag</code> , <code>git stash</code>	yes	local refs and objects
<code>git bisect</code>	yes	walks the local graph
<code>git show origin/main</code>	yes, but stale	cached tracking ref
<code>git fetch</code> , <code>git pull</code>	no	must read the remote
<code>git push</code>	no	must write remote refs
<code>git clone</code>	no	nothing local yet
<code>git ls-remote</code>	no	asks the other side
Open or merge a PR	no	GitHub database only
Read review comments	no	GitHub database only
Check a CI run result	no	GitHub service only
Create or close an issue	no	GitHub database only
Get a release asset	no	GitHub storage only
Branch protection verdict	no	server-side rule
Any gh CLI command	no	wraps the GitHub API

- Every row above was reproduced in a sandbox for this chapter. The remote was pointed at `203.0.113.9`, an address reserved by RFC 5737 for documentation, so nothing could answer. The outage was real, not faked with a flag.

```

git log --oneline -1      rc=0 f657694 First commit
git branch -a             rc=0 * main
git switch -c feature/x   rc=0
git commit --allow-empty  rc=0
git tag v0.9              rc=0
git diff HEAD~1 --stat    rc=0
git rev-parse origin/main rc=0 f6576945ec22d250a17a6fe3...
git status -sb            rc=0 ## feature/x

```

3. Note line seven. `git rev-parse origin/main` succeeded and printed a full SHA while the server was unreachable. That is the cached photo.
4. In the same second, on the same repository, the network commands:

```

$ git ls-remote origin
ssh: connect to host 203.0.113.9 port 22: Connection timed out
fatal: Could not read from remote repository.

$ git fetch origin
ssh: connect to host 203.0.113.9 port 22: Connection timed out
fatal: Could not read from remote repository.

$ git push origin main
ssh: connect to host 203.0.113.9 port 22: Connection timed out
fatal: Could not read from remote repository.

```

5. Three failures, all identical, all at the same layer: the connection never opened. Git never got as far as speaking git.
6. That is the same shape as the reader's own evidence. `curl -v https://github.com` printed Trying 20.207.73.82:443... and then timed out after 15 seconds with no response of any kind. No RST, no ICMP unreachable, silence.
7. What that proves: nothing on that path answered. What it does not prove: who dropped it. The same request succeeded instantly over mobile data, which is consistent with a silent drop somewhere on the fixed-line path, and not consistent with GitHub being down.

■ 40.8.4 PLAIN – what is really happening inside

1. Take the three cases apart, one at a time.
2. Case one, a local git command. `git commit` reads the index file, writes blob, tree and commit objects under `.git/objects`, and rewrites one 41-byte ref file. Zero sockets are opened. There is nothing to fail.
3. Case two, a remote git command. `git push` must change a ref on another machine. It resolves a name, opens TCP port 22 or 443, authenticates, runs the remote helper, and negotiates. Any broken step ends it.
4. Case three, a GitHub feature. Opening a pull request is an HTTPS request to `api.github.com`. There is no offline path, because there is no local copy of the data and no local code that understands it.
5. Now the important consequence for the reader. Because case one never touches a network, work does not stop. The reader committed to branches all through the outage and pushed them later, when the path came back.
6. Nothing had to be redone. The commits made during the outage were already complete, already hashed, already in the object database. Pushing later just copied objects that already existed.
7. That is the entire practical payoff of the distributed design, and the reader lived it.

8. One more piece from the same session. Twice, a push failed with `send-pack: unexpected disconnect while reading sideband packet`.
9. That message is different from the ones above. It means the connection was established, authentication passed, objects were sent, and the link died while git was reading the server's reply.
10. So the error tells you the answer never arrived. It tells you nothing about whether the server acted before the link died.
11. Both times the operation had not been applied. But that could not be known from the message. It had to be re-checked with `git ls-remote origin`, which asks the server what its refs are now.
12. The rule to carry away: a failure while reading a response is not evidence that the write did not happen. Only a fresh query is evidence.

■ 40.8.5 TECHNICAL – the engineer's version

1. What each operation actually needs on the wire.

Operation	Endpoint	Port
git over SSH	ssh to the host	22
git over HTTPS	the smart HTTP path	443
GitHub REST API	api.github.com	443
GitHub GraphQL API	api.github.com/graphql	443
gh CLI	the same two APIs	443
Web pages, PR view	github.com	443
Release asset download	objects storage host	443
Actions log fetch	a logs host	443

2. Note what that table means. Everything except plain SSH shares one port and often one path. When that path is silently dropped, git over HTTPS and the whole API fail together, which is what the reader observed.
3. Plain SSH on port 22 is a different destination and can survive when HTTPS does not. In the reader's session one transport worked while the other failed, in the same minute, on the same machine.
4. That asymmetry is a diagnostic, not a fix. If SSH works and HTTPS does not, the fault is path-specific or port-specific, not a dead server.
5. Failure taxonomy, with the exact signal for each:

Symptom	What it means	Proven or suggested
Connection refused	a host sent RST	server or firewall said no
Timed out, no reply	silent drop	proven: nothing answered
TLS handshake error	reached, policy fail	proven: TCP worked
403 from the API	authenticated, denied	proven: service is up
sideband disconnect	reply lost mid-stream	write status unknown

6. Only the last row is ambiguous, and it is the one that bites. The git protocol sends the `report-status` response inside sideband channel 1 during `receive-pack`. If the stream dies there, the client has no report.
7. The correct recovery is a re-query, never a retry decided by guesswork:

```
$ git ls-remote origin refs/heads/main
0a95cc8a5f0b... refs/heads/main
```

8. Compare that SHA with your local main. If they match, the push landed. If not, it did not. That is a fact from the server, not an inference.
9. There is a second reason the re-query matters. A push is not atomic across refs by default. Pushing several refs at once can leave some applied and some not, unless you use `--atomic`, added in Git 2.4 (April 2015), which makes the whole push succeed or fail as one unit on servers that support it.
10. Chapter 37 gave the local versus remote split. This chapter adds the second axis. Together they explain every row of the table in 40.8.3:

	needs no network	needs the network
pure git	commit, log, merge, rebase, tag, bisect	fetch, push, clone, ls-remote
forge feature	(nothing)	PR, review, CI, issues, releases

11. The bottom-left cell is empty, and that emptiness is the point. There is no such thing as an offline pull request.
12. Tools that appear to break this rule cache data locally, which is the same stale-photo problem as origin/main and should be treated the same way.

■ 40.8.6 WORDS – remember these

1. Local operation — anything git can do with your own files — a command whose entire working set is the object database, index and refs on this disk.
2. Remote-tracking ref — your last photo of the server — a ref under refs/remotes/ updated only by fetch, never live.
3. Silent drop — a packet vanishes with no reply — no RST and no ICMP time-exceeded, observed as a connection timeout.
4. Sideband — the channel carrying the server's messages — multiplexed stream inside the pack protocol, where the report-status reply travels.
5. Re-query — ask again instead of guessing — using `git ls-remote` to learn the server's current refs after an ambiguous failure.
6. Atomic push — all refs or none — `git push --atomic`, added in Git 2.4 (April 2015), requiring server support.

40.9 The self-hosted alternatives

■ 40.9.1 PLAIN – in simple words

1. If you do not want to use GitHub, you have four honest choices. Here they are from heaviest to lightest.
2. GitLab Community Edition. Close to GitHub in features, and heavy to run.
3. Gitea, or its community fork Forgejo. Most of what a small team needs, in a program small enough for a Raspberry Pi.
4. Bitbucket Data Center from Atlassian. Bought mainly by companies that already run Jira and want one supplier.
5. Plain SSH remotes with no web interface at all. A machine, an account, and git. Nothing else.
6. All four store bare repositories the same way and speak the same protocol. Every difference is in the layers above git.

7. What you gain by self-hosting: control over the rules, privacy for the code, and no lock-in to one company's features or prices.
8. What you lose is simpler than people admit. You become the operator. Backups are yours. Uptime is yours. Upgrades and security patches are yours. At three in the morning, it is you.

■ 40.9.2 PLAIN – a picture in your head

1. Compare eating in a canteen with cooking at home.
2. In the canteen, somebody else buys the food, cooks it, washes up and keeps the doors open. You accept their menu and pay their price.
3. Cooking at home, you choose everything. You know exactly what is in the food. Nobody can change the price or close the kitchen.
4. But you also shop, cook, wash up and mend the cooker. Every day. Including days you are ill and days you are away.
5. Most people cook at home for four and hire a caterer for four hundred.
6. Self-hosting scales the same way. It is genuinely easy for a small trusted group and genuinely hard for a large public project full of strangers.

Where this comparison breaks:

7. A broken cooker ruins one dinner. A broken git server does not lose your history at all, because every clone is a complete copy. The stakes are lower than the picture suggests.
8. What an outage really costs you is the meeting point, plus the metadata: issues, reviews and build results. Those are the parts with no second copy on anybody's laptop.

■ 40.9.3 PLAIN – a worked example

1. Here is the smallest useful server anyone can build, and it needs no software beyond git and an SSH daemon.
2. Make one account on a machine, called git. Set its login shell to `git-shell`, a program that ships with git and refuses everything except git's own transport commands.
3. Put each developer's public key in that account's `authorized_keys` file.
4. Proof that `git-shell` really is a lock and not a suggestion:

```
$ git-shell -c "ls -la"
fatal: unrecognized command 'ls -la'
```

5. And proof that the git commands still work through it. This is the real first bytes of a ref advertisement, wrapped to fit the page:

```
$ git-shell -c "git-upload-pack '/srv/git/proj.git'"
01032304fd03...16d6 HEAD multi_ack thin-pack side-band
side-band-64k ofs-delta shallow deepen-since deepen-not
no-progress include-tag multi_ack_detailed
symref=HEAD:refs/heads/main object-format=sha1
agent=git/2.43.0
0046a3bfdce1...3a refs/heads/feature/login
```

6. That is the whole server. Somebody who logs in with that key can clone, fetch and push, and cannot get a shell, list files or read other accounts.
7. Cost of running it: git itself. Measured in the sandbox for this chapter, peak memory of the git process during real work:

```
git clone (11,772 commits) 0.58 s peak RSS 42.1 MB
git gc (blobless clone) 0.93 s peak RSS 39.2 MB
```

```
git status (1,356 files)    0.29 s    peak RSS 16.5 MB
git log --oneline (all)     0.17 s    peak RSS 14.8 MB
```

8. Tens of megabytes, for a few seconds, per operation. That is why a plain SSH git server runs happily on the smallest machine you can rent.
9. Now compare that with what each option needs before it will start at all.

Option	Min RAM	Web UI	Effort
SSH plus git-shell	tens of MB	none	very low
Gitolite added	tens of MB	none	low
cgit or gitweb	~128 MB	read only	low
Gitea or Forgejo	1 GB	full	low
GitLab CE	8 to 16 GB	full	high
Bitbucket Data Center	8 GB and up	full	high, paid

10. And what you actually get for that memory.

Feature	SSH only	Gitea, Forgejo	GitLab CE
Push and pull	yes	yes	yes
Server-side hooks	yes	yes	yes
Web code browsing	no	yes	yes
Pull requests	no	yes	yes, called MRs
Issues, labels	no	yes	yes
Built-in CI	no	yes	yes
Package registry	no	yes	yes
Fine-grained roles	limited	yes	yes

■ 40.9.4 PLAIN – what is really happening inside

1. The four options differ in exactly one way: how many processes must be running before a developer can do anything.
2. Plain SSH: one process, and only while somebody is connected. `sshd` accepts the connection, `git-shell` checks the command, and `git-upload-pack` or `git-receive-pack` does the work and exits.
3. Nothing runs between pushes. There is no database, no queue, no web server and no cache. That is why it costs almost nothing.
4. Gitolite adds a single Perl program in front. It reads one configuration repository that lists users, repositories and per-branch rules, and it installs an update hook to enforce them.
5. Gitea and Forgejo are one compiled program written in Go. It contains the web server, the issue tracker, the review system and its own SSH server. Storage is SQLite for small installations, or PostgreSQL or MySQL.
6. GitLab is not one program. A standard installation runs a Ruby on Rails application, background job workers, PostgreSQL, Redis, a reverse proxy, a metrics stack, and Gitaly, which is GitLab's own service that every other part must call in order to touch git at all.
7. That list is the entire reason for the memory difference. It is not that GitLab is badly built. It is that it is many services, and each one has a floor.

8. In every case the bytes on the wire are identical. A developer cloning from plain SSH and a developer cloning from GitLab run the same client, speak the same protocol and get the same objects.

■ 40.9.5 TECHNICAL – the engineer’s version

1. Provenance and licensing, which matter more than feature lists when you are choosing something to depend on for ten years.

Project	Origin	Licence
Gogs	2014, by Jiahua Chen	MIT
Gitea	Nov 2016 fork of Gogs	MIT
Forgejo	Dec 2022 fork of Gitea	GPL since Aug 2024
GitLab CE	2011, D. Zaporozhets	MIT
GitLab EE	same codebase, add-ons	proprietary
Bitbucket	2008, by Jesper Nohr	proprietary

2. The Gitea and Forgejo split is a governance story, not a technical one. Gitea’s trademarks and operations were transferred to a company in 2022, contributors asked for community control, the request was refused, and Forgejo was forked in December 2022. Codeberg e.V., a German non-profit association, leads it.
3. Forgejo stopped tracking Gitea after Gitea 1.21, in February 2024, and became a hard fork. It relicensed from MIT to the GNU General Public Licence in August 2024. Code written before that remains MIT.
4. Bitbucket was founded in 2008 by Jesper Nohr, acquired by Atlassian on 29 September 2010, and dropped Mercurial support in July 2020. Support for the self-hosted **Bitbucket Server** product ended on 15 February 2024. Self-hosting now means Bitbucket Data Center, which is licensed and priced for larger organizations.
5. That last date is a lesson in its own right. A self-hosted product can be discontinued by its vendor. Self-hosting removes the hosting dependency; it does not by itself remove the vendor dependency. Only an open licence with a living community does that.
6. Hardware, from the projects’ own documentation as of 2026. Gitea states that “2 CPU cores and 1GB RAM is typically sufficient for small teams/projects” and that a Raspberry Pi 3 is enough for small workloads. GitLab’s single-node baseline is 8 vCPU and 16 GB of RAM, with 8 GB possible only with extra configuration. Older GitLab documentation quoted 4 GB, so check the version-specific page rather than trusting a number you remember.
7. That is a factor of roughly sixteen in memory between Gitea and GitLab for the same core job. It is the single most useful figure in this section.
8. Access control on a plain SSH server, in increasing order of precision: UNIX file permissions on the repository directory; one shared git account with git-shell; Gitolite for per-repository and per-branch rules; and server-side pre-receive or update hooks for anything else.
9. Read-only web browsing without a full forge: cgit, a C program, or gitweb, which ships with git itself. Both render a repository as HTML and neither has accounts, issues or reviews.
10. Honest trade-off summary. Self-hosting is a good decision when the group is small and trusted, when the code must not leave your control for legal reasons, when your costs at a hosted provider have become large, or when you need rules a hosted provider will not give you.

11. It is a poor decision when your project depends on drive-by contributions from strangers, when nobody on the team wants to be on call, or when the real cost of an engineer's time doing operations exceeds the subscription you were avoiding.
12. Experts disagree about the middle ground. One camp argues that a Forgejo instance is so cheap to run that any team should own its own. The other argues that discovery, and the fact that contributors already have GitHub accounts, outweighs everything else for public projects. Both are right about different projects, and the deciding question is whether strangers need to contribute.

■ 40.9.6 WORDS – remember these

1. Self-hosting — running the server yourself — operating the repository host, its storage, its backups and its upgrades on infrastructure you control.
2. git-shell — a login that can only do git — a restricted shell shipped with git that accepts only git-upload-pack, git-receive-pack and git-upload-archive.
3. Gitolite — a rules file in front of a git server — a Perl layer using one configuration repository and an update hook for per-branch access control.
4. Forge — a website around repositories — the general term for GitHub, GitLab, Gitea, Forgejo and Bitbucket.
5. Gitaly — GitLab's git service — the component every other GitLab process must call to read or write a repository.
6. Vendor lock-in — hard to leave — dependence on features whose data has no portable export, as opposed to the commit history, which always exports.
7. Operator — the person who is responsible when it breaks — the role you take on the moment you self-host, covering backups, uptime and patching.

40.10 Migrating and mirroring

■ 40.10.1 PLAIN – in simple words

1. Two jobs get confused with each other, so separate them first.
2. Mirroring means keeping a second copy that stays in step with the first. Both live. You did not leave anywhere.
3. Migrating means moving to a new home and shutting the old one, or leaving it as an archive.
4. For the git part, both jobs are easy and take one or two commands.
5. For everything else, both jobs are hard, and some of it simply cannot be moved at all.
6. Here is the sentence to remember, and it is the honest warning of this whole section: **your commit history is portable, and almost nothing else is.**
7. Commits, branches, tags and file contents move perfectly, byte for byte, with identical hashes.
8. Issues, pull requests, review approvals, build history, release downloads, stars, webhooks and protection rules do not move by themselves, because none of them is in git.

■ 40.10.2 PLAIN – a picture in your head

1. Think about moving house.
2. Your books, clothes and furniture go in boxes. They arrive exactly as they left. Nothing about them changes.
3. The fitted kitchen does not come. The neighbours do not come. Your children's school place does not come. The postal redirection has to be arranged, lasts a while, and then stops.
4. Git history is the boxes. Everything the website added is the fitted kitchen and the school place.
5. And notice which one people actually worry about on moving day. Not the boxes.

Where this comparison breaks:

6. Moving house removes things from the old place. Cloning does not. After you migrate, the old repository still exists, complete, and can still serve.
7. That is why a git migration is unusually safe. You can run both for weeks and switch back with one `git remote set-url`.

■ 40.10.3 PLAIN – a worked example

1. First, what `--mirror` actually gives you. This is a real run.

```
$ git clone --mirror proj mirror.git

$ git -C mirror.git config --get remote.origin.mirror
true

$ git -C mirror.git config --get remote.origin.fetch
+refs/*:refs/*

$ git -C mirror.git show-ref
a3bfdce1... refs/heads/feature/login
2304fd03... refs/heads/main
fb32611e... refs/tags/v1.0
```

2. Compare that with an ordinary clone of the same repository:

```
$ git clone proj plain
$ git -C plain show-ref
2304fd03... refs/heads/main
2304fd03... refs/remotes/origin/HEAD
a3bfdce1... refs/remotes/origin/feature/login
2304fd03... refs/remotes/origin/main
fb32611e... refs/tags/v1.0
```

3. Read the difference carefully. The ordinary clone has one local branch and puts the rest under `refs/remotes/origin/`. The mirror keeps every ref at its original name.
4. That is the whole point of a mirror: it is a copy of the ref namespace, not a copy for working in.
5. Second, keeping a mirror up to date. One command, and `--prune` matters:

```
$ git -C mirror.git remote update --prune
From /tmp/mir/up
- [deleted]    (none) -> hotfix/urgent
```

6. Third, pushing one commit to two servers with one command. Add a second push URL to the same remote:

```
$ git remote set-url --add --push origin /tmp/lab3/srv1.git
$ git remote set-url --add --push origin /tmp/lab3/srv2.git
$ git remote -v
origin /tmp/lab3/srv1.git (fetch)
origin /tmp/lab3/srv1.git (push)
origin /tmp/lab3/srv2.git (push)

$ git push origin main
To /tmp/lab3/srv1.git
  99b5afc..a98de55  main -> main
To /tmp/lab3/srv2.git
  * [new branch]    main -> main
```

- One `git push`, two servers, and both now hold the identical SHA `a98de5517c0e80180ed2a1ccd6795d47f155beb5`.
- Fourth, the trap. A mirror push is a replica, not a backup. Watch what happens when a branch is deleted upstream:

```
$ git -C up branch -D hotfix/urgent
Deleted branch hotfix/urgent (was d9a902a).

$ git -C mir.git remote update --prune
- [deleted] (none) -> hotfix/urgent

$ git -C mir.git push --mirror /tmp/mir/dest.git
To /tmp/mir/dest.git
- [deleted] hotfix/urgent
```

- The deletion was faithfully copied to the second server. A mirror reproduces mistakes exactly as well as it reproduces work. Keep that in mind for the backup section.

■ 40.10.4 PLAIN – what is really happening inside

- `git clone --mirror` does three things and nothing magical.
- It makes the new repository bare, so there is no working tree.
- It sets `remote.origin.mirror` to `true`.
- It sets the fetch refspec to `+refs/*:refs/*`, meaning “take every ref, of every kind, and store it under the same name, forcing where needed”.
- Compare that with the ordinary clone refspec, `+refs/heads/*:refs/remotes/origin/*`, which takes only branches and files them under a separate prefix.
- That single line of configuration is the entire difference between a copy you work in and a copy that stands in for the original.
- On the push side, `git push --mirror` sends every local ref and deletes every destination ref that you do not have. That is why the deletion propagated above.
- One consequence surprises everybody who mirrors a GitHub repository. `+refs/*:refs/*` includes GitHub’s pull request refs. Here is a real mirror clone of a public repository:

```
$ git clone --mirror https://github.com/octocat/Hello-World hw.git
$ git -C hw.git show-ref | wc -l
3439
$ git -C hw.git show-ref | grep -c refs/pull/
3436
```

- Three branches, and three thousand four hundred and thirty-six pull request refs. If you then push that mirror to a new host, you are trying to create three thousand refs the new host neither wants nor allows.
- The fix is to fetch or push only what you mean, using an explicit refspec such as `+refs/heads/*:refs/heads/*` and `+refs/tags/*:refs/tags/*`.

■ 40.10.5 TECHNICAL – the engineer’s version

- What actually travels, and what does not.

Thing	Inside git?	Copied by clone
Commits, trees, blobs	yes	yes
Branches	yes	yes, renamed

Thing	Inside git?	Copied by clone
Lightweight tags	yes	yes
Annotated tag objects	yes	yes
Notes in refs/notes	yes	only with refspec
Signatures on commits	yes	yes
Issues and comments	no	no
Pull requests, reviews	no	no
CI runs and logs	no	no
Releases and assets	no	no
Wiki content	separate repo	separate clone
Webhooks, secrets	no	no
Protection rules	no	no
Stars, forks, watchers	no	no

- The wiki row is worth knowing precisely: GitHub stores a repository wiki as a separate git repository, reachable at the repository address with `.wiki.git` on the end. So the wiki does migrate, but only if you deliberately clone it as a second repository.
- A correct migration of the git part, in full:

```
git clone --mirror https://github.com/you/proj.git
cd proj.git
git remote set-url --push origin git@newhost:you/proj.git
git push --prune origin \
    '+refs/heads/*:refs/heads/*' \
    '+refs/tags/*:refs/tags/'
```

- That is deliberate rather than clever. It moves branches and tags, prunes what is gone, and does not attempt to move `refs/pull/*`.
- Verify the move by comparing ref lists rather than by looking at a web page:

```
OLD=https://old/proj.git
NEW=git@newhost:you/proj.git
git ls-remote --heads --tags $OLD | sort > old.txt
git ls-remote --heads --tags $NEW | sort > new.txt
diff old.txt new.txt && echo "identical"
```

- Metadata is a separate project with separate tools, and no standard format exists. As of 2026 the realistic options are:

Source and target	Tool	What it carries
GitHub to Gitea, Forgejo	built-in migration	issues, PRs, labels
GitHub to GitLab	GitLab importer	issues, MRs, comments
GitLab to GitLab	project export file	most things
Anything to anything	the REST API	whatever you code

- For a do-it-yourself export the practical route is the API. The GitHub CLI can page through issues and write JSON, which you then store in the repository itself so it is versioned with the code:

```
gh api --paginate repos/OWNER/REPO/issues?state=all \
    > issues.json
gh api --paginate repos/OWNER/REPO/pulls?state=all \
    > pulls.json
```


8. GitHub also offers migration archives through its migrations API for organizations, producing a downloadable archive of repositories and associated metadata. Availability and contents depend on your plan, so check before you rely on it.
9. Now the honest warnings, which matter more than the commands.
10. Issue numbers will not survive. Every forge assigns its own numbering, so “fixes #42” in a commit message will point at the wrong thing after migration. The commit message is immutable, so this cannot be corrected without rewriting history.
11. Cross-references, mentions of users, and links between issues and pull requests are stored as identifiers that mean nothing on the new host.
12. Timestamps and authorship on imported issues usually become “imported by the migration account on the day of import” unless the importer has special support, because most APIs will not let you post as someone else.
13. Attachments in comments are files on the old host, served from the old host’s storage, and vanish when the repository does.
14. Build history and logs are effectively unmovable. Treat them as data with a retention period, not as a permanent record.
15. So the practical rule for anything you truly cannot lose: put it in the repository. A decision written in a Markdown file in docs/ migrates perfectly, forever, because it is a blob. The same decision written in an issue comment is the platform’s, not yours.

■ 40.10.6 WORDS – remember these

1. Mirror clone — a copy of every ref under its own name — `git clone --mirror`, which sets `remote.origin.mirror=true` and the `refspec+refs/*:refs/*` on a bare repository.
2. Mirror push — make the far side match me exactly — `git push --mirror`, which creates, updates and deletes refs to match the local set.
3. Prune — remove refs the other side no longer has — `--prune` on fetch, remote update or push.
4. Second push URL — one remote, two destinations — extra push URLs added with `git remote set-url --add --push`.
5. Migration — moving to another host — copying refs and objects with git, then copying metadata with each platform’s own tools, imperfectly.
6. Platform metadata — everything not in git — issues, reviews, runs, releases and rules, which have no interchange standard.

40.11 Working with others, practically

■ 40.11.1 PLAIN – in simple words

1. There are only two ways to arrange a team around a repository.
2. The shared repository model. Everyone can push to the same repository, and they push branches, not straight to main.
3. The fork model. Nobody outside the core team can push at all. Outsiders make their own server-side copy, push there, and ask for their work to be taken.
4. The shared model is for people you trust: a company team, a class group, a startup.
5. The fork model is for strangers: any public project where anyone in the world may send a change.
6. Both end at the same place, a pull request, so they look identical in the web interface. The difference is only in who may write where.
7. On top of that choice sits a second one: how branches are organized over time. There are three well-known answers, and they have real trade-offs.

8. And on top of that sits one rule everybody agrees on: protect main.

■ 40.11.2 PLAIN – a picture in your head

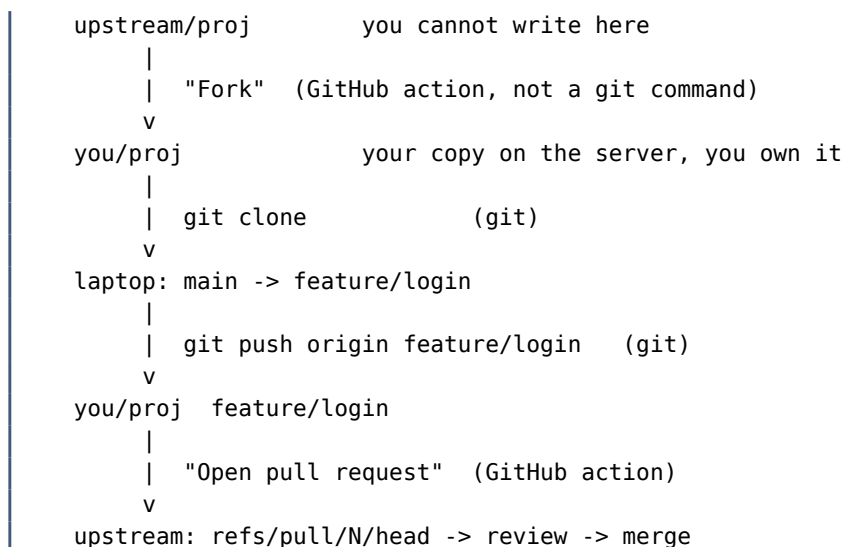
1. Think of a shared document at a college.
2. The shared repository model is a group project. Everyone has edit rights, but the agreement is that you work on your own page and ask before merging your page into the final report.
3. The fork model is a published book. You cannot write in the book. You make a photocopy, mark it up, and post your marked copy to the editor, who decides whether to take your change.
4. A publisher does not give edit rights to every reader. That would be absurd, and it is exactly why open-source projects use forks.
5. A group of four friends does not photocopy the report for each other. That would be pointless ceremony, and it is why company teams use the shared model.

Where this comparison breaks:

6. A photocopy goes out of date the moment the book changes. A fork does not have to. You can add the original as a second remote and pull from it, so your copy stays current.
7. And the editor in this picture merges by hand. A pull request merge is performed by the server, exactly, using git.

■ 40.11.3 PLAIN – a worked example

1. Here is the fork-and-pull-request path drawn out. Only two steps in it are git; the rest are the website.



2. In the shared repository model, the first two boxes disappear. You clone upstream/proj directly, push feature/login to it, and open the pull request from there.
3. Now branch names. Git's rules are stricter than people expect, and this is real output from git check-ref-format:

```

OK   feature/login
OK   feature/ABC-123-add-login
OK   release/1.0
OK   Feature/Login
BAD  feature/add login
BAD  feature//login
BAD  feature/login.lock
  
```

```
BAD feature/.hidden
BAD feature/login~1
BAD feature/login^
```

4. So a space is illegal, a double slash is illegal, a component starting with a dot is illegal, an ending of .lock is illegal, and the characters ~ ^ : ? * [\ are illegal.
5. Feature/Login is legal but a bad idea, because on macOS and Windows the filesystem usually ignores case and two branches differing only in case will collide.
6. And here is the trap that catches every team once:

```
$ git branch feature/login
$ git branch feature
fatal: cannot lock ref 'refs/heads/feature':
'refs/heads/feature/login' exists;
cannot create 'refs/heads/feature'
```

7. A loose ref is a file on disk, so feature cannot be both a file and a directory. Pick a prefix scheme and never use a bare prefix as a branch.
8. Finally, the three ways a pull request can land. All three of these are real graphs from the same starting point:

```
1. merge commit (--no-ff)
* 0685fe9 Merge pull request: login
|\
| * 00ee7d7 Fix login typo
| * 046ed60 Start login
* | ec53eed Unrelated work on main
|/
* cd18e97 Base commit

2. squash merge
* 23a1f88 Add login (#42)
* ec53eed Unrelated work on main
* cd18e97 Base commit

3. rebase then fast-forward
* 12a1676 Fix login typo
* 0ae1582 Start login
* ec53eed Unrelated work on main
* cd18e97 Base commit
```

9. Look at what each one costs. The merge commit keeps every step and the shape. The squash keeps one tidy commit and throws away the steps. The rebase keeps the steps and throws away the shape, and it made new SHAs: 046ed60 became 0ae1582.
10. That last point is the reader's own experience. Rebasing a local branch onto a moved main changed the branch's SHA, because rebase does not move commits. It creates new commit objects with new parents, and therefore new hashes.

■ 40.11.4 PLAIN - what is really happening inside

1. A fork, mechanically, is a repository on the server plus a database row saying "this came from that". Git has no idea it happened.
2. When you fork on GitHub, the new repository may share object storage with the original, which is why forking a huge repository is instant. That is an implementation detail of one product, not part of git.

3. A pull request from a fork works because both repositories are on the same server, so the server can read your head commit directly.
4. Now the three branching models, described by what they actually do to the graph.
5. Git flow, published by Vincent Driessen on 5 January 2010, keeps two permanent branches, `main` and `develop`, plus `feature/*`, `release/*` and `hotfix/*`. Work flows feature into `develop`, `develop` into a release branch, release into `main` and back into `develop`.
6. GitHub flow, published by Scott Chacon on 31 August 2011, keeps one permanent branch. Branch off `main`, open a pull request, get review, merge, deploy. There is no `develop` branch and no release branch.
7. Trunk-based development keeps one permanent branch and adds a rule about time: branches live hours, not days, and unfinished work is hidden behind feature flags rather than hidden on a branch.
8. Notice what the three actually disagree about. Not tooling. They disagree about how long a change may live away from the mainline.
9. Long-lived branches are the cost. Every day a branch lives, the mainline moves under it, and the merge gets harder. This is not an opinion; it is what a graph does when two lines diverge.
10. Protecting `main` is the mechanism that makes any of the three safe. A protected branch cannot be pushed to directly, cannot be force-pushed and cannot be deleted, and merges into it require whatever checks you set.
11. Without protection, all three models are just polite suggestions, exactly like a client-side hook.

■ 40.11.5 TECHNICAL – the engineer’s version

1. The two collaboration models compared on the things that actually differ.

Question	Shared repo	Fork model
Who may push branches	the team	only your fork
Write access needed	yes	no
Number of remotes	one	two
Suits	trusted teams	public projects
CI secrets exposure	trusted	needs care

2. The last row is a real security point. A workflow triggered by a pull request from a fork must not be given repository secrets, or any stranger could exfiltrate them. GitHub’s `pull_request` event withholds secrets from fork pull requests for exactly this reason, while `pull_request_target` runs with them and is dangerous if misused.
3. The three branching models, with honest trade-offs.

Model	Long-lived branches	Best for
Git flow	<code>main</code> and <code>develop</code>	versioned releases
GitHub flow	<code>main</code> only	continuous deploy
Trunk-based	<code>main</code> only	high-rate teams

4. Driessen added a note to his own 2010 article in March 2020 saying that if your team does continuous delivery you should adopt a much simpler workflow such as GitHub flow, and that git flow still fits software that is explicitly versioned or must support several versions in the wild.
5. That is the fairest summary available, and it comes from the author of the model people argue about.

6. The research position: the 2018 book *Accelerate*, by Nicole Forsgren, Jez Humble and Gene Kim, reports that high-performing teams have fewer than three active branches, branch lifetimes under a day, and rarely or never have code freezes.
7. Read that as a correlation found in survey data, not as a law. The critics' point is fair too: trunk-based development requires strong automated tests and feature flags, and a team without them will simply break main faster.
8. Recommended day-one setup for a small team, concretely. This is a working configuration, not a philosophy.
9. Set the default branch name once per machine, so nobody argues:

```
| git config --global init.defaultBranch main
```
10. That option arrived in Git 2.28, released 27 July 2020. GitHub changed the default branch for newly created repositories to `main` in October 2020. Older repositories were not renamed, which is why both names are still everywhere.
11. On the repository, turn on branch protection for `main` with these settings: require a pull request before merging, require at least one approving review, require status checks to pass, require branches to be up to date before merging, and block force pushes and deletions.
12. Choose one merge method and disable the others, so history has one shape. Squash merge is the usual choice for small teams because it makes `main` one commit per change and keeps `git log --oneline` readable.
13. Agree a branch prefix scheme and write it in the README. A workable one is `feature/`, `fix/`, `chore/`, `docs/`, plus the ticket identifier, giving names like `feature/ABC-123-add-login`.
14. Add a CODEOWNERS file even with three people. It costs two lines and it means nobody has to remember who reviews what.
15. Add one CI workflow that runs the tests on every pull request, and make it a required check. An unenforced test suite decays within weeks.
16. Turn on tag protection or use signed tags for anything you release, so a version number cannot be quietly repointed.
17. Verify CI by run identifier against the exact head SHA rather than by trusting a green tick beside a branch name. In the reader's own session that distinction was the difference between believing a result and knowing one.
18. Do not add: a `develop` branch, a release branch per version, or a rule that two people must approve, until you have a concrete reason. Each of those is a real cost paid every day.

■ 40.11.6 WORDS – remember these

1. Fork — your own server-side copy of a project — a forge-level repository copy plus a recorded parent relationship, invisible to git.
2. Upstream — the original project you forked from — conventionally a second remote named `upstream` alongside `origin`.
3. Shared repository model — everyone pushes branches to one repository — collaboration requiring write access, with policy enforced by protection rules.
4. Git flow — the two-permanent-branch model from 2010 — `main` plus `develop` with `feature/*`, `release/*` and `hotfix/*` branches.
5. GitHub flow — branch, pull request, merge, deploy — a single permanent branch model published in 2011.
6. Trunk-based development — branches measured in hours — mainline development with short-lived branches and feature flags instead of long branches.

7. Feature flag — a switch that hides unfinished work — a runtime condition allowing incomplete code to be merged but not executed.
8. Protected branch — main cannot be pushed to directly — a server-side rule blocking direct pushes, force pushes and deletion.
9. Squash merge — many commits become one — a merge that applies the combined change as a single new commit with one parent.
10. check-ref-format — the rules for legal ref names — `git check-ref-format`, which rejects spaces, `..`, `~`, `^`, `:`, `?`, `*`, `[`, trailing `.` lock and leading dots.

40.12 Monorepos and large repositories

■ 40.12.1 PLAIN - in simple words

1. A monorepo is one repository holding many projects that could have been separate repositories.
2. Big companies use them for one reason above all others: one commit can change a library and every user of that library at the same time.
3. That removes the hardest problem in large systems, which is not writing code but keeping many versions of many pieces compatible.
4. You also get one place to search, one build configuration, and no arguing about which repository a change belongs in.
5. The price is size, and size hurts git in specific, measurable ways.
6. Clones get slow, because a clone copies the whole history by default.
7. Everyday commands get slow, because git checks every file it tracks.
8. Tools get slow or give up, because editors and search programs were not written with a million files in mind.
9. Git has four answers, and you can combine them: shallow clone, partial clone, sparse checkout, and Git LFS for large files.
10. And one weakness has no good answer: binary files. Git stores them, but it stores them badly, and this section will show exactly how badly.

■ 40.12.2 PLAIN - a picture in your head

1. Think of a library that keeps every edition of every book it has ever held.
2. If you want to borrow one novel, the librarian hands you the entire archive on a trolley, including every draft of every book, and says “it is all yours now, take it home”.
3. That is `git clone`. It is wonderful once you have it and painful the first time.
4. A shallow clone is asking for only the newest edition of everything.
5. A partial clone is asking for the catalogue now and the actual pages later, only when you open a book.
6. A sparse checkout is taking the whole archive home but only unpacking the one shelf you work on.
7. Git LFS is the library keeping the heavy art books in a warehouse and giving you a slip of paper saying where each one is.

Where this comparison breaks:

8. Fetching a page later feels free in a library. In a partial clone it is a network round trip, and if you ask for thousands of pages one at a time it is dramatically slower than having taken the trolley. There are real numbers for this in the worked example, and they are worse than most people guess.

■ 40.12.3 PLAIN – a worked example

1. Every number here was measured for this chapter against one real public repository, the GitHub CLI source, which has 11,772 commits and 1,356 tracked files. It is a medium repository, not a giant one.
2. First, the three ways to clone it.

Clone type	.git size	Time	Commits
Full clone	81 MB	5.9 s	11,772
Blobless, blob:none	27 MB	3.4 s	11,772
Shallow, depth 1	15 MB	2.3 s	1

3. Read the middle row carefully. The blobless clone still has every commit and every tree. It has 11,772 commits, same as the full clone. What it does not have is old file contents.
4. That is usually the right trade, because history browsing needs commits, and commits are small. File contents are what is big.
5. Object counts make the same point:

```
full clone    in-pack 90,539    size-pack 79.75 MiB
blobless     in-pack 64,891    size-pack 25.66 MiB
shallow 1    in-pack 1,776    size-pack 14.58 MiB
```

6. Second, sparse checkout, which is about the working tree rather than the history. This checks out one directory only:

```
$ git sparse-checkout set --cone pkg/cmd/pr
$ git ls-files | wc -l
1356
$ find . -path ./.git -prune -o -type f -print | wc -l
115
```

7. Git still tracks all 1,356 files. Only 115 exist on disk. The working tree shrank from 27 MB to 1.4 MB.
8. The index shows how it is done. 1,241 entries are marked skip-worktree and 115 are present.
9. Third, the honest cost of a partial clone. The same command in both:

Command	Full clone	Blobless clone
git log -p -300	0.197 s	106 s

10. That is not a typo. Reading the patch text of the last 300 commits took a fifth of a second with all objects present, and one minute forty-six seconds when each missing blob had to be fetched over the network.
11. The .git directory grew from 27 MB to 29 MB during that command, and 51 extra pack files appeared, each marked .promisor, meaning “fetched lazily from the promising remote”.
12. So partial clone is not free speed. It moves cost from clone time to the first command that needs old content.
13. Fourth, Git LFS. Here is what git actually stores when a 1 MiB binary file is tracked by LFS:

```
$ git cat-file -p HEAD:blob.bin
version https://git-lfs.github.com/spec/v1
oid sha256:5e20a633da26ff9bceb11f08d4671c234
```



```
38c49362cefce5042e8c77bb58bbdf9
size 1048576

$ git cat-file -s HEAD:blob.bin
132
```

14. The object in git is 132 bytes of text. The real megabyte lives outside the object database, under `.git/lfs/objects`, and is fetched by a separate program over a separate protocol.
15. Note the oid line. It is a SHA-256 of the file contents, and it matches what `sha256sum` reports for the real file. The pointer is content-addressed, just like git itself.
16. Fifth, why binaries are git's weak point. Two repositories, ten commits each, both starting from nothing:

Repository, 10 revisions	Packed size
1 MiB random binary	10.01 MiB
2.1 MB text, tiny edits	148.27 KiB

17. Ten revisions of a one-megabyte binary cost ten megabytes. Ten revisions of a two-megabyte text file cost 148 kilobytes.
18. The text repository is fourteen times smaller while holding twice as much content per revision. That gap is the whole story of binaries in git.

■ 40.12.4 PLAIN – what is really happening inside

1. Git compresses history in two stages: `zlib` on each object, then delta compression, where one object is stored as a small set of instructions for turning another object into it.
2. Delta compression is why a text repository stays tiny. Change one line in a 60,000-line file and the delta is a few dozen bytes.
3. A compressed or random binary file has no useful similarity to the previous version. Change one pixel in a PNG and the compressed bytes change everywhere. The delta is as big as the file, so git stores a whole new copy.
4. That is not a bug and no setting fixes it. It is what content-addressed storage plus general-purpose compression can do.
5. Worse, git never forgets. Delete the 500 MB file today and it is still in every clone forever, because it is still reachable from an old commit.
6. Git LFS works around this by keeping a tiny text pointer in the repository and the real bytes elsewhere. A `clean` filter converts the file to a pointer on `git add`, and a `smudge` filter converts it back on checkout.
7. So the history contains only 132-byte pointers, which delta beautifully, and the big files are transferred by a separate service that fetches only the versions you actually check out.
8. The four scaling techniques attack four different costs, and it is worth being exact about which is which.
9. Shallow clone cuts the number of commits you download. It is the right tool for a build machine that only needs the newest source.
10. Partial clone cuts the number of objects you download, keeping the full commit graph. It is the right tool for a developer who needs history but not every old file.
11. Sparse checkout cuts the number of files written to disk. It does not reduce download size at all, but it makes `git status`, editors and search tools fast again.

12. Git LFS cuts what enters the history in the first place. It is the only one of the four that changes what is committed.
13. And the index is the quiet cost nobody expects. It is one file listing every tracked path with its metadata. For 1,356 files it was 156,135 bytes. Scale that to a million files and the index alone is over one hundred megabytes, read and written by ordinary commands.

■ 40.12.5 TECHNICAL – the engineer’s version

1. Real monorepos, with their published numbers.

Repository	Figure	Source year
Google, all code	86 TB, 2bn lines	2016 paper
Google, source files	~9 million unique	2016 paper
Google, commits	~35 million	2016 paper
Windows in git	~300 GB, 3.5m files	2017 post
Windows pushes	8,421 per day	2017 post

2. The Google figures come from the paper “Why Google Stores Billions of Lines of Code in a Single Repository” by Rachel Potvin and Josh Levenberg, published in Communications of the ACM in July 2016 and describing a January 2015 snapshot. It also reports roughly 25,000 developers, about 16,000 human commits and about 24,000 automated commits on a typical workday.
3. Important honesty point that most retellings omit: Google’s monorepo does not run on git. It runs on Piper, an in-house system, with a virtual filesystem client. So it is evidence that monorepos work at scale, not evidence that git works at that scale.
4. The Windows figures come from Brian Harry’s post of 24 May 2017, “The largest Git repo on the planet”, describing about 4,000 engineers, 1,760 official builds per day across 440 branches, and the Git Virtual File System that Microsoft had to build to make it possible.
5. GVFS was announced in February 2017, later renamed VFS for Git, and the ideas were pushed upstream over the following years as partial clone, sparse checkout in cone mode, the commit-graph file and the built-in filesystem monitor.
6. Version history of the relevant features, so you know what your git can do.

Capability	Git version	Date
Shallow clone, <code>--depth</code>	1.5 era	2007 onward
commit-graph file	2.18	Jun 2018
Partial clone, <code>--filter</code>	2.19 era	Sep 2018
git sparse-checkout	2.25	13 Jan 2020
Built-in fsmonitor	2.37	27 Jun 2022

7. Partial clone filters worth knowing: `--filter=blob:none` omits all blobs; `--filter=blob:limit=1m` omits blobs above a size; `--filter=tree:0` omits trees as well and is meant for tooling, not for humans.
8. A partial clone records the remote as a promisor. Objects arrive lazily, and lazily fetched packs are marked with a `.promisor` file. In the measurement above, 51 such packs appeared during one `git log -p`.
9. The server must permit it. `uploadpack.allowFilter` must be true on the serving side, or `--filter` is silently ignored and you get a full clone.

10. Sparse checkout in cone mode restricts patterns to whole directories, which lets git match paths with prefix comparisons instead of running every pattern against every path. Non-cone mode accepts full gitignore-style patterns and is much slower on large trees.
11. Practical settings for a large repository, all real configuration keys:

```
git config core.fsmonitor true
git config core.untrackedCache true
git config feature.manyFiles true
git maintenance start
```
12. `feature.manyFiles` is an umbrella that sets `index version 4` and `core.untrackedCache`. `git maintenance start` schedules background repacking and commit-graph writing rather than waiting for `git gc` to interrupt you.
13. Git LFS, released by GitHub in April 2015, is not part of git. It is a separate program plus a separate HTTP API. `git lfs track "*.psd"` writes a line to `.gitattributes` naming the clean, smudge and merge filters.
14. Consequences people discover late: anyone cloning your repository needs git-lfs installed or they get pointer files instead of content; LFS storage and bandwidth are billed separately by most hosts; and converting an existing repository to LFS rewrites history, which changes every SHA.
15. Alternatives worth naming: `git-annex`, which predates LFS and is more flexible; and simply keeping large assets out of the repository entirely, in object storage referenced by a versioned manifest file.
16. Removing a large file that is already in history requires rewriting it out. The current recommended tool is `git filter-repo`; the older `git filter-branch` is deprecated for being slow and easy to misuse. Both change every commit hash after the touched point, so every collaborator must re-clone.

■ 40.12.6 WORDS - remember these

1. Monorepo — one repository for many projects — a single version-controlled tree containing multiple deliverables with one commit history.
2. Shallow clone — only the newest commits — `--depth N`, producing a truncated history with grafted boundary commits.
3. Partial clone — download objects on demand — `--filter=blob:none` or similar, with the origin recorded as a promisor remote.
4. Promisor remote — the server that owes you objects — the remote a partial clone will lazily fetch missing objects from.
5. Sparse checkout — only some files on disk — index entries marked `skip-worktree` so git neither writes nor scans them.
6. Cone mode — directory-shaped sparse patterns — the fast form of sparse checkout, matching whole directories by prefix.
7. Delta compression — store the difference, not the file — packfile encoding of one object as instructions against a similar base object.
8. Git LFS — big files kept outside history — a filter that commits a small pointer file and stores real content on a separate service.
9. Pointer file — the 130-odd bytes git actually stores — a three-line text blob giving spec version, a SHA-256 oid and a byte size.
10. `filter-repo` — the supported way to rewrite history — the recommended replacement for `git filter-branch`, which rewrites every affected commit and therefore every hash.

40.13 Backing up a git repository properly

■ 40.13.1 PLAIN – in simple words

1. People say “every clone is a backup”. That is half true, and the missing half is the half that hurts.
2. A clone gives you every commit reachable from the branches you cloned, and the tags. That really is a lot, and it really does survive the server burning down.
3. A clone does not give you other refs, the reflog, stashes, the repository’s own configuration, its hooks, or anybody’s uncommitted work.
4. A mirror gives you every ref, of every kind, under its original name. That is more than a clone.
5. A mirror still does not give you reflogs, hooks or configuration, and it has a specific danger: it copies deletions too.
6. Neither gives you a single thing that lives on the website: issues, pull requests, reviews, build history, releases, secrets or settings.
7. And neither, on its own, protects you from the most common real disaster, which is not hardware failure. It is somebody force-pushing over a branch, or deleting one, and the automatic mirror faithfully copying that within the hour.
8. A real backup therefore has three properties a mirror does not have: it is dated, it is read-only once written, and it is kept for a while.

■ 40.13.2 PLAIN – a picture in your head

1. Imagine three ways of protecting a handwritten manuscript.
2. A friend borrowed a copy last month. That is a clone. It is genuinely useful, it is complete up to when they took it, and it is out of date.
3. A machine in the next room copies your desk every hour, exactly. That is a mirror. It is always current, and if you spill ink on page 40, the machine copies the ink stain within the hour.
4. A sealed, dated envelope goes into a fire safe every night and stays there for ninety days. That is a backup. It cannot be edited, it can be opened at any date you choose, and the ink stain does not reach it.
5. Most teams have the first two and call it done.

Where this comparison breaks:

6. A copy of a manuscript can be a slightly bad copy. A git clone cannot. Every object is verified against its hash, so a corrupted copy is detectable and usually refuses to load rather than lying to you.
7. And the friend with the month-old copy is genuinely enough to save the project, which is not true of a month-old photocopy of an edited book. Git history is append-only in normal use, so old copies stay valid.

■ 40.13.3 PLAIN – a worked example

1. First, what a clone leaves behind. A real source repository was set up with a stash, extra reflog entries, untracked files, a hook and a local setting, then cloned normally:

source repository	clone
stashes: 1	stashes: 0
reflog entries: 3	reflog entries: 1
untracked files: 2	untracked files: 0
custom hooks: 1	custom hooks: 0
local user.signingkey	(absent)
commits: 2	commits: 2

2. The commits came across perfectly. Everything else did not.
3. Second, what a mirror adds. The same source, cloned with `--mirror`:

```
$ git -C mtest.git show-ref
dbcd961d... refs/heads/main
f4caa812... refs/stash
141f4579... refs/tags/backup-2026-08-13

$ ls mtest.git/hooks | grep -v sample | wc -l
0
$ git -C mtest.git reflog | wc -l
0
```

4. Look at the second line of `show-ref`. The mirror picked up `refs/stash`, because its refspec is `+refs/*:refs/*` and a stash is a ref. A plain clone never does that.
5. The hooks did not come. The `reflog` did not come. The source's local configuration did not come.
6. Third, a real backup run, start to finish. This is the whole procedure and every line below is genuine output.

```
$ git -C repo.git remote update --prune

$ git -C repo.git fsck --full
dangling commit de306ffa47620fa7e0d5ef7146...

$ git -C repo.git bundle create vault/proj-2026-08-13.bundle --all

$ ls -l vault/
1045 proj-2026-08-13.bundle

$ git -C repo.git bundle verify vault/proj-2026-08-13.bundle
vault/proj-2026-08-13.bundle is okay
The bundle contains these 4 refs:
a3bfdce1... refs/heads/feature/login
2304fd03... refs/heads/main
fb32611e... refs/tags/v1.0
2304fd03... HEAD
The bundle records a complete history.
The bundle uses this hash algorithm: sha1
```

7. That bundle is one ordinary file. You can copy it anywhere, put it on any storage, and it needs no git server to be useful.
8. `dangling commit` in the `fsck` output is normal, not an error. It means an object exists that no ref points at any more, which happens after a rebase or an amended commit.
9. Fourth, and this is the step almost everyone skips, the restore drill:

```
$ git clone vault/proj-2026-08-13.bundle restore
$ git -C restore log --oneline | head -3
2304fd0 Add app.py
20b8927 First commit: add README
$ git -C restore rev-list --count --all
3
```

10. It restored. Now you know. A backup you have never restored is not a backup; it is a hope.

■ 40.13.4 PLAIN – what is really happening inside

1. Git keeps an object only while something can reach it. Refs are the roots. Anything reachable from a ref is kept; anything not is eventually removed by garbage collection.

2. That single rule explains the whole backup problem.
3. When you clone, git asks for the refs it wants and receives everything reachable from them. Objects reachable only from refs you did not ask for simply do not come.
4. A plain clone asks for refs/heads/* and tags. So refs/stash, refs/notes/* and any custom ref namespace are left behind.
5. A mirror asks for refs/*, so it gets all of them.
6. Neither asks for the reflog, because the reflog is not a ref. It is a local log file under .git/logs, private to one repository, recording where each ref used to point.
7. That matters because the reflog is what saves you from your own mistakes. It is the thing that lets you recover a commit after a bad reset. It is also the thing no backup of the server can give you.
8. Now the deletion problem. Suppose somebody force-pushes over main, discarding fifty commits. On the server the old commits become unreachable.
9. Your hourly mirror runs, sees main moved, moves its own main, and now the mirror is unreachable from those commits too. Run garbage collection on the mirror and they are gone for good.
10. A dated bundle written before the force-push is unaffected, because it is a frozen file, not a live replica. This is the reason bundles exist in a backup plan alongside mirrors.
11. What a bundle actually is: a packfile with a small header listing ref names and, for an incremental bundle, the commits the receiver must already have. `git bundle verify` checks both the pack and those prerequisites.
12. And a bundle is a first-class remote. `git clone file.bundle`, `git fetch file.bundle`, and `git ls-remote file.bundle` all work, which is why the restore drill above is one ordinary clone.

■ 40.13.5 TECHNICAL – the engineer’s version

1. Coverage table. This is the section in one picture.

Item	Clone	Mirror	Bundle –all
Branch commits	yes	yes	yes
Tags	yes	yes	yes
refs/stash	no	yes	yes
refs/notes	no	yes	yes
Custom ref namespaces	no	yes	yes
Reflog	no	no	no
Hooks	no	no	no
Repository config	no	no	no
Uncommitted work	no	no	no
Issues, PRs, CI, releases	no	no	no
Survives a force-push	no	no	yes, if dated

2. Incremental bundles work and are small. A real example: the full bundle was 1,497 bytes and the bundle of everything since a tag was 703 bytes.

```
$ git bundle create incr.bundle main --not backup-2026-08-13
$ git bundle verify incr.bundle
incr.bundle is okay
The bundle contains this ref:
dbcd961d... refs/heads/main
The bundle requires this ref:
141f4579...
```

3. “Requires this ref” is the prerequisite. Applying it to a repository that already has that commit works; applying it to an empty one fails cleanly rather than silently producing a broken repository.

```
$ git fetch /tmp/bk/incr.bundle main:restored
$ git log --oneline restored
dbcd961 Fourth
9cbb377 Third
141f457 Second
69199b0 First
```

4. A backup procedure that actually holds up. Run it from a machine that is not the git server.

```
# once
git clone --mirror git@host:org/proj.git /srv/bk/proj.git

# every hour: refresh the replica
git -C /srv/bk/proj.git remote update --prune

# every night: dated, immutable snapshot
D=$(date +%F)
git -C /srv/bk/proj.git fsck --full --strict
git -C /srv/bk/proj.git bundle create \
  /srv/vault/proj-$D.bundle --all
git -C /srv/bk/proj.git show-ref > /srv/vault/proj-$D.refs
git -C /srv/bk/proj.git bundle verify \
  /srv/vault/proj-$D.bundle
```

5. Then copy /srv/vault off-site, keep snapshots for a stated retention period, and delete older ones on a schedule you wrote down.
6. Why each step is there:

Step	Protects against
Mirror refresh	server loss
fsck --full	silent corruption
Dated bundle	force-push, deletion
show-ref file	knowing what was there
bundle verify	an unreadable backup
Off-site copy	site-wide loss

7. The show-ref file is cheap and disproportionately useful. When somebody asks “what was main pointing at on the ninth?”, it answers in one line without unpacking anything.
8. The 3-2-1 convention is worth applying here: three copies, on two different kinds of storage, one of them off-site. It is a convention from the photography world, not a standard, but it is sound.
9. Verification is not optional and is cheap in git, because every object is named by the hash of its own contents. `git fsck --full --strict` recomputes those hashes and reports any mismatch. No other backup system gets this for free.
10. Restore drills should be scheduled, not improvised. Once a quarter, clone the newest bundle into a scratch directory, compare `git rev-parse main` with the recorded `.refs` file, and delete it.
11. Now the part git cannot help with. Back up the platform metadata separately, because none of it is in any bundle:

```
gh api --paginate repos/OWNER/REPO/issues?state=all \
  > vault/issues-$D.json
gh api --paginate repos/OWNER/REPO/pulls?state=all \
```



```

    > vault/pulls-$D.json
gh api repos/OWNER/REPO/branches/main/protection \
    > vault/protection-$D.json
gh release list --repo OWNER/REPO \
    > vault/releases-$D.txt

```

12. Secrets cannot be exported at all by design. Keep the authoritative copy in a password manager or a secrets service, and treat the platform's copy as a deployment detail, not as storage.
13. Settings are best kept as code. Branch protection, teams and repository options can be declared in a configuration tool and re-applied to a fresh repository in minutes, which is a better recovery story than a JSON dump nobody knows how to replay.
14. Encryption note: a bundle is not encrypted and not signed. If your history is confidential, encrypt the file before it leaves your control, and record the checksum separately so you can prove the file is intact.
15. Retention worth defaulting to, in the absence of a policy: hourly mirror, nightly bundle kept 30 days, weekly bundle kept 12 weeks, monthly bundle kept 12 months. Adjust for how much you would pay to recover a given day.

■ 40.13.6 WORDS - remember these

1. Reachability — an object is kept if a ref can reach it — the rule that decides what a clone transfers and what garbage collection removes.
2. Dangling object — nothing points at it any more — an object with no ref path to it, reported by `git fsck`, normal after rebase or amend.
3. Bundle — a whole repository in one file — a packfile plus a ref header, usable directly as a remote for clone, fetch and `ls-remote`.
4. Prerequisite — what an incremental bundle assumes you have — commit identifiers listed in the bundle header and checked by `bundle verify`.
5. Replica — always current, always obedient — a mirror that reproduces deletions and force-pushes as faithfully as it reproduces work.
6. Snapshot — frozen and dated — an immutable copy taken at a known time and kept for a stated retention period.
7. Restore drill — proving the backup works — a scheduled rehearsal that clones the newest snapshot and compares refs against a recorded list.
8. `fsck` — check every object against its own hash — `git fsck --full --strict`, which detects corruption that ordinary commands might not notice.

40.98 Common wrong ideas

1. Wrong: GitHub stores my code differently from my laptop. Right: it stores the identical objects with the identical hashes. A commit is `2304fd03...` in both places, or one of them is corrupt. GitHub's replication and routing are invisible to the protocol.
2. Wrong: a pull request is a git feature. Right: git has no such concept. `git help -a` lists nothing called pull request. A pull request is a row in a forge's database plus two ref names, exposed to git only as read-only `refs/pull/N/head`.
3. Wrong: forking is a git operation. Right: forking is a forge action. Git's word for taking a copy is `clone`. A fork is a server-side repository plus a recorded parent relationship that git never sees.
4. Wrong: if GitHub goes away my history is lost. Right: every clone holds the complete commit history. What would be lost is the meeting point and the metadata: issues, reviews, build results and releases. Back those up separately, because git cannot.

5. Wrong: a git server needs special server software. Right: it needs a bare repository and a way to reach it. `git init --bare` plus an SSH account with `git-shell` is a complete, secure server.
6. Wrong: a mirror is a backup. Right: a mirror copies deletions and force-pushes too. It protects against the server dying and not against a person making a mistake. Dated bundles protect against the person.
7. Wrong: a clone copies everything in the repository. Right: it copies objects reachable from branches and tags. It leaves behind the reflog, stashes, hooks, local configuration and all uncommitted work.
8. Wrong: `git push` failing means the push did not happen. Right: a failure while reading the reply, such as `send-pack: unexpected disconnect while reading sideband packet`, tells you the answer was lost, not what the server did. Re-query with `git ls-remote` to find out.
9. Wrong: LFS makes big binary files cheap in git. Right: it moves them out of git. The repository holds a 132-byte pointer, and the bytes live on a separate service with separate storage, separate billing and a separate program that everyone must install.
10. Wrong: a partial or shallow clone is strictly better because it is smaller. Right: it moves cost, it does not remove it. The same `git log -p -300` took 0.197 seconds in a full clone and 106 seconds in a blobless clone, because every missing blob became a network round trip.

40.99 Chapter summary in 20 lines

1. A git server is just another git repository that other machines can reach; there is no special server format and no central authority.
2. The meeting-point copy is normally bare, meaning it has the contents of `.git` at its top level and no working files.
3. `git init --bare proj.git` builds one in about ten milliseconds, and pushing into a non-bare repository's checked-out branch is refused by default.
4. Running the whole loop by hand proves the model: create the bare repository, push, clone elsewhere, commit there, push back, pull into the first copy.
5. A remote is a name for a URL plus a refspec, and a URL is only a path plus a way of reaching it, so `/tmp/lab/srv/proj.git` and `git@host:proj.git` are the same idea.
6. The refspec `+refs/heads/*:refs/remotes/origin/*` says: take every branch, store it under a separate prefix, and allow non-fast-forward updates.
7. Server hooks are real enforcement: `pre-receive` gates the whole push, `update` gates one ref, `post-receive` acts afterwards and cannot refuse.
8. Client hooks are advice, because `--no-verify` skips them, they are not versioned, and a non-executable hook file is silently ignored.
9. GitHub is four layers: hosting for bare repositories, a web interface, identity and permissions, and automation. Founded 2008, bought by Microsoft in 2018 for 7.5 billion United States dollars.
10. The git layer of GitHub is ordinary git; `git ls-remote` against it prints exactly the same shape of output as against a folder in `/tmp`.
11. Pull requests, reviews, branch protection, required checks, CODEOWNERS behaviour, issues, projects, releases, packages, Actions, both APIs and the permission model are all GitHub, not git.
12. The reader's outage proves the boundary: local git kept working, remote git and the whole GitHub API did not, exactly as the local/remote split predicts.
13. `origin/main` reading `0a95cc8` during the outage was a cached photograph, not a live fact, and a failure while reading a reply says nothing about whether the write landed.
14. Self-hosting is real and cheap: SSH plus `git-shell` costs tens of megabytes, Gitea or Forgejo runs in 1 GB, GitLab CE wants 8 to 16 GB.

15. Self-hosting buys control, privacy and no lock-in, and it makes you the operator, responsible for backups, uptime and patches at three in the morning.
16. Migration moves commits perfectly and almost nothing else; issue numbers, cross-references, timestamps, attachments and build history do not survive.
17. Choose the collaboration model by trust: shared repository for a team you trust, forks for strangers; then protect main, or every rule is a suggestion.
18. Git flow, GitHub flow and trunk-based development disagree about exactly one thing: how long a change may live away from the mainline.
19. Large repositories are managed with shallow clones, partial clones, sparse checkout and LFS, and binaries remain git's weak point because they do not delta-compress.
20. A clone is not a backup, a mirror is not a backup, and a backup you have never restored is not a backup either.

CI/CD and What a “Run” Really Is

41.0 What this chapter gives you

1. You will be able to explain what continuous integration is as a practice, and say clearly why it is not the same thing as a tool that runs tests.
2. You will be able to state the difference between continuous delivery and continuous deployment in one sentence each, without muddling them.
3. You will be able to describe what a runner is, what it has installed when it starts, and why a command that works on your laptop fails there.
4. You will be able to read a complete GitHub Actions workflow file line by line, and translate the same pipeline into GitLab CI.
5. You will be able to name every common trigger, and explain why `pull_request_target` is a security trap while `pull_request` is not.
6. You will be able to explain the single most important idea here: a run is attached to one commit hash, never to a branch name.
7. You will be able to verify continuous integration by run ID against an exact head hash, using real commands, and list six ways a branch-level green tick can lie to you.
8. You will be able to explain what a merge queue is, what problem it solves, and what every `mergeStatus` value means and implies.
9. You will be able to lay out a sensible pipeline in order, choose a deployment strategy, and handle secrets without leaking them.
10. You will be able to read a failed run methodically instead of guessing.

41.1 What continuous integration means and the problem it solves

■ 41.1.1 PLAIN – in simple words

1. Several people work on the same program at the same time.
2. Each person has their own copy, and each changes different files.
3. Sooner or later all those changes must be joined into one program.
4. Joining them is called **integration**. It is the moment of truth.
5. If everyone waits weeks before joining, the changes have drifted far apart and joining them is slow, painful and full of surprises.
6. Old teams had a word for this: integration hell. It could take days.
7. **Continuous integration** is the opposite habit. Everyone joins their work into the shared copy very often, at least once a day.
8. Small joins are easy. Big joins are hard. So you make the joins small.
9. But joining often is only safe if you also check often.

10. So every time anyone joins, a machine automatically builds the program and runs the tests, and tells you within minutes if it broke.
11. That habit, plus that automatic check, is continuous integration.
12. Notice that the habit comes first. The machine only enforces the habit.

■ 41.1.2 PLAIN – a picture in your head

1. Five students must write one report together, one chapter each.
2. Plan A: everyone writes alone for three weeks, then meets the night before to glue the chapters together.
3. That night is chaos. Two people defined the same term differently. Three people numbered their figures from one. The introduction promises a section that nobody wrote.
4. Plan B: every evening, each student pastes what they have into the one shared document, and one person reads the whole thing top to bottom.
5. Under plan B a clash is found the same evening it is created, while both students still remember what they meant.
6. Now replace “one person reads it top to bottom” with a machine that reads it every single time anyone pastes anything, in two minutes.
7. That machine is the continuous integration server. The evening paste is the practice.

Where this comparison breaks: a report can be a bit inconsistent and still be readable. A program cannot. One mismatched function name and nothing runs at all. Also, a human reader gets bored and misses things on the tenth read. A test suite does not get bored, but it only checks what somebody wrote a test for, which is always less than everything.

■ 41.1.3 PLAIN – a worked example

1. Take a team of five people. Each writes about 200 changed lines a day.
2. Under plan A they integrate once every three weeks, so 15 working days.
3. Lines waiting to be joined: 5 people $\times$ 200 lines $\times$ 15 days = 15,000 lines.
4. Under plan B they integrate once a day: 5 $\times$ 200 $\times$ 1 = 1,000 lines.
5. The pain of a merge does not grow in a straight line with size. Two changes can clash, and the number of possible clashing pairs grows roughly with the square of the number of changes.
6. So a batch fifteen times bigger can feel far worse than fifteen times worse.

Integration gap	Lines in flight	Typical feel
Every 3 weeks	15,000	Days of merge work
Every week	5,000	Hours of merge work
Every day	1,000	Minutes, usually clean
Every commit	~200	Almost always clean

7. The numbers in that table are illustrative, not measured. The shape is what matters: smaller batches, cheaper joins.

■ 41.1.4 PLAIN – what is really happening inside

1. You finish a small piece of work and commit it locally.
2. You push it to the shared server. The push is just data transfer.
3. The server notices that a branch moved and fires an event.

4. That event is delivered to whatever is listening. On a hosted service it goes to an internal queue. For an external tool it goes out as a **webhook**, which is a small message posted to a web address you registered.
5. The listener picks a free machine, or starts a fresh one.
6. That machine downloads the exact commit that was pushed. Not the branch. The commit.
7. It installs the tools the project needs, builds the program, and runs the tests.
8. It records success or failure, and posts that result back, attached to the same commit hash it downloaded.
9. The web page you look at then draws a tick or a cross next to that commit.
10. The whole loop is usually two to twenty minutes. Under ten minutes is the target most teams aim at, because longer than that and people stop waiting for the answer and start guessing.

■ 41.1.5 TECHNICAL – the engineer’s version

1. The phrase “continuous integration” was coined by Grady Booch in his 1991 book Object-Oriented Design With Applications, where it described the habit of integrating incrementally rather than in a big bang.
2. Extreme Programming, developed by Kent Beck on the Chrysler C3 project from 1996 and set out in Extreme Programming Explained in 1999, made it one of its named practices, with the rule that the team integrates and builds many times a day.
3. Martin Fowler and Matthew Foemmel published the article Continuous Integration in 2000, revised by Fowler in 2006 and again in 2024. That article is where most of the working definition comes from.
4. CruiseControl, written at ThoughtWorks and first released on 30 March 2001, was the first widely used continuous integration server. It polled the version control system on a timer, because webhooks did not exist yet.
5. Hudson was started by Kohsuke Kawaguchi at Sun Microsystems in 2004 and first released in 2005. After a dispute with Oracle, a rename vote was called on 11 January 2011 and approved on 29 January 2011. The renamed project is Jenkins.

Year	Thing	Who
1991	Term coined	Grady Booch
1999	XP names the practice	Kent Beck
2000	The defining article	Fowler and Foemmel
2001	CruiseControl released	ThoughtWorks
2011	Hudson renamed Jenkins	Community vote
2019	GitHub Actions CI GA	GitHub

6. Travis CI launched in 2011 and popularized the idea of a build configuration file living inside the repository. CircleCI launched the same year. GitLab CI began as a separate application in 2012 and was folded into GitLab itself in version 8.0 in September 2015.
7. GitHub Actions was announced as a general automation product in October 2018. Its continuous integration capability was announced in beta on 8 August 2019 and became generally available on 13 November 2019.
8. The practice, in its strict form, has three testable properties: everyone merges to the mainline at least daily, every merge triggers an automated build and test, and a broken mainline is fixed immediately and takes priority over new work.

9. Say this plainly: a team running Jenkins on a branch that merges to main once a quarter is not doing continuous integration. It owns a continuous integration tool. The practice and the tool are different things.
10. Observing tools: `gh run list`, `gh run view`, `git log --oneline --since='1 day ago' origin/main` to see integration frequency, and any dashboard that plots merges to trunk per day.

■ 41.1.6 WORDS - remember these

1. Integration — joining everyone’s work into one copy — merging feature branches into the mainline and producing a buildable tree.
2. Integration hell — the painful big join at the end — the superlinear cost of merging long-lived divergent branches.
3. Continuous integration — join often, check automatically every time — the practice of at-least-daily mainline merges verified by an automated build.
4. Webhook — a message the server sends when something happens — an HTTP POST to a registered endpoint carrying a JSON event payload.
5. Mainline or trunk — the one branch everybody joins into — usually main, the integration target for all short-lived branches.
6. Build — turning source into a runnable thing — compile, link, package, including dependency resolution.

41.2 Continuous delivery and continuous deployment

■ 41.2.1 PLAIN - in simple words

1. Continuous integration answers one question: does the joined code work.
2. There is a second question: could we give it to users right now.
3. **Continuous delivery** means the answer is always yes. Every change that passes the checks is packaged, tested in a realistic place, and left in a state where one button press would put it in front of users.
4. A human still decides when to press the button.
5. **Continuous deployment** goes one step further. There is no button. If the checks pass, the change goes live automatically.
6. So the difference is exactly one thing: whether a human approves the last step.
7. Continuous delivery is a capability. Continuous deployment is a policy you can only adopt once you have that capability.
8. Both are abbreviated CD, which is why people confuse them constantly.
9. If somebody says “we do CI/CD”, you have learned almost nothing. Ask which D they mean, and ask what the last manual step is.

■ 41.2.2 PLAIN - a picture in your head

1. Think of a bakery that supplies a shop.
2. Continuous integration is the taste test after mixing. Is the dough right.
3. Continuous delivery is: the bread is baked, wrapped, labelled, loaded onto the van, and the van is parked outside with the engine running.
4. Any moment the manager says go, it leaves. Nothing more to prepare.
5. Continuous deployment is: the van leaves the moment the loading is finished. No manager, no decision.
6. Notice that a bakery can be excellent at delivery and still choose to send the van only twice a day, for business reasons.

7. That is a real and respectable choice. The point of delivery is that the delay is a decision, not a limitation.

Where this comparison breaks: bread cannot be un-delivered. Software can be rolled back, sometimes. Also the van either goes or does not, whereas software can go to one percent of users first, which has no bakery equivalent.

■ 41.2.3 PLAIN – a worked example

1. Here is one change moving through a delivery pipeline, with times.

Stage	What happens	Time
Commit stage	Lint, build, unit tests	4 min
Package	Build image, push it	2 min
Staging deploy	Install to staging	1 min
Acceptance tests	End-to-end suite	9 min
Ready to release	Waiting for approval	0 min
Production deploy	Rolling update	3 min

2. Total machine time is 19 minutes. Under continuous delivery the change sits at “ready to release” until a person approves it.
3. Under continuous deployment, the row “Ready to release” is deleted and the change is live 19 minutes after the commit.
4. The test suites did not change. The pipeline did not change. Only the gate changed.

■ 41.2.4 PLAIN – what is really happening inside

1. The pipeline is a chain of stages. Each stage takes the output of the one before, not a fresh copy of the source.
2. That is the key rule: you build the artifact once, and the same bytes are what gets tested and what gets deployed.
3. If you rebuild before deploying, you have tested something other than what you shipped, and the test result no longer applies.
4. Early stages are fast and cheap and run on everything.
5. Later stages are slow and expensive and only run on things that passed.
6. Each stage either passes the artifact forward or stops it dead.
7. The pipeline itself records which commit each artifact came from, so you can always answer “what is running in production, exactly”.
8. Continuous deployment simply removes the human gate from the last hop, and compensates with automatic checks after the deploy: health checks, error rate alarms, and automatic rollback.

■ 41.2.5 TECHNICAL – the engineer’s version

1. The deployment pipeline as a named pattern comes from Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation by Jez Humble and David Farley, published in 2010.
2. Its core rules are: build binaries once, deploy the same way to every environment, smoke-test every deployment, and stop the line if any stage fails.
3. Continuous delivery: every change that passes automated checks is a release candidate that could be deployed on demand. Continuous deployment: every such change is deployed without human intervention.

4. The DORA research programme, published in the State of DevOps reports from 2014 and in the 2018 book *Accelerate* by Nicole Forsgren, Jez Humble and Gene Kim, measures four things.

DORA metric	What it measures
Deployment frequency	How often you ship
Lead time for changes	Commit to production time
Change failure rate	Percent of deploys causing harm
Failed deployment recovery	Time to restore service

5. The consistent finding across those reports is that speed and stability move together rather than trading off. That is a statistical association in survey data, not a controlled experiment. Treat it as strong evidence, not proof.
6. There is genuine expert disagreement about whether continuous deployment is right for every domain. Web services with cheap rollback lean yes. Firmware, medical devices and payment cores lean no, because the cost of a bad release is not symmetric with the cost of a slow one.
7. The honest version: most organizations that say “we do CD” mean continuous delivery to a staging environment and manual promotion to production. That is a fine place to be. It is just not continuous deployment.

■ 41.2.6 WORDS - remember these

1. Continuous delivery — always ready to ship — every passing change is a deployable release candidate, promoted on demand.
2. Continuous deployment — ships by itself — every passing change is deployed to production without manual approval.
3. Deployment pipeline — the chain of stages a change walks — an automated implementation of the path from commit to release, with binary reuse.
4. Release candidate — a version that could go live — an immutable artifact that has passed all preceding pipeline stages.
5. Promotion — moving an artifact to the next environment — deploying the same artifact to a higher environment without rebuilding.
6. Lead time for changes — how long from writing to live — the DORA metric measuring commit timestamp to production deploy timestamp.

41.3 What a runner actually is

■ 41.3.1 PLAIN - in simple words

1. Something has to actually execute your build. That something is a computer.
2. In continuous integration that computer is called a **runner**, or an agent, or an executor, depending on the product.
3. It can be a whole machine, a virtual machine, or a container. From your side it does not matter much: it is a place with a filesystem and a shell.
4. When work arrives, the runner downloads your code at one exact commit and runs your steps one after another, in order, in a shell.
5. It reports the exit code of each step. Zero means success. Anything else means failure, and by default the job stops there.

- Two kinds exist. **Hosted** runners are supplied by the service, created fresh for your job, and destroyed afterwards. **Self-hosted** runners are machines you own, running an agent program that asks for work.
- The most important fact about a fresh hosted runner: it contains only what the image ships with. Nothing of yours. No files, no settings, no tools you installed last year and forgot about.
- That is exactly why builds fail on the runner and pass on your laptop.

■ 41.3.2 PLAIN – a picture in your head

- Imagine a hotel kitchen you can rent by the minute.
- It is spotless. Every surface wiped. Standard ovens, standard knives.
- Whatever you want to cook, you must bring the recipe and the ingredients, or ask the hotel for them at the door.
- Your own kitchen at home has a jar of your grandmother’s spice mix on the top shelf. You use it without thinking. It is not in your recipe.
- In the hotel kitchen, the dish comes out wrong, and you cannot see why, because the missing thing was never written down anywhere.
- When your work is finished, the hotel demolishes the kitchen and builds a new one for the next customer. Nothing you left behind survives.

Where this comparison breaks: a real kitchen is never truly identical twice. Runner images are updated every week or two, so tool versions drift under you. And a self-hosted runner is more like your own kitchen: it does keep the spice jar, which is convenient right up until it becomes the bug.

■ 41.3.3 PLAIN – a worked example

- This step works perfectly on the reader’s macOS laptop and fails on a Linux runner.

```
| python script.py --config Config.json
```
- On macOS the default filesystem is APFS, set up case-insensitive. The file on disk is `config.json`, and `Config.json` finds it anyway.
- On the Linux runner the filesystem is case-sensitive. `Config.json` does not exist. The script exits non-zero. The job fails.
- The error message is `FileNotFoundError`, which sounds like the file was not committed, so the first hour is spent looking in the wrong place.
- Here are the usual suspects, all of the same shape.

Works locally because	Runner reality
Case-insensitive filesystem	Case-sensitive filesystem
Tool installed years ago	Only image defaults present
<code>.env</code> file on disk	Untracked files do not exist
Local time zone	Runner is UTC
Logged in to a registry	No credentials at all
Dependencies already cached	Cold download every time

■ 41.3.4 PLAIN – what is really happening inside

- A job goes through the same seven phases every time. Learn them and you can place any failure.

```
| 1 QUEUE    job waits for a free runner
| 2 ALLOCATE a runner is chosen or a VM is created
```

```

3 PROVISION boot image, start agent, set environment
4 CHECKOUT  fetch the repository at one exact commit
5 RESTORE   pull caches: packages, compiler output
6 STEPS     run each step in order, stop on failure
7 UPLOAD    save artifacts and logs
8 CLEAN     destroy the machine or wipe the workspace

```

2. Phase 1 costs nothing but wall-clock time. If jobs sit here, you have hit a concurrency limit, not a bug.
3. Phase 3 is where the operating system image is decided. That image is a product of the service and it changes.
4. Phase 4 usually does a shallow fetch of depth one: only the one commit, not the whole history, because history is large and rarely needed.
5. Phase 5 restores a cache keyed on something like the hash of your lock file. A cache miss is slow but correct. A wrong cache hit is fast and wrong, which is worse.
6. Phase 6 runs each step as a separate shell invocation. Variables set with plain `export` in one step do not survive to the next step, because the shell exited.
7. Phase 8 is why “it left a file behind last time” is never an explanation on a hosted runner. There is no last time.

■ 41.3.5 TECHNICAL - the engineer’s version

1. GitHub-hosted runners are ephemeral virtual machines in Microsoft Azure, with a fresh instance per job, running the open-source actions/runner agent. Figures below were taken from GitHub documentation in August 2026 and change over time.

Label	vCPU	RAM	Disk
ubuntu-latest public	4	16 GB	14 GB
ubuntu-slim public	1	5 GB	14 GB
windows-latest public	4	16 GB	14 GB
macos-latest (M1)	3	7 GB	14 GB

2. For private repositories the standard Linux and Windows runners are smaller, at 2 vCPU and 8 GB RAM as of August 2026. Larger runners are a paid option.
3. Hard limits for GitHub Actions, as documented in August 2026: a job may run up to 6 hours on a hosted runner and up to 5 days on a self-hosted runner; a workflow run is cancelled after 35 days; a matrix can generate at most 256 jobs per run; a self-hosted job may sit queued for 24 hours.
4. Concurrent job limits by plan: 20 on Free, 40 on Pro, 60 on Team and 500 on Enterprise. The `GITHUB_TOKEN` is rate limited to 1,000 API requests per hour per repository.
5. GitLab uses a separate `gitlab-runner` binary with pluggable executors: `docker`, `shell`, `kubernetes`, `docker-machine` and others. Jenkins calls the same idea an agent, historically a “slave”, connected by JNLP or SSH.
6. Self-hosted runners on a public repository are a documented security hazard: a fork can propose a workflow change that executes on your hardware. GitHub documents this and recommends self-hosted runners only for private repositories, or with ephemeral, isolated, single-use runners.
7. Runner images are built in the open. The `actions/runner-images` repository publishes the exact contents of each image and a changelog per release, which is the correct place to check whether a tool version moved under you.

8. Observing tools: runs -on in the workflow file, the “Set up job” log group which prints the image version and the resolved runner name, `uname -a, df -h, nproc and env | sort` as diagnostic steps.

■ 41.3.6 WORDS - remember these

1. Runner — the machine that executes your build — an ephemeral or persistent host running an agent that claims and executes jobs.
2. Hosted runner — supplied and destroyed by the service — a per-job ephemeral VM from a published image, with no state between jobs.
3. Self-hosted runner — your own machine doing the work — a long-lived or ephemeral host running the agent, with persistent state unless wiped.
4. Job — one unit of work on one runner — a named collection of steps sharing a workspace, filesystem and environment.
5. Step — one command or one action inside a job — a separate shell invocation or a container/JavaScript action execution.
6. Workspace — the folder your code is checked out into — the working directory, exposed as `GITHUB_WORKSPACE` or `CI_PROJECT_DIR`.
7. Exit code — the number a command returns — 0 for success, non-zero for failure; the runner uses it to decide whether to continue.

41.4 The workflow file

■ 41.4.1 PLAIN - in simple words

1. Somewhere you must write down what the machine should do. That written description is the **workflow file**.
2. It is a plain text file in YAML, a format that uses indentation instead of brackets to show what belongs to what.
3. On GitHub it lives in the folder `.github/workflows/` inside your repository, and any file there ending in `.yml` or `.yaml` is picked up.
4. On GitLab it is a single file called `.gitlab-ci.yml` at the top of the repository.
5. It sits inside the repository, next to the code, in the same history. That is the important part.
6. Because it is in the history, the workflow that ran for an old commit is the workflow as it existed at that commit. Not today’s version.
7. So changing your tests and changing how they run is one change, one review, one commit. They can never drift apart.

■ 41.4.2 PLAIN – a picture in your head

1. Think of a recipe card taped inside the lid of the box that holds the ingredients.
2. If you change the ingredients, you open the box, so you see the recipe and change it too.
3. Compare with a recipe pinned on a noticeboard in another room. It slowly becomes a description of a dish nobody makes any more.
4. A configuration stored in a web interface on the build server is the noticeboard. A workflow file in the repository is the card inside the lid.
5. And if you open a box from three years ago, the card inside tells you exactly how that batch was made.

Where this comparison breaks: the recipe card is not fully self-contained. It says “use action X at version 4”, and version 4 is fetched fresh at run time from somewhere else. So an old commit can still build differently today. Section 41.13 explains how pinning fixes that.

■ 41.4.3 PLAIN – a worked example

1. Here is a complete, working GitHub Actions workflow. Every line is explained underneath.

```
name: CI

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

permissions:
  contents: read

concurrency:
  group: ci-${{ github.ref }}
  cancel-in-progress: true

env:
  CI: "true"

jobs:
  lint:
    runs-on: ubuntu-latest
    timeout-minutes: 10
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: "22"
          cache: npm
      - run: npm ci
      - name: Static checks
        run: npm run lint

  test:
    needs: [lint]
    runs-on: ${{ matrix.os }}
    timeout-minutes: 20
    strategy:
      fail-fast: false
    matrix:
```

```

    os: [ubuntu-latest, macos-latest]
    node: ["20", "22"]
  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v4
      with:
        node-version: ${ matrix.node }
        cache: npm
    - run: npm ci
    - name: Unit tests
      run: npm test
    - env:
        TZ: UTC
    - name: Save the report
      if: always()
      uses: actions/upload-artifact@v4
      with:
        name: junit-${ matrix.os }-${ matrix.node }
        path: reports/junit.xml

```

2. name is the label shown in the user interface. Cosmetic only.
3. on lists the events that start a run. Here: a push to main, and any pull request whose target is main.
4. permissions sets what the automatic token may do. contents: read means the job can read the repository and nothing else.
5. concurrency groups runs. Two runs in the same group cannot overlap, and cancel-in-progress: true kills the older one. This stops five pushes in a minute from starting five full pipelines.
6. env at the top sets environment variables for every job.
7. jobs holds one or more jobs. lint and test are job IDs, chosen by you.
8. runs-on picks the runner label. ubuntu-latest means a hosted Linux VM.
9. timeout-minutes kills a job that hangs. Without it a stuck job burns until the six-hour limit.
10. steps is an ordered list. Each item is either uses or run.
11. uses runs a prepackaged **action**, here actions/checkout at major version 4, which clones the repository at the right commit.
12. with passes named inputs to that action.
13. run executes a shell command on the runner.
14. name on a step is the label in the log. Give every non-obvious step one, because that label is what you will search for when it fails.
15. needs: [lint] makes test wait for lint to succeed. Without needs, jobs run in parallel.
16. strategy.matrix expands one job into several. Two operating systems by two Node versions gives four jobs.
17. fail-fast: false says: if one matrix cell fails, let the others finish. The default, true, cancels the siblings, which is faster but tells you less.
18. if: always() on the upload step means “run this even if a previous step failed”, which is how you get the test report from a failing build.
19. \${ ... } is an expression. It is evaluated by the service before the step runs, and substituted as text.

■ 41.4.4 PLAIN – what is really happening inside

1. Here is the same pipeline written for GitLab CI, in .gitlab-ci.yml.

```

stages: [lint, test]

```



```

default:
  image: node:22-bookworm
  cache:
    key:
      files: [package-lock.json]
    paths: [.npm/]

variables:
  CI: "true"

lint:
  stage: lint
  script:
    - npm ci --cache .npm --prefer-offline
    - npm run lint

unit-test:
  stage: test
  needs: [lint]
  parallel:
    matrix:
      - NODE_TAG: ["20", "22"]
  image: node:${NODE_TAG}-bookworm
  script:
    - npm ci --cache .npm --prefer-offline
    - npm test
  artifacts:
    when: always
    reports:
      junit: reports/junit.xml
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH

```

2. The differences are worth naming, because they are differences of model, not of syntax.
3. GitLab has a first-class concept of **stages**: named phases that run in order, with all jobs in a stage running together. GitHub has no stages, only needs edges between jobs, which form a graph.
4. GitLab jobs run inside a container image by default, set with `image`. GitHub jobs run directly on the runner VM, and containers are opt-in.
5. GitLab has no uses. Reuse comes from `include`, `extends` and YAML anchors, plus CI/CD components. GitHub’s marketplace of actions has no direct GitLab equivalent.
6. GitLab checks out the code for you automatically. GitHub requires an explicit `actions/checkout` step, and forgetting it is a classic first-day mistake.
7. GitLab’s `rules` replaces GitHub’s `on` plus `if`, combining trigger and condition into one list evaluated top to bottom.
8. Both put the file in the repository, and both therefore get the same benefit: the pipeline definition is versioned, reviewable and bisectable.

■ 41.4.5 TECHNICAL – the engineer’s version

1. GitHub reads every file matching `.github/workflows/*.yml` and `.github/workflows/*.yaml` on the branch that the event applies to. This is a documented behaviour, not a convention.
2. For push events the workflow file used is the one at the pushed commit. For `pull_request` events it is the one at the pull request’s head. For `pull_request_target` and `schedule` it is the one on the base or default branch. That asymmetry is the root of section 41.5’s security discussion.

3. Available contexts inside `${{ }}` include `github`, `env`, `vars`, `secrets`, `job`, `steps`, `runner`, `strategy`, `matrix`, `needs` and `inputs`. Context values are substituted before the shell sees them, which is why interpolating untrusted text directly into a run block is a script injection hole. Pass it through `env` instead.
4. Each run block is written to a temporary script and executed with the default shell: `bash -e` on Linux and macOS, `pwsh` on Windows. Because `-e` is set, the first failing command ends the step. Add `set -o pipefail` if you use pipes, since a pipeline’s exit code is the last command’s.
5. YAML has a real trap here. In YAML 1.1, the bare word `on` is a boolean `true`, so strict linters flag `on:` as the key `true`. GitHub’s parser special-cases it. Quoting as `"on"`: also works. This is an implementation detail worth knowing when a linter shouts at a file that runs fine.
6. Reuse mechanisms: composite actions (steps bundled in an `action.yml`), reusable workflows called with `uses` at job level and triggered by `workflow_call`, and Docker container actions. Reusable workflows can be nested up to four levels deep.
7. GitLab equivalents: `include:local`, `include:project`, `include:remote`, `extends`, and CI/CD components introduced in GitLab 16.6 in late 2023.
8. Because the file is versioned, `git log -p .github/workflows/` answers “when did this check start being required” and `git bisect` can walk a pipeline regression the same way it walks a code regression.
9. A practical consequence the reader met directly: a GitHub personal access token needs the `workflow` scope to push a commit that adds or edits a file under `.github/workflows/`. Without it the push is refused, even though the same token can push ordinary code. The workflow file is treated as more dangerous than code, because it is.

■ 41.4.6 WORDS – remember these

1. Workflow file — the written plan for the machine — a YAML document defining triggers, jobs and steps, stored in the repository.
2. YAML — a text format that uses indentation — a data serialization language; GitHub Actions and GitLab CI both use it for pipeline definitions.
3. Action — a reusable prepackaged step — a JavaScript, Docker or composite unit referenced by `uses: owner/repo@ref`.
4. Matrix — one job definition expanded into many — a cross product of variable values generating up to 256 jobs per run.
5. Stage — a named phase in GitLab — an ordered grouping of jobs; GitHub models the same idea as `needs` dependencies instead.
6. Context — the data available to expressions — named objects such as `github`, `matrix` and `secrets` resolved before the step runs.
7. Script injection — untrusted text becoming code — interpolating attacker controlled values into a run block, mitigated by passing through `env`.

41.5 Triggers: what starts a run

■ 41.5.1 PLAIN – in simple words

1. A run never starts on its own. Something has to ask for it.
2. The thing that happens is called an **event**. The rule in your file that listens for it is called a **trigger**.
3. The commonest event is a **push**: you send new commits to the shared server.
4. The next commonest is a **pull request**: you propose joining one branch into another.
5. A clock can be the event: run every night at two in the morning.

6. A person can be the event: somebody clicks a “Run workflow” button.
7. Another program can be the event: an outside system sends a message asking for a run.
8. Creating a **tag**, which is a permanent name pinned to one commit, can be the event.
9. Publishing a **release**, which is a tag plus notes and downloadable files, can be the event.
10. Every event carries facts with it: which repository, which branch, which commit, and who caused it.
11. Your workflow file lists the events it cares about. Every other event is ignored by that file.
12. You can narrow an event further: only on this branch, only when these files changed.
13. Narrowing matters because every run costs minutes and money, and most runs on most changes are wasted work.

■ 41.5.2 PLAIN – a picture in your head

1. Picture a workshop with a row of bells along one wall.
2. Bell one rings when a delivery van drops parts at the door. That is push.
3. Bell two rings when somebody posts a proposed change to the plans. That is pull_request.
4. Bell three is wired to a timer and rings itself at 2 a.m. every night. That is schedule.
5. Bell four has a rope hanging down, and any worker may pull it whenever they like. That is workflow_dispatch.
6. Bell five is a phone line from the factory next door, so they can call you. That is repository_dispatch.
7. Beside each bell hangs a small card saying which deliveries are worth getting up for.
8. One card says “steel only, ignore paint”. That is a path filter.
9. Another says “only deliveries addressed to the main workshop”. That is a branch filter.

Where this comparison breaks: bells in a workshop ring one at a time. Real events overlap. A single push to a branch that already has an open pull request rings two bells and can start two nearly identical runs on nearly the same code. Also a bell carries no information, while an event carries a whole document describing exactly what happened, which your job can read.

■ 41.5.3 PLAIN – a worked example

1. Here is one on: block using most of the common triggers at once.

```
on:
  push:
    branches:
      - main
      - 'release/**'
  tags:
    - 'v*.*.*'
  paths:
    - 'src/**'
    - 'package-lock.json'
  pull_request:
    types: [opened, synchronize, reopened, ready_for_review]
    branches: [main]
  schedule:
    - cron: '17 2 * * *'
  workflow_dispatch:
    inputs:
      environment:
        description: 'Where to deploy'
        type: choice
        options: [staging, production]
```

```

    default: staging
  dry_run:
    type: boolean
    default: true
  repository_dispatch:
    types: [deploy-request]
  release:
    types: [published]

```

2. `push.branches` lists branch name patterns. `release/**` matches `release/2026-08` and `release/hotfix/db`.
3. `push.tags` lists tag patterns. `v*.*.*` matches `v2.4.1`. If both branches and tags are present, a match on either one starts a run.
4. `push.paths` says: only start if at least one changed file matches. A commit that only edits `README.md` starts nothing.
5. `pull_request.types` lists the sub-kinds of the event. `synchronize` means “the pull request branch got a new push”, which is the one people forget.
6. `ready_for_review` fires when a draft pull request stops being a draft. It is not in the default list, so you must ask for it.
7. `schedule.cron` is five fields: minute, hour, day of month, month, day of week. `17 2 * * *` is 02:17 every day, in UTC always.
8. The 17 is deliberate. Everyone writes `0 * * * *`, so the top of the hour is the busiest queue on the service. Pick an odd minute.
9. `workflow_dispatch.inputs` builds a small form in the web page and also accepts the same values from the command line.
10. `repository_dispatch.types` filters on a label chosen by the outside caller, so one repository can serve several kinds of external request.
11. `release.types: [published]` fires when a release is published, which is the usual moment to build and upload artifacts.

Trigger	Typical use
<code>push</code>	Test the mainline
<code>pull_request</code>	Test a proposal
<code>schedule</code>	Nightly, slow scans
<code>workflow_dispatch</code>	Manual deploy
<code>repository_dispatch</code>	Called by another system
<code>release</code>	Publish artifacts

■ 41.5.4 PLAIN - what is really happening inside

1. Now the confusing pair. `pull_request` and `pull_request_target` look almost the same and behave very differently.
2. When somebody outside your team proposes a change from their own copy, that code is a stranger’s code. You have not read it yet.
3. A run started by `pull_request` does check out the stranger’s code, but it runs with almost no power.
4. It gets a read-only token, and it gets none of your secrets.
5. So even if the stranger’s code is hostile, it can make a mess inside its own throwaway machine and steal nothing.

6. That is the safe default. It is also why a first-time contributor often sees “waiting for a maintainer to approve running workflows”.
7. But it is inconvenient. A run with no secrets cannot post a comment, cannot upload to your storage, and cannot add a label.
8. So a second trigger exists: `pull_request_target`.
9. `pull_request_target` reads the workflow file from your own base branch, not from the stranger’s branch.
10. Because the file is yours and was reviewed by you, the service trusts it and hands it a write token and your secrets.
11. Here is the trap, and it is a sharp one. The workflow file is yours. The code being proposed is still theirs.
12. If your trusted workflow checks out the pull request’s head and then executes anything from it, you have just handed a stranger your keys.
13. “Executes anything” is wider than it sounds. It includes `npm test`. It also includes `npm install`, because a dependency can run an install script.
14. It includes a build tool that reads a config file naming a plugin, a Makefile, a test fixture, or a linter with a custom rule.
15. GitHub’s own security team named this family of bug **pwn requests** in an article published in August 2020.
16. The rule is one sentence. With `pull_request_target`, do not check out the head; and if you truly must, do not execute one byte of it.

	<code>pull_request</code>	<code>pull_request_target</code>
	-----	-----
workflow file	PR head branch	base branch
checked out	PR merge result	base branch by default
token	read-only (fork)	read and write
secrets	none (fork)	all of them
safe by	having no power	you not running their code

■ 41.5.5 TECHNICAL - the engineer’s version

1. GitHub Actions documents more than thirty webhook-backed events plus the non-webhook triggers `schedule`, `workflow_dispatch` and `workflow_call`.

Trigger	Fires when	Note
<code>push</code>	commits or tags land	branch, tag, path filters
<code>pull_request</code>	PR opened, updated	forks get no secrets
<code>pull_request_target</code>	same, base context	write token, dangerous
<code>schedule</code>	cron time in UTC	5 minute minimum
<code>workflow_dispatch</code>	manual or API call	up to 10 inputs
<code>repository_dispatch</code>	external POST	default branch only
<code>release</code>	release published	types include published
<code>merge_group</code>	merge queue asks	<code>checks_requested</code>
<code>workflow_run</code>	another run ends	base branch file

2. The default activity types for `pull_request` are `opened`, `synchronize` and `reopened`. `synchronize` fires on every new push to the head branch.
3. `schedule` uses five-field POSIX cron, always evaluated in UTC, never in local time. The shortest documented interval is every 5 minutes.

4. Scheduled runs are queued, not guaranteed punctual. Delays of many minutes at the top of the hour are normal and documented.
5. In a public repository, a scheduled workflow is disabled automatically after 60 days with no repository activity. This is documented behaviour and a very common cause of “our nightly build silently stopped”.
6. `workflow_dispatch` accepts at most 10 top-level inputs, of types string, choice, boolean and environment, with a total payload limit of 65,535 characters. Start it with `gh workflow run` or `POST /repos/{owner}/{repo}/actions/workflows/{id}/dispatches`.
7. `repository_dispatch` is `POST /repos/{owner}/{repo}/dispatches` carrying `event_type` and an optional `client_payload` JSON object, readable as `github.event.client_payload`. It only starts workflows whose file lives on the default branch, which surprises people testing it from a branch.
8. Filter keys are branches, branches-ignore, tags, tags-ignore, paths and paths-ignore. You cannot use a key and its -ignore partner for the same event in the same workflow.
9. Glob syntax in these filters: `*` matches any characters except `/`, `**` matches any characters including `/`, `?` matches one character, `+` and `*` act as repeaters, `[]` is a character class, and a leading `!` negates.
10. A documented caveat with real consequences: if a push contains more than 1,000 commits, or GitHub cannot compute the diff because it timed out, path filters are ignored and the workflow runs anyway.
11. The opposite caveat matters more. A path-filtered workflow that does not run produces no check at all, and a required status check that never arrives blocks the pull request forever. Section 41.8 covers the fix.
12. The precise difference between the two pull request triggers:

Aspect	<code>pull_request</code>	<code>pull_request_target</code>
Workflow file from	PR head	base branch
<code>GITHUB_REF</code>	<code>refs/pull/N/merge</code>	base branch ref
Fork secrets	none	all
Fork token	read-only	read and write

13. For a fork-originated `pull_request`, the `GITHUB_TOKEN` is issued with read permissions only, and no repository, environment or organization secret is exposed. Public repositories can additionally require maintainer approval before a first-time contributor’s workflow runs at all.
14. The dangerous pattern, written out so you can recognize it in review:

```
# DANGEROUS when combined with pull_request_target
- uses: actions/checkout@v4
  with:
    ref: ${github.event.pull_request.head.sha}
- run: npm ci && npm test    # their code, your secrets
```

15. The safe pattern when you genuinely need write access on a fork pull request: split into two workflows. The `pull_request` workflow does the untrusted work with no secrets and uploads an artifact. A second workflow on `workflow_run` has permissions, downloads that artifact, and treats its contents as data, never as code.
16. `workflow_run` carries the same hazard in a different coat: it also runs the file from the default branch with a full token, so an artifact produced by an untrusted run must never be executed or expanded into a path you trust.
17. GitLab expresses all of this through rules and the `CI_PIPELINE_SOURCE` variable, whose values include `push`, `merge_request_event`, `schedule`, `web`, `api`, `trigger` and `pipeline`. GitLab further

distinguishes merge request pipelines from merged results pipelines, which test the combined result rather than the source branch.

18. Observing tools: `gh run list --json event,headBranch,headSha`, the event name shown at the top of any run page, and `gh api repos/OWNER/REPO/hooks` to see which webhooks a repository sends to outside systems.

■ 41.5.6 WORDS - remember these

1. Event — the thing that happened — a webhook payload describing a repository or platform change.
2. Trigger — the rule that listens — the on: key naming events, activity types and filters.
3. Activity type — the sub-kind of an event — such as opened or synchronize for `pull_request`.
4. Path filter — only run when these files changed — paths or paths-ignore globs matched against the push diff.
5. `workflow_dispatch` — the manual run button — a non-webhook trigger with up to 10 typed inputs, also callable by API.
6. `repository_dispatch` — an outside system asking for a run — a POST to the repository’s dispatches endpoint with an `event_type`.
7. Pwn request — a stranger’s code running with your keys — arbitrary code execution obtained by checking out untrusted head under `pull_request_target`.

41.6 The most important idea: a run belongs to a commit, not a branch

■ 41.6.1 PLAIN - in simple words

1. This is the idea the whole chapter is built around. Read it twice.
2. A branch name is not a box holding commits. It is a label stuck on exactly one commit.
3. Chapter 38 put it this way: a branch is a moving pointer. Add a commit and the label moves forward to the new one.
4. A test run is not stuck to a label. It is stuck to one exact commit.
5. The machine downloaded one commit, tested those exact bytes, and wrote the answer against that commit’s fingerprint.
6. That fingerprint is the commit hash, a long string of hexadecimal digits computed from the contents.
7. So the sentence “CI passed on main” is short for something much longer.
8. In full it means: “CI passed on whichever commit main happened to point at when that run started”.
9. If main has moved since, the sentence stays true and becomes useless. It is a fact about a commit nobody is looking at any more.
10. The tick you see beside a branch name on a web page is a summary drawn at the moment the page loaded. It is not a promise about now.
11. So never accept a tick beside a branch name as evidence that anything in particular was tested.
12. Get the exact commit hash first. Then ask whether that hash passed.

■ 41.6.2 PLAIN - a picture in your head

1. Think of a hotel with a brass plate on one door reading MANAGER.
2. A health inspector visits, inspects the room behind that plate, and issues a certificate.
3. The certificate does not say MANAGER. It says room 412, and it lists what was in room 412 on that day.

4. Months later the manager moves office. Somebody unscrews the brass plate and screws it onto room 509.
5. Now walk up to the door marked MANAGER and ask: is this room certified?
6. The plate is not the room. The certificate belongs to 412. The door in front of you is 509, and nobody has inspected it.
7. The plate is the branch name. The room number is the commit hash. The certificate is the run.

Where this comparison breaks: a hotel room keeps its contents when the plate moves, so you could at least argue about it. A commit is defined by its contents, so it cannot change at all. Alter one byte and it becomes a different commit with a different hash and no certificate whatsoever. Git is stricter than the hotel, not looser.

■ 41.6.3 PLAIN – a worked example

1. During the reader’s outage, the machine reported origin/main as 0a95cc8.
2. That was a cached remote-tracking value written by the last successful fetch, not a live fact about the server.
3. Now watch the sequence that makes a green tick lie. The clock times are invented; the shape is exactly real.

```
09:00 main -> 0a95cc8
09:01 run 1187 starts, head_sha = 0a95cc8
09:07 run 1187 finishes, conclusion = success
09:07 branch page shows: main [tick]

09:40 a pull request is merged
09:40 main -> 3f1d02c          (the label moved)
09:40 branch page still shows: main [tick]  <- nothing
                                           tested 3f1d02c

09:41 run 1194 starts, head_sha = 3f1d02c
09:53 run 1194 finishes, conclusion = failure
```

4. Between 09:40 and 09:41 the page showed a tick beside main while nothing at all had tested 3f1d02c.
5. At 09:53 the honest summary is: 0a95cc8 passed, 3f1d02c failed, and main is 3f1d02c.
6. If at 09:40 somebody asked “did main pass?”, the correct answer is that the question is badly formed.
7. The well-formed question is “did 3f1d02c pass?”, and at 09:40 the answer was “no run for that commit exists yet”.
8. There is a second way to get burned where nobody does anything wrong.
9. The reader rebased a local branch onto a moved main, and the branch’s hash changed.
10. Rebase does not carry commits across. It builds new commit objects with new parents, and new parents mean new hashes.
11. Every run recorded against the old hashes is still there, still correct, and now attached to commits that no branch points at.
12. The branch label now sits on brand new commits with zero runs. The tick you remember belonged to commits that are effectively gone.

■ 41.6.4 PLAIN – what is really happening inside

1. A run record on the server stores many fields. Two of them matter here.
2. head_sha is the exact commit the run checked out. This field is the truth.
3. head_branch is the branch name as it read at trigger time. This field is a label, and labels move.

4. When you open a branch page, the service does roughly this: find the commit the branch points at right now, then look up the statuses and check runs attached to that commit.
5. That is correct behaviour, and it gives a right answer for the instant the page was rendered.
6. So where does the lie come from? Three places, none of them a bug.
7. Caching in your own tools. The reader’s machine reported 0a95cc8 from a file on disk, either .git/refs/remotes/origin/main or an entry in .git/packed-refs, written by the last fetch.
8. Human memory. You looked at the tick five minutes ago, then merged three pull requests, and then trusted the memory.
9. Frozen evidence. A screenshot or a chat message saying “CI is green, see attached” is a statement about a moment, not about now.
10. The fix is mechanical and takes one extra command.
11. Never carry a branch name into the verification step. Resolve it to a hash once, from the server, and carry the hash.
12. If somebody pushes between your resolve and your check, your conclusion is still exactly true, because it was a statement about the hash you named.
13. That is the whole difference between a claim you can defend and a claim you merely believe.

■ 41.6.5 TECHNICAL – the engineer’s version

1. Step one: get the head SHA from the server, not from your local cache.

```
# live from the remote, no local cache involved
git ls-remote origin refs/heads/main
# 3f1d02c9d5f0a1b2c3d4e5f60718293a4b5c6d7e refs/heads/main

# or through the API
export SHA=$(gh api repos/OWNER/REPO/commits/main --jq .sha)
echo "$SHA"
```

2. git rev-parse origin/main does not do this. It reads the remote-tracking ref from disk and answers from cache. That is what produced 0a95cc8 during the reader’s outage while the real server value was unknown.
3. Step two: list the runs filtered by that exact SHA, never by branch.

```
gh run list --commit "$SHA" \
  --json databaseId,workflowName,event,status,conclusion,headSha
```

4. The same query in raw REST, which is what gh calls underneath:

```
gh api "repos/OWNER/REPO/actions/runs?head_sha=$SHA" \
  --jq '.workflow_runs[]
    | {id, name, event, status, conclusion, head_sha}'
```

5. Step three: look at the one run you care about, by its numeric ID, and read both its conclusion and its head SHA.

```
gh run view 11947722301 \
  --json status,conclusion,headSha,headBranch,event,url
```

6. Step four: assert rather than eyeball. A human reading two long hex strings side by side will miss a mismatch.

```
gh run view "$RUN_ID" --json headSha,conclusion \
  --jq 'if .headSha == env.SHA and .conclusion == "success"
    then "VERIFIED" else "NOT VERIFIED" end'
```

7. Two commit-level views give the rolled-up answer for that SHA, one per API:

```
# newer checks API, one entry per check run
gh api "repos/OWNER/REPO/commits/$SHA/check-runs" \
  --jq '.check_runs[] | {name, status, conclusion}'

# older commit status API, already rolled up
gh api "repos/OWNER/REPO/commits/$SHA/status" \
  --jq '{state, total_count}'
```

8. For a pull request, ask for the head object ID directly rather than deriving it, and pull the rollout in the same call:

```
gh pr view 42 --json headRef0id,statusCheckRollup \
  --jq '{sha: .headRef0id,
        checks: [.statusCheckRollup[]
                  | {name, conclusion}]}'
```

9. Field naming differs between the two interfaces and it trips people up. The `gh --json` output uses camelCase such as `headSha` and `databaseId`. REST uses snake_case such as `head_sha` and `id`. Same values, two spellings.
10. GET `/repos/{owner}/{repo}/actions/runs` accepts `head_sha`, `branch`, `event`, `status`, `actor`, `created`, `check_suite_id` and `exclude_pull_requests` as query parameters.
11. One commit can carry many runs: one per workflow file, plus every re-run, plus a separate run for the push event and the pull_request event. So “did this SHA pass” is incomplete without “in which workflow”.
12. A subtlety worth memorizing. For a pull_request event the run’s `head_sha` is the pull request branch head, but the code checked out is the ephemeral merge of that head into the base, which is what `GITHUB_SHA` holds. The run is filed under one hash and tested another.
13. The hash itself is a SHA-1 digest over the commit object in classic repositories, and SHA-256 in repositories created with the newer object format, which Git has supported experimentally since version 2.29 in October 2020. The commit object names its tree and its parents, so the hash transitively covers the entire content and history.
14. Because of that, verifying against a hash is verifying against exact bytes. There is no room left for “roughly the same code”.
15. Rebase consequence, stated precisely: rebasing writes new commit objects, so check runs and statuses remain attached to the abandoned hashes. They are not migrated, and there is no mechanism that would migrate them. `git reflog` still finds the old commits locally until garbage collection.
16. The sentence to memorize: a run R proves a property about commit R.`head_sha`, and proves nothing whatsoever about any ref.

■ 41.6.6 WORDS – remember these

1. Commit SHA — the fingerprint of one exact snapshot — a 40-hex SHA-1, or 64-hex SHA-256, digest over the commit object.
2. Ref — a name pointing at a commit — a file under `.git/refs` or a line in `.git/packed-refs`.
3. `head_sha` — the commit the run really tested — the field on a workflow run record, spelled `head-Sha` in `gh --json` output.
4. Remote-tracking ref — your last known copy of the server’s branch — `refs/remotes/origin/main`, updated only by fetch, never live.

5. `git ls-remote` — ask the server what it has right now — a live ref advertisement over the transport, bypassing every local cache.
6. Stale tick — a pass mark for a commit that is no longer the head — a status attached to a superseded SHA and displayed next to a moved branch name.

41.7 Why verifying by run ID against a head SHA is the stronger claim

■ 41.7.1 PLAIN – in simple words

1. In the reader’s session, continuous integration was verified by run ID against the exact head hash.
2. It was not verified by glancing at a tick beside a branch name. That is the difference between proof and belief.
3. Here is the procedure in words, four steps.
4. Ask the server what the branch points at right now. Write that hash down.
5. List the runs whose tested commit is that hash.
6. Pick the run you care about and note its number.
7. Read that run’s result, and read the commit that run tested, and confirm it equals the hash you wrote down.
8. Only then say “this commit passed”.
9. That claim survives questioning, because every step names something that cannot move.
10. A hash cannot move. A run number cannot move. A branch name moves all day.
11. The extra cost is about ten seconds and three commands.
12. The cost it avoids is a bad deploy at six o’clock on a Friday.
13. There is a second reason this method is stronger, and it is the reason engineers care most about.
14. It fails loudly. If nothing tested your commit, the list comes back empty and you cannot mistake an empty list for a pass.
15. A tick beside a branch name never comes back empty. It always shows something, and that is exactly the problem.

■ 41.7.2 PLAIN – a picture in your head

1. A school posts a notice on the board: “Class 10B: PASS”.
2. You are in class 10B. Are you finished?
3. Not yet. The board is a summary about a group, at a moment, written by somebody in a hurry.
4. Your own certificate carries your roll number. That number was never reassigned to anybody else.
5. Ask for the certificate bearing your roll number, and read the result printed on it.
6. Then compare the roll number on the certificate with the roll number on your identity card.
7. Two documents, one shared identifier, one comparison. That is the entire method, in school and in continuous integration.

Where this comparison breaks: a class notice can only be wrong in one way, by summarizing. A branch tick can be wrong in at least six different ways, and several of them look identical on the screen. Worse, a school certificate is issued by one office, while a commit status can be posted by anything holding a token, including a person typing an API call by hand.

■ 41.7.3 PLAIN – a worked example

1. Start from the branch name and refuse to trust it.

```
$ export SHA=$(gh api repos/OWNER/REPO/commits/main --jq .sha)
$ echo "$SHA"
3f1d02c9d5f0a1b2c3d4e5f60718293a4b5c6d7e
```

```
$ gh run list --commit "$SHA" \
  --json databaseId,workflowName,conclusion,headSha
[]
```

2. An empty list. Nothing has ever tested this commit, even though the branch page a moment ago showed a green tick.
3. That tick belonged to 0a95cc8, the previous head. The claim “main is green” was true about a commit nobody is deploying.
4. Now the good case, after the run has actually happened.

```
$ gh run list --commit "$SHA" \
  --json databaseId,workflowName,conclusion,headSha
[
  {
    "databaseId": 11947722301,
    "workflowName": "ci",
    "conclusion": "success",
    "headSha": "3f1d02c9d5f0a1b2c3d4e5f60718293a4b5c6d7e"
  }
]
```

5. Now name the run and read it back on its own, so the answer does not depend on the filter having worked.

```
$ gh run view 11947722301 \
  --json conclusion,headSha,status,event,runAttempt
{
  "conclusion": "success",
  "headSha": "3f1d02c9d5f0a1b2c3d4e5f60718293a4b5c6d7e",
  "status": "completed",
  "event": "push",
  "runAttempt": 1
}
```

6. Four things were checked and all four matter.
7. status is completed, so the run finished rather than still running.
8. conclusion is success, so it finished well.
9. headSha equals the hash you resolved, so it tested the right commit.
10. runAttempt is 1, so nobody re-ran it until it went green.
11. Now the sentence you may say out loud: commit 3f1d02c passed workflow ci in run 11947722301 on the first attempt.

■ 41.7.4 PLAIN - what is really happening inside

1. Six ways a branch-level tick can lie. Each is a real, ordinary situation, not a bug and not sabotage.
2. **The run was for an older commit.** The branch moved after the run. The page shows the newest result it can find, and briefly that is the old one.
3. **The run tested the merge result, not your head.** For a pull request the service builds a temporary merge of your branch into the target and tests that. Your head commit was never tested alone.
4. **A required check was skipped, not passed.** A job that never ran can report in a way that satisfies a rule, and a workflow that never started reports nothing at all.
5. **The status came from something other than the run you have in mind.** The name “build” on the screen is just a text label. Anything with a token can post that label.

6. **A re-run replaced the result.** Somebody pressed re-run until it went green. The run ID is the same, the number of attempts is not.
7. **The check belongs to a merge commit that no longer exists.** After a squash or rebase merge, the commit that was tested is not the commit that landed on the target branch.
8. Notice what all six have in common. In every case the tick is truthful about something, and that something is not the thing you are about to deploy.
9. The method in 41.7.1 defeats all six because it never asks a question whose subject can move.

■ 41.7.5 TECHNICAL – the engineer’s version

1. The failure modes and the exact field to inspect for each.

Failure mode	Detect by
Run is for an older commit	compare headSha to head
Run tested the merge result	read the event field
Check skipped, not passed	read conclusion
Status posted by another app	list app.slug per check
Re-run replaced the result	read runAttempt
Merge commit gone after merge	resolve the new head SHA

2. Older commit. `gh run list --commit "$SHA"` returns an empty array when nothing tested that hash. Treat an empty array as a hard failure of verification, never as an absence of bad news.
3. Merge result. If event is `pull_request`, the run’s `head_sha` is the pull request branch head, but the job checked out `refs/pull/N/merge`, and `GITHUB_SHA` inside the job was that ephemeral merge commit. GitHub’s own troubleshooting page states that when the test merge commit has a status, it is the test merge commit that must pass, not the head commit.
4. Skipped versus passed. This is the sharpest edge in the whole area, and GitHub documents three different outcomes:

Situation	Reported as
Workflow skipped by path filter	stays Pending, blocks
Workflow skipped by <code>[skip ci]</code>	stays Pending, blocks
Job skipped by an <code>if: condition</code>	Success
Job skipped after a failed needs	skipped, may not block

5. Read that table again. A job you skipped on purpose reports success and satisfies a required check. A workflow that never started reports nothing, stays pending, and blocks forever. Same English word, opposite consequences.
6. The documented mitigation for the fourth row is to add `if: always()` to a final summary job that depends on the others and explicitly fails if any dependency did not succeed.
7. Wrong poster. Any token with write access can create a commit status with any context and state, including success:

```
gh api -X POST repos/OWNER/REPO/statuses/$SHA \
  -f state=success -f context=build \
  -f description='looks fine to me'
```

8. That call is legitimate API use and is indistinguishable on the branch page from a real build result. Inspect the poster:

```
gh api "repos/OWNER/REPO/commits/$SHA/check-runs" \
  --jq '.check_runs[] | {name, conclusion, app: .app.slug}'

gh api "repos/OWNER/REPO/commits/$SHA/status" \
  --jq '.statuses[] | {context, state, creator: .creator.login}'
```

9. Re-runs. A re-run does not create a new run ID. It increments `run_attempt` on the same run, and the interface shows the latest attempt. Fetch an earlier attempt explicitly:

```
gh api repos/OWNER/REPO/actions/runs/11947722301/attempts/1 \
  --jq '{attempt: .run_attempt, conclusion, head_sha}'
```

10. "Re-run failed jobs" is more subtle still: it produces a new attempt in which the previously successful jobs are not executed again. The green you see is assembled from two different points in time.
11. Vanished merge commit. A squash merge creates one brand new commit on the target with a new hash and no checks. A rebase merge creates new commits for the same reason. Only a true merge commit preserves the tested parent.
12. So after any squash or rebase merge, the correct next step is to resolve the new head of the target branch and verify that hash on its own. What passed before the merge is evidence about a commit that is no longer in the first parent line of your branch.
13. Two further modes worth knowing, beyond the six:
14. `continue-on-error: true` on a step or job makes a real failure report as success at the job or run level. Read the step conclusions, not only the run conclusion.
15. A `strategy.matrix` job cancelled by `fail-fast: true` reports cancelled, which is not failure, and people read the absence of red as green.
16. A script for the whole verification, safe to put in a release checklist:

```
set -euo pipefail
REPO=OWNER/REPO
SHA=$(gh api "repos/$REPO/commits/main" --jq .sha)
RUN=$(gh run list --repo "$REPO" --commit "$SHA" \
  --workflow ci --limit 1 --json databaseId \
  --jq '.[0].databaseId')
test -n "$RUN" || { echo "no run for $SHA"; exit 1; }
gh run view "$RUN" --repo "$REPO" \
  --json conclusion,headSha,status,runAttempt \
  --jq --arg sha "$SHA" '
    if .status=="completed" and .conclusion=="success"
      and .headSha==$sha
    then "VERIFIED \($sha) run \(.runAttempt) attempt(s)"
    else error("NOT VERIFIED") end'
```

17. GitLab equivalent concepts: a pipeline has a sha, a status, and an id. `glab ci status` and the pipelines API filtered by sha do the same job. Jenkins records the built revision in the build metadata, reachable through the `api/json` endpoint of a build.

■ 41.7.6 WORDS – remember these

1. Verification by run ID — naming the run rather than the branch — asserting `run.head_sha == resolved_head` and `conclusion == success`.
2. Test merge commit — the temporary join of your branch into the target — the `refs/pull/N/merge` ref that pull request runs actually check out.
3. Run attempt — which try you are looking at — the `run_attempt` counter, incremented by a re-run, with the latest shown by default.
4. Commit status — a small note pinned to a commit — a state plus a context, creatable by any token with write access.
5. `continue-on-error` — keep going after a failure — a step or job setting that turns a real failure into a reported success upstream.
6. Squash merge — many commits collapsed into one new commit — a merge strategy producing a target-branch commit that no check has ever seen.

41.8 Statuses, checks, and the two APIs behind them

■ 41.8.1 PLAIN – in simple words

1. There are two separate systems for pinning a result onto a commit. They were built years apart and both are still alive.
2. The older one is the **commit status**. It is a small note with four parts: a state, a short description, a link, and a name called the context.
3. Its states are `pending`, `success`, `failure` and `error`. That is all it can say.
4. The newer one is the **check run**, part of the checks system.
5. A check run says much more: whether it is `queued`, `running` or `finished`; what the outcome was; when it started and ended; a block of output text; and notes pinned to particular lines of particular files.
6. Check runs are collected into a **check suite**, which is roughly “everything one tool reported about one commit”.
7. Both systems attach their results to a commit hash. Both draw ticks and crosses in the same place on the page. That is why they get confused.
8. Branch protection lets you name some of these results as **required**.
9. A required result means: this pull request may not merge until a result with exactly that name reports success.
10. And here is the sharp edge. A required check that never arrives blocks the merge forever.
11. It blocks because “has not arrived” is not the same as “failed”. It sits in a third state that means “we are still waiting for this”.
12. Nothing times it out. Nothing gives up. It waits, and the merge button stays grey.

■ 41.8.2 PLAIN – a picture in your head

1. Imagine a building site where a flat cannot be handed over until three inspection slips are pinned to the door.
2. Slip one is from the electrician, slip two from the plumber, slip three from the fire officer.
3. The old slips were postcards: a tick or a cross, a line of writing, nothing else.
4. The new slips are full forms: who inspected, when they started, when they finished, and photographs with arrows pointing at specific pipes.
5. The handover clerk does not judge quality. The clerk only checks that all three named slips are present and say “pass”.

6. Now suppose the fire officer was never told this flat exists.
7. There is no cross on the door. There is simply no third slip.
8. The clerk waits. Nobody is at fault, no alarm rings, and the flat is never handed over.

Where this comparison breaks: real inspectors eventually notice and telephone somebody. A required check has nobody to telephone. The rule is purely mechanical, and the only cure is a human editing the rule or arranging for the missing slip to appear.

■ 41.8.3 PLAIN – a worked example

1. A commit status, created by hand, exactly as an old-style build tool would:

```
gh api -X POST repos/OWNER/REPO/statuses/$SHA \
  -f state=success \
  -f context='ci/unit-tests' \
  -f target_url="$BUILD_PAGE_URL" \
  -f description='412 tests, 0 failures'
```

2. A check run, as posted by an app, has far more structure:

```
{
  "name": "unit-tests",
  "head_sha": "3f1d02c9d5f0a1b2c3d4e5f6...",
  "status": "completed",
  "conclusion": "failure",
  "started_at": "2026-08-13T09:41:02Z",
  "completed_at": "2026-08-13T09:53:11Z",
  "output": {
    "title": "3 failing",
    "summary": "412 tests, 3 failures, 0 skipped",
    "annotations": [
      {
        "path": "src/cart/total.ts",
        "start_line": 44,
        "annotation_level": "failure",
        "message": "expected 1200, received 1199"
      }
    ]
  }
}
```

3. The annotation is the visible difference. It is what draws a red marker on line 44 of that file inside the pull request diff.
4. Reading both systems for one commit:

```
# newer: individual check runs
gh api "repos/OWNER/REPO/commits/$SHA/check-runs" \
  --jq '.check_runs[] | {name, status, conclusion}'

# older: the rolled-up commit status
gh api "repos/OWNER/REPO/commits/$SHA/status" \
  --jq '{state, statuses: [.statuses[]].context[]}'
```

5. The rollup rule for the older API is documented and simple: the combined state is failure if any context is error or failure; pending if there are no statuses at all or any context is pending; success only if every context's latest status is success.
6. Read that middle clause again. Zero statuses rolls up to pending, not to success. Silence is treated as “not yet”, which is exactly the behaviour that makes a missing required check block forever.

■ 41.8.4 PLAIN – what is really happening inside

1. When a run starts, the service creates a check suite for that commit and one check run per job.
2. Each check run begins as queued, becomes in progress, and finally becomes completed with a conclusion.
3. The conclusion is written once, when the job ends. Until then there is no conclusion at all, only a status.
4. That two-field design is the whole reason “pending” and “failed” are different things: status says whether it is over, conclusion says how it went, and the second is empty until the first says completed.
5. Branch protection compares a list of required names against the results present on the head commit of the pull request.
6. The comparison is by name, as text. It has no idea which tool produced the result and no idea what the tool did.
7. So renaming a job in your workflow file silently breaks a required check. The old name never arrives again, and every pull request blocks.
8. This is the single commonest cause of “the repository is stuck and nobody changed the rules”. Somebody did change something: the job’s name.
9. Path filters cause the same shape of failure from the other direction. If the workflow does not start, the check does not exist, and the required name never arrives.
10. The cure is a small job that always runs and always reports the required name, whose only work is to look at the other jobs and decide.

■ 41.8.5 TECHNICAL – the engineer’s version

1. The commit status API is the older of the two and dates from the early years of the GitHub API. Create with `POST /repos/{owner}/{repo}/statuses/{sha}`. Fields: state (one of error, failure, pending, success), target_url, description, and context which defaults to the literal string default.
2. Documented limit: 1,000 statuses per SHA and context pair in a repository. Read them with `GET /repos/{owner}/{repo}/commits/{ref}/statuses` for the list, or `.../status` for the rollout.
3. In GraphQL the type is `StatusState`, with exactly five values: `ERROR`, `EXPECTED`, `FAILURE`, `PENDING`, `SUCCESS`. `EXPECTED` is the one you cannot post yourself; it is what a required-but-absent check looks like.
4. The checks API arrived with GitHub Apps in 2018 and is richer. A **check suite** is a collection of check runs created by one app for one commit. A **check run** is one individual test of that commit.
5. Only a GitHub App can create check runs. `GITHUB_TOKEN` inside Actions acts as an app installation token, which is why Actions can create them and your personal token generally cannot.
6. In GraphQL, `CheckStatusState` has six values: `QUEUED`, `REQUESTED`, `WAITING`, `PENDING`, `IN_PROGRESS`, `COMPLETED`. The REST spelling is lowercase.
7. `CheckConclusionState` has nine values. This table is the one to keep.

Conclusion	Means	Required check
success	it passed	satisfied
failure	it failed	blocks
neutral	ran, no judgement	satisfied
cancelled	stopped early	blocks
skipped	did not execute	see 41.7.5
timed_out	exceeded its limit	blocks

Conclusion	Means	Required check
action_required	needs a human	blocks
stale	superseded by GitHub	blocks
startup_failure	never got going	blocks

8. Notes on the awkward ones. `neutral` exists so a linter can report advice without blocking; it counts as not-failing. `action_required` is what an app returns when it needs you to authorize something, such as granting access.
9. `stale` can only be set by GitHub itself, and marks a result the platform has decided is no longer meaningful. `startup_failure` means the run could not begin, most often an invalid workflow file or an unavailable runner label.
10. `timed_out` on GitHub Actions arrives at `timeout - minutes` if you set one, and at the platform limit of 6 hours for a hosted job if you did not.
11. Branch protection matches required checks by name string against results on the head commit. Configure with `PATCH /repos/{owner}/{repo}/branches/{branch}/protection` or, in the newer model, with repository rulesets.
12. The permanent-block conditions, all documented, all real:

Cause	Effect
Workflow skipped by filter	check absent, stays Expected
Job name renamed	old name never arrives
App uninstalled	its checks never post
Required name never existed	typo blocks every PR

13. `[skip ci]`, `[ci skip]`, `[no ci]`, `[skip actions]` and `[actions skip]` in a commit message, or a `skip-checks: true` trailer, suppress the run entirely. On a repository with required checks that is a self-inflicted permanent block.
14. The standard workaround is a gate job that always runs:

```

ci-gate:
  if: always()
  needs: [lint, test, build]
  runs-on: ubuntu-latest
  steps:
    - name: Fail if any dependency did not succeed
      if: contains(needs.*.result, 'failure')
        || contains(needs.*.result, 'cancelled')
      run: exit 1

```

15. Require `ci-gate` and nothing else. It runs on every event that the workflow runs on, and its name is stable even when the jobs beneath it are renamed.
16. GitLab models the same idea differently: a job has `allow_failure`, a pipeline has a single status, and merge request approval rules reference the pipeline rather than individual named checks. There is no direct equivalent of a required check name, which removes this class of bug and removes the fine-grained control with it.
17. Observing tools: `gh pr checks 42`, `gh pr checks 42 --watch`, `gh api repos/OWNER/REPO/commits/$SHA/check-runs`, `gh api repos/OWNER/REPO/branches/main/protection --jq .required_status_checks`.

■ 41.8.6 WORDS – remember these

1. Commit status — the old-style note on a commit — a state, context, description and target_url posted to the statuses endpoint.
2. Context — the name of an old-style status — the string branch protection matches against, defaulting to default.
3. Check run — the new-style detailed result — an app-created record with status, conclusion, timings, output and line annotations.
4. Check suite — all check runs from one app for one commit — the grouping the interface shows as a single reporting tool.
5. Annotation — a note pinned to a line of a file — a check run output entry with path, start_line and annotation_level.
6. Required status check — a name that must report success — a branch protection rule compared by text against results on the head commit.
7. Expected — waiting for a result that never came — StatusState.EXPECTED, the state that blocks a merge indefinitely.

41.9 Merge queues

■ 41.9.1 PLAIN – in simple words

1. Here is a problem that green ticks cannot solve on their own.
2. Two people each propose a change. Each change is tested alone, and each passes.
3. Both are merged within a minute of each other. The result is broken.
4. Nothing went wrong with either test. The tests answered the questions they were asked.
5. Neither test was ever asked the only question that matters: what happens when both of these land together?
6. Chapter 39 called this a semantic conflict: two changes that do not touch the same lines, and still contradict each other.
7. One person renames a function. The other person adds a new call to the old name. No line overlaps. The program does not build.
8. A **merge queue** exists to ask the missing question before the merge, not after.
9. It works like a queue at a counter. Changes wait in a line, and only one thing is joined at a time.
10. For each change in the line, the machine builds the result you would get if that change landed on top of everything ahead of it.
11. It tests that combined result.
12. Only if that passes does the change actually land.
13. So the thing that is tested is the thing that lands, in the order it lands.
14. That is the whole idea, and it is the only mechanical way to keep a mainline that always builds.

■ 41.9.2 PLAIN – a picture in your head

1. Think of an aeroplane runway with one strip and a queue of aircraft.
2. Each pilot has already done a check on their own aircraft in the hangar. All passed.
3. But the tower does not clear them by hangar checks. The tower clears one aircraft at a time, in a known order, and re-checks the runway state each time.
4. Now add impatience. Waiting for each aircraft to finish before starting the next check is slow.
5. So the tower guesses. It assumes number one will be fine, and starts checking number two against a runway that already has number one on it.
6. It does the same for three and four. Four checks run at once, each assuming everything ahead of it succeeded.
7. If all guesses hold, everything lands in one runway’s worth of time instead of four.
8. If number two turns out to be faulty, the guesses made for three and four were built on a false assumption and must be thrown away and redone.
9. That guessing is called speculative execution, and it is the trick that makes queues fast enough to use.

Where this comparison breaks: aircraft are physical and independent; software changes interact. A faulty aircraft does not make the next one faulty, but a bad merge genuinely can invalidate every speculation behind it. Also, a runway handles one landing at a time by physics. A merge queue chooses to serialize, and could choose not to, at the cost of correctness.

■ 41.9.3 PLAIN – a worked example

1. Three pull requests are ready. The test suite takes 12 minutes.
2. The old discipline, “rebase onto the latest mainline and re-run before merging”, plays out like this.

```

0 min  A rebases onto main, starts CI
12 min  A green, A merges. main moves.
12 min  B rebases onto new main, starts CI
24 min  B green, B merges. main moves.
24 min  C rebases onto new main, starts CI
36 min  C green, C merges.
```

3. Total elapsed: 36 minutes, and three humans had to sit and watch.
4. The same three in a merge queue with speculation.

```

0 min  A, B, C enter the queue in that order
0 min  build main+A          (entry 1)
0 min  build main+A+B        (entry 2, speculative)
0 min  build main+A+B+C      (entry 3, speculative)
12 min  all three green
12 min  A, B, C merge in order. main moves once.
```

5. Total elapsed: about 12 minutes, and nobody watched anything.
6. Now the interesting case. Entry 2 fails, because B contradicts A.

```

12 min  entry 1 green, entry 3 green, entry 2 FAILED
12 min  A merges
12 min  B is removed from the queue and its author told
12 min  entry 3 is discarded: it assumed B would land
12 min  rebuild main+A+C
24 min  green, C merges
```

7. Entry 3’s result was correct about a world that never happened, so it is thrown away. That is the cost of speculation.
8. Costs compared, using 12-minute runs and a 5 percent chance that any given change fails when combined:

Approach	3 changes, all pass	Human effort
Merge and hope	12 min	none, breaks main
Rebase and re-run	36 min	three people wait
Queue, no speculation	36 min	none
Queue, speculative	12 min	none

9. The wasted machine time under speculation is real: a failure at position two discards every build behind it. Teams accept that because machine minutes are cheaper than a broken mainline and cheaper than engineers waiting.

■ 41.9.4 PLAIN – what is really happening inside

1. When a pull request is added to the queue, the service does not merge it. It creates a temporary branch.
2. That temporary branch contains the target branch, plus every queued change ahead of this one, plus this one.
3. That temporary branch is the thing your workflows run against. It is a real commit, with a real hash, and the run is filed against that hash.
4. This is why a merge queue needs its own trigger. A workflow that only listens for `pull_request` will never run for a queue entry.
5. When the run for an entry finishes green and everything ahead of it has merged, the entry is merged into the target branch.
6. If the run fails, that entry is removed from the queue, the author is told why, and the temporary branches for everything behind it are rebuilt without it.
7. Batching is the other lever. Instead of one entry per build, group five entries into one temporary branch and test the group.
8. A green batch merges five changes for the price of one build. A red batch tells you only that one of five is bad, and you must split it to find out which.
9. Splitting a failed batch in half repeatedly finds the culprit in about $\log_2(n)$ further builds. That is the same bisection idea as `git bisect` from Chapter 40.
10. Every serious queue has the same three knobs: how many entries to build at once, how many to merge at once, and how long to wait before giving up on filling a batch.

■ 41.9.5 TECHNICAL – the engineer’s version

1. The property a merge queue enforces has a name in the Rust community: the “not rocket science rule of software engineering”, a phrase attributed to Graydon Hoare, Rust’s original author. Stated plainly: automatically maintain a repository of code that always passes all the tests.
2. History. The Rust project’s `bors` bot implemented this from the early 2010s, later reimplemented as `homu` and `bors-ng`. OpenStack’s `Zuul`, in use from 2012, added speculative execution across multiple projects and called the mechanism a gate pipeline. `Zuul` v3 arrived in 2017.
3. Commercial and hosted implementations followed: `Mergify`, `Aviator`, `Graphite`, and `GitLab’s` merge trains, which appeared in `GitLab` 11.2 in 2018.
4. `GitHub’s` own merge queue entered public beta on 8 February 2023 and became generally available on 12 July 2023.

5. Mechanically, on GitHub: entries create temporary branches under a reserved prefix, `gh-readonly-queue/`, one per entry, named after the base branch and the pull request. You cannot push to them.
6. Workflows must listen for the `merge_group` event with activity type `checks_requested`, or they will not run for queue entries at all:

```
on:
  pull_request:
    branches: [main]
  merge_group:
    types: [checks_requested]
```

7. Forgetting `merge_group` is the classic first-day merge queue failure. The entry sits waiting for a required check that no workflow will ever produce, which is exactly the permanent-block condition from 41.8.
8. Documented settings, with their ranges as of August 2026:

Setting	Range	Meaning
Build concurrency	1 to 100	speculative builds at once
Maximum to merge	1 to 100	largest batch
Minimum to merge	1 to 100	smallest batch
Wait time	minutes	how long to fill a batch

9. Merge method is configurable as `merge`, `rebase` or `squash`, and applies to the whole queue rather than per pull request.
10. “Only merge non-failing pull requests” controls batch semantics. When on, every entry must pass. When off, a group may merge if the final entry passes for the combined changes, which is faster and admits a change that never passed on its own.
11. On failure, GitHub removes the entry from the queue, records the reason on the pull request timeline, recreates the temporary branch without it, and continues with the rest.
12. Cost model. With batch size b , per-run time T , and independent failure probability p per change, a batch is green with probability $(1 - p)^b$. Expected builds per merged change falls with b while all is well, and rises sharply once $b * p$ approaches 1. Most teams land on batches of 2 to 8.
13. The older discipline is still available and still correct: branch protection offers “Require branches to be up to date before merging”, which forces each author to rebase or merge the latest target and re-run before the merge button unlocks.
14. That setting has two costs a queue does not: it serializes humans rather than machines, and it has a race. Between your last green run and your click, somebody else can merge. The window is small but not zero.
15. It also scales badly. On a busy repository, the mainline moves faster than your test suite runs, so you can never get to the front of the line. Teams hit this at roughly $\text{merges per hour} \times \text{test minutes} > 60$.
16. Comparison of approaches:

Approach	Mainline safety	Cost
Merge on green PR	not guaranteed	cheapest
Up-to-date required	guaranteed	humans serialize
Queue, serial	guaranteed	slow, no waste

Approach	Mainline safety	Cost
Queue, speculative	guaranteed	fast, wastes builds

17. Observing tools: `gh pr merge --auto`, `gh api repos/OWNER/REPO/merge-queue` for queue state, the `merge_group` payload inside the run, and `git ls-remote origin 'refs/heads/gh-readonly-queue/*'` to see live entries.

■ 41.9.6 WORDS – remember these

1. Merge queue — a line of changes joined one at a time — a serializing gate that tests the prospective merged state before merging.
2. Semantic conflict — two changes that clash without touching the same lines — a textually clean merge that fails to build or fails tests.
3. Speculative execution — testing entries assuming the ones ahead succeed — parallel builds of stacked prospective states, discarded on any failure.
4. Batch — several entries tested as one — a group merged together, requiring bisection to locate a culprit when it fails.
5. `merge_group` — the event a queue entry raises — the GitHub Actions trigger with activity type `checks_requested`.
6. Merge train — GitLab’s name for the same idea — sequential pipelines on the projected result of merging each queued request.
7. Not rocket science rule — always keep the mainline passing — the invariant a gating queue mechanically enforces.

41.10 mergeStateStatus, value by value

■ 41.10.1 PLAIN – in simple words

1. When you ask GitHub about a pull request, one field tells you why the merge button is or is not available. It is called `mergeStateStatus`.
2. It has exactly eight possible values, and they are not all the same kind of thing, which is why people misread them.
3. Some values are about the code itself: can these two branches be joined at all?
4. Some are about the rules: the code is joinable, but a policy says not yet.
5. One is about the pull request’s own state: it is a draft.
6. And one is about GitHub itself: I have not worked it out yet, ask again.
7. The reader saw three of them in one session: `CLEAN`, `BLOCKED` and `UNKNOWN`.
8. Two of these are worth learning before the rest.
9. `UNKNOWN` almost never means something is wrong. It means the answer is still being computed. The correct response is to ask again in a second.
10. `BLOCKED` does not mean a merge conflict. It means the branches join fine and a protection rule is not satisfied.
11. Getting those two right removes most of the confusion this field causes.

■ 41.10.2 PLAIN – a picture in your head

1. Think of a clerk at a counter deciding whether your form can be filed.
2. `DIRTY` is: your form physically will not fit in the folder. The paper is the wrong shape. Fix the paper.
3. `BEHIND` is: your form is on an old version of the template. Copy it onto the current template.

4. **BLOCKED** is: the form is perfect and the folder is ready, but a signature is missing. Nothing is wrong with your paper.
5. **UNSTABLE** is: everything is signed, but one optional report attached to it came back with a complaint. The clerk will still file it if you insist.
6. **DRAFT** is: you wrote “draft” across the top yourself. The clerk is obeying you.
7. **CLEAN** is: file it.
8. **UNKNOWN** is: the clerk has not finished reading it yet. Come back in a moment.

Where this comparison breaks: a clerk gives you one reason. `mergeStateStatus` gives you the first blocking reason it finds, in an order GitHub does not publish. A pull request can be behind and blocked and have failing checks all at once, and you will be shown a single word. Never treat that word as a complete list of what is wrong.

■ 41.10.3 PLAIN – a worked example

1. Asking for the field, together with the things that explain it:

```
gh pr view 42 --json \
  number,mergeable,mergeStateStatus,isDraft,headRef0id,\
  reviewDecision
```

2. A typical answer while GitHub is still thinking:

```
{
  "number": 42,
  "mergeable": "UNKNOWN",
  "mergeStateStatus": "UNKNOWN",
  "isDraft": false,
  "headRef0id": "3f1d02c9d5f0a1b2c3d4e5f60718293a4b5c6d7e",
  "reviewDecision": "APPROVED"
}
```

3. Wait one or two seconds and ask again. This is not a workaround. It is the documented way the field behaves.

```
{
  "mergeable": "MERGEABLE",
  "mergeStateStatus": "BLOCKED",
  "reviewDecision": "APPROVED"
}
```

4. Read those two lines together, because alone each is misleading.
5. `mergeable: MERGEABLE` says the branches join without conflict.
6. `mergeStateStatus: BLOCKED` says a rule is unsatisfied. The code is fine. The policy is not.
7. To find out which rule, ask the checks, not this field:

```
gh pr checks 42
gh api repos/OWNER/REPO/branches/main/protection \
  --jq .required_status_checks.contexts
```

8. Nine times out of ten the answer is a required check sitting in `Expected`, which is exactly the permanent block from 41.8.

■ 41.10.4 PLAIN – what is really happening inside

1. GitHub does not keep a permanent answer to “can this merge?”.
2. It computes one on demand, in the background, by attempting a test merge of the head into the base.
3. That computation takes time, and the API answers immediately whether or not it has finished.
4. So the first response after any change to either branch is very often UNKNOWN, with the REST equivalent field mergeable set to null.
5. Asking again triggers or picks up the computation. This is documented, and it is why every serious script polls this field rather than reading it once.
6. Once the test merge succeeds, GitHub knows the branches join, and moves on to the rules.
7. It then reports the first obstacle it finds: draft status, protection rules, check results, or nothing at all.
8. This is why the values are a mixture of categories. The field is not a classification of the pull request. It is a report of the first thing stopping the button from lighting up.

■ 41.10.5 TECHNICAL – the engineer’s version

1. mergeStateStatus is a GraphQL enum on the PullRequest type. GitHub’s documented values and descriptions, with the action each implies:

Value	GitHub’s description	Do this
BEHIND	head ref is out of date	update the branch
BLOCKED	the merge is blocked	find the unmet rule
CLEAN	mergeable, status passing	merge it
DIRTY	merge commit not creatable	resolve conflicts
DRAFT	blocked, PR is a draft	mark ready for review
HAS_HOOKS	mergeable, pre-receive hooks	merge, hooks may reject
UNKNOWN	state cannot be determined	re-query, conclude nothing
UNSTABLE	mergeable, status not passing	fix or override

2. UNKNOWN, precisely. The mergeability computation is asynchronous. In REST, GET /repos/{owner}/{repo}/pulls/{number} returns mergeable: null until the background job completes, and the documentation instructs you to retry. In GraphQL the same condition surfaces as UNKNOWN.
3. Therefore: UNKNOWN is not an error, not a conflict, and not a policy failure. It is “not yet”. Poll with a short delay, a few times, before treating it as anything else. Concluding anything from a single UNKNOWN is the commonest mistake made with this field.
4. BLOCKED, precisely. It means the branches merge cleanly and a branch protection or ruleset requirement is unmet. It is not a merge conflict. DIRTY is the merge conflict value.
5. Common causes of BLOCKED, none of which are visible in the field itself: a required review missing, a requested change outstanding, a required status check absent or failing, a CODEOWNERS review missing, a required conversation unresolved, or a required deployment or signature rule.
6. UNSTABLE versus BLOCKED is the distinction people most often get backwards. UNSTABLE means a check is not passing but that check is not required, so a user with permission can still merge. BLOCKED means a rule forbids it outright.
7. BEHIND appears only when the repository has “Require branches to be up to date before merging” enabled. Without that setting, an out-of-date branch reports CLEAN or UNSTABLE instead, and merges happily.

8. HAS_HOOKS is specific to GitHub Enterprise Server, where an administrator can install pre-receive hooks. It means everything is fine but a server-side hook still gets the final word at push time, so a merge can still be rejected after you click.
9. DIRTY means Git cannot construct the merge commit without conflict resolution. Nothing in continuous integration will fix it. A human must rebase or merge and resolve.
10. Companion fields you should always fetch alongside it, because mergeStateStatus alone is not diagnostic:

Field	Tells you
mergeable	conflict-free or not
reviewDecision	approved, changes needed
statusCheckRollup	per-check results
isDraft	explains DRAFT

11. On some GitHub Enterprise Server versions, mergeStateStatus sits behind a schema preview and requires the Accept header application/vnd.github.merge-info-preview+json. The gh CLI sets what it needs; a hand-written GraphQL client may not.
12. Practical polling pattern, which is what a release script should contain:

```
for i in 1 2 3 4 5; do
  S=$(gh pr view 42 --json mergeStateStatus \
    --jq .mergeStateStatus)
  [ "$S" != "UNKNOWN" ] && break
  sleep 2
done
echo "state: $S"
case "$S" in
  CLEAN|HAS_HOOKS) echo "safe to merge" ;;
  BLOCKED) echo "a rule is unmet; run gh pr checks" ;;
  UNSTABLE) echo "a non-required check is failing" ;;
  BEHIND) echo "update branch and re-run CI" ;;
  DIRTY) echo "resolve conflicts by hand" ;;
  DRAFT) echo "mark ready for review" ;;
  UNKNOWN) echo "still computing; do not conclude" ;;
esac
```

13. GitHub’s own documentation notes that the REST field mergeable_state, the lowercase sibling of this enum, has historically been treated as unofficial and subject to change. Prefer the GraphQL enum, and prefer reading the underlying facts over branching on this one word.

■ 41.10.6 WORDS – remember these

1. mergeStateStatus — why the merge button is or is not lit — a GraphQL enum on PullRequest with eight values.
2. mergeable — can the branches be joined at all — a separate tri-state field, null or UNKNOWN while still computing.
3. CLEAN — nothing is in the way — mergeable with passing commit status.
4. BLOCKED — a rule says no, not a conflict — branch protection or ruleset requirement unmet.
5. UNSTABLE — mergeable but something non-required is failing — merge is permitted, quality is not vouched for.
6. DIRTY — the branches genuinely conflict — Git cannot construct the merge commit.

7. **BEHIND** — your branch is on an old base — only reported when “require branches to be up to date” is enabled.
8. **UNKNOWN** — GitHub has not finished computing — re-query; it carries no information about the pull request.

41.11 What a good pipeline actually does, in order

■ 41.11.1 PLAIN – in simple words

1. A pipeline is a list of checks in a chosen order. The order is not decoration.
2. The rule is: cheapest and most likely to fail goes first.
3. A check that takes 8 seconds and catches a typo should never run after a check that takes 20 minutes.
4. So a sensible order looks like this.
5. **Lint**: read the code without running it, and complain about style and obvious mistakes. Seconds.
6. **Build**: turn source into something runnable. If it does not build, nothing after this matters.
7. **Unit tests**: test small pieces on their own, with nothing real attached. Fast, many, precise about where the fault is.
8. **Integration tests**: start real pieces together, a database, a queue, and check they talk correctly. Slower, fewer, vaguer about where the fault is.
9. **Security scanning**: look for known-bad dependencies, leaked keys, and dangerous code patterns.
10. **Artifact build**: package the thing you will actually ship, once, and keep it.
11. **Deploy to staging**: put that exact package on a copy of production.
12. **Smoke tests**: a handful of checks against the running copy. Does the home page load? Can one user log in?
13. **Production**: ship the same package that passed everything above.
14. Notice the package is built once and reused. Rebuilding before production would mean shipping something nobody tested.

■ 41.11.2 PLAIN – a picture in your head

1. Think of security at an airport, in the order they actually use.
2. First a glance at your ticket at the door. One second, catches most people who are in the wrong building.
3. Then the document check. Thirty seconds.
4. Then the bag scanner. Two minutes.
5. Then, for a few people, a hand search. Ten minutes.
6. Nobody hand-searches everybody first. It would be correct and it would be useless, because the queue would never move.
7. Each stage is more expensive and catches fewer people than the one before.

Where this comparison breaks: airport stages are independent, and a pipeline’s stages are not. A build failure makes every later stage meaningless, so a pipeline stops, while an airport keeps processing the rest of the queue. Also, the airport cannot rerun you from a cache; a pipeline can and should.

■ 41.11.3 PLAIN – a worked example

1. Real timings from a medium web project. Yours will differ; the shape holds.

Stage	Time	Catches
Lint	25 s	style, unused vars
Build	1 m 40 s	type and syntax errors
Unit tests	3 m	logic faults
Integration	7 m	wiring, SQL, contracts
Security scan	2 m	known bad dependencies
Package	1 m 10 s	nothing, it produces
Staging deploy	2 m	config and secrets faults
Smoke tests	40 s	it is alive

2. Run in strict order, that is about 18 minutes. Too slow.
3. Run lint, unit tests and the security scan in parallel after the build, and the critical path drops to about 12 minutes.
4. Caching the dependency download saves roughly 60 to 90 seconds of that on every single run.
5. A worked cache and artifact pair in GitHub Actions:

```

- uses: actions/cache@v4
  with:
    path: ~/.npm
    key: npm-${{ hashFiles('package-lock.json') }}
    restore-keys: npm-

- run: npm ci && npm run build

- uses: actions/upload-artifact@v4
  with:
    name: app-${{ github.sha }}
    path: dist/
    retention-days: 14

```

6. Note the cache key. It is a hash of the lock file, so a dependency change makes a new key automatically, and a stale cache can never be used.
7. Note the artifact name. It carries the commit SHA, so the package you deploy can be traced back to the exact commit that produced it.

■ 41.11.4 PLAIN – what is really happening inside

1. Two words that sound alike and are not: cache and artifact.
2. A **cache** is a speed trick. It holds things you could recreate, like downloaded dependencies. Losing it costs time and nothing else.
3. An **artifact** is a product. It holds the thing you built and will ship. Losing it means you cannot deploy without rebuilding.
4. Never treat a cache as a source of truth, and never rebuild an artifact between test and deploy.
5. **Parallelism** splits work across machines. Four machines running a quarter of the tests each finish in roughly a quarter of the time, plus the fixed startup cost of each machine.
6. That fixed cost is why splitting a 40-second suite into eight parts makes it slower, not faster.
7. **Matrix builds** are parallelism over configurations rather than over test files: the same suite on three operating systems and two language versions.
8. **Fail-fast** decides what happens when one parallel branch fails. On, the siblings are cancelled, saving money. Off, they finish, giving you the full picture.
9. Use fail-fast on for pull requests, where speed matters, and off for nightly matrix runs, where the full picture matters.

10. A **flaky test** is a test that passes and fails on the same code, with no change in between.
11. Flaky tests do something worse than waste time. They teach people that red means “press the button again”.
12. Once a team believes that, red stops carrying information, and the whole pipeline becomes decoration. This is the way continuous integration dies.

■ 41.11.5 TECHNICAL – the engineer’s version

1. The stage order above is the pipeline described in Continuous Delivery by Jez Humble and David Farley in 2010, where it is called a deployment pipeline, with the commit stage first and later stages progressively slower and more production-like.
2. The proportions come from the test pyramid, named by Mike Cohn in Succeeding with Agile in 2009: many fast unit tests, fewer integration tests, very few end-to-end tests.
3. The 10-minute build rule comes from Extreme Programming and is repeated in the Continuous Delivery literature. Past roughly 10 minutes, developers stop waiting for the result and context-switch, and the feedback loop that justifies the whole apparatus is gone.
4. GitHub Actions cache limits as documented in August 2026: 10 GB total per repository, with least-recently-used eviction, and entries removed after 7 days without access. Cache is keyed by an exact string, with restore-keys giving ordered prefix fallbacks.
5. Artifact retention on GitHub Actions defaults to 90 days, configurable from 1 to 90 days for public repositories and 1 to 400 days for private and enterprise repositories.
6. Caches are scoped by branch: a run on a branch can read caches from that branch and from the default branch, but not from unrelated branches. This is a deliberate isolation boundary, since a cache is a writable path shared between runs and therefore a cross-branch attack surface.
7. Security scanning splits into named categories, and mixing them up leads to thinking you are covered when you are not:

Kind	Looks at	Example tools
SAST	your source code	CodeQL, Semgrep
SCA	your dependencies	Dependabot, Trivy
Secret scanning	committed keys	Gitleaks, TruffleHog
DAST	the running system	OWASP ZAP

8. CodeQL came to GitHub through the acquisition of Semmle in September 2019. Dependabot came through the acquisition of Dependabot in May 2019. Both are now built into the platform.
9. Software bills of materials have two mainstream formats: SPDX, whose version 2.2.1 became ISO/IEC 5962:2021, and CycloneDX, standardized as ECMA-424 in 2024. Producing one is now a common late pipeline stage.
10. Flakiness, with a real figure: Google’s testing blog reported in May 2016 that almost 16 percent of their roughly 4.2 million tests showed some level of flakiness, and that flaky failures consumed a substantial share of engineering attention. Treat the exact percentage as specific to Google, and the phenomenon as universal.
11. Standard flakiness controls, in increasing order of honesty: retry the test and hope; quarantine it into a non-blocking suite and file a bug; fix the root cause, which is nearly always shared state, real time, real network, or unordered iteration.
12. Retrying without quarantining is the trap. It hides the signal and lets the flake rate grow until the suite is useless.

13. Required versus optional checks, restated in terms of 41.8: a required check must appear and must report success. An optional check appears and is advisory, and produces UNSTABLE rather than BLOCKED when it fails.
14. A workable policy: require the gate job, the build, and the unit tests. Leave slow end-to-end suites, performance benchmarks and advisory linters optional, and review them on a schedule instead.
15. Test time budgets, as targets rather than laws:

Stage	Budget	If exceeded
Lint plus build	under 3 min	cache, incremental build
Unit tests	under 5 min	shard across runners
Full PR pipeline	under 10 min	move work to nightly
Nightly full matrix	under 2 h	reduce matrix cells

16. The four DORA metrics from the 2018 book Accelerate by Nicole Forsgren, Jez Humble and Gene Kim are the standard way to judge whether the pipeline is actually helping: deployment frequency, lead time for changes, change failure rate, and time to restore service. Note that two of the four are about recovery, not prevention.
17. Observing tools: `gh run view --log`, the timing breakdown on a run page, `gh api repos/OWNER/REPO/actions/cache/usage`, and any test reporter that emits JUnit XML for trend analysis.

■ 41.11.6 WORDS – remember these

1. Deployment pipeline — the ordered set of checks from commit to production — the staged model set out in Continuous Delivery, 2010.
2. Cache — a recreatable speed-up — keyed storage of dependencies, safe to lose, never a source of truth.
3. Artifact — the built product you will ship — a retained output passed between jobs and to deployment, built once.
4. Matrix build — the same job across configurations — a cross product of variables expanded into parallel jobs.
5. Fail-fast — cancel siblings on first failure — a matrix strategy trading diagnostic completeness for speed and cost.
6. Flaky test — passes and fails on identical code — a non-deterministic test, usually caused by shared state, timing or ordering.
7. SAST, SCA, DAST — code scan, dependency scan, running-system scan — static analysis, software composition analysis, dynamic testing.
8. Test pyramid — many small tests, few large ones — the proportional model named by Mike Cohn in 2009.

41.12 Deployment strategies

■ 41.12.1 PLAIN – in simple words

1. You have a package that passed everything. Now you must put it in front of real users. How you do that is the deployment strategy.
2. **Recreate**: stop the old version, start the new one. Simple, and there is a gap where nothing works.
3. **Rolling**: you have ten copies running. Replace them a few at a time. No gap, but for a while both versions are live at once.

4. **Blue-green:** run two complete environments. Blue is serving users. Deploy to green, test it, then switch all traffic to green in one moment. Blue stays there, ready to take traffic back.
5. **Canary:** send 1 percent of users to the new version. Watch the error rate. If it looks fine, go to 5 percent, then 25, then everyone.
6. **Feature flags:** ship the new code to everyone but keep it switched off, then switch it on for chosen users later, without deploying anything.
7. That last one is the important idea, so say it clearly.
8. Deploying is putting code on machines. Releasing is letting users reach it.
9. Feature flags separate those two things, so a deploy stops being a scary event and becomes routine.
10. And turning a feature off is instant, while undoing a deploy is not.

■ 41.12.2 PLAIN – a picture in your head

1. Picture changing the menu at a restaurant with a hundred tables.
2. Recreate: close the restaurant, change everything, reopen. Everyone waits outside.
3. Rolling: change the menus table by table. Nobody waits, but for twenty minutes two menus exist and the kitchen must cook from both.
4. Blue-green: build an identical second dining room, set it up completely, check it, then walk everyone across in one go. Keep the old room furnished for a week in case.
5. Canary: give the new menu to two tables, watch what they send back, then widen it.
6. Feature flags: print every dish on every menu from the start, but mark the new ones “ask the waiter”. Turning them on is a word to the staff, not a reprint.

Where this comparison breaks: menus have no state. Software has databases, and a database cannot be walked across to a new room. That single difference is what makes rollback hard, and it is the subject of 41.12.4.

■ 41.12.3 PLAIN – a worked example

1. A service with 12 identical copies, a 90-second start time, and a rolling update replacing 25 percent at a time.
2. 12 copies, 25 percent surge means 3 new copies start while at most 3 old ones go away.
3. Four waves of 3, each about 90 seconds plus a health check, is roughly 7 to 8 minutes to fully roll out.
4. During those 8 minutes, both versions serve real traffic. Both must be able to read the same data and answer the same requests.
5. Compare the same release under four strategies:

Strategy	Downtime	Rollback speed
Recreate	60 to 120 s	full redeploy
Rolling	none	another full roll
Blue-green	none	seconds, flip back
Canary	none	seconds, shift traffic

6. Blue-green and canary buy fast rollback with double the running cost during the change. That is the trade, stated honestly.
7. A Kubernetes rolling update is configured with two numbers:

```
strategy:
  type: RollingUpdate
```

```
rollingUpdate:
  maxSurge: 25%
  maxUnavailable: 25%
```

8. `maxSurge` is how many extra copies may exist above the target. `maxUnavailable` is how many may be missing. Both default to 25 percent.
9. Setting `maxUnavailable: 0` guarantees no reduction in capacity and requires spare room to surge into.

■ 41.12.4 PLAIN - what is really happening inside

1. Rollback sounds symmetric. It is not.
2. Code rollback is easy: the old package still exists, so put it back.
3. Data rollback is often impossible, because the new version has already written data in a new shape.
4. Suppose the release renamed a column from `phone` to `phone_number` and copied the values across.
5. New code writes to `phone_number`. Users create records for twenty minutes.
6. Now roll the code back. The old code reads `phone`, which no longer receives writes. Those twenty minutes of phone numbers are invisible.
7. Nothing crashed. The data is simply wrong, which is worse.
8. The discipline that fixes this is called expand and contract, or parallel change, and it takes three releases instead of one.
9. **Expand**: add `phone_number` while keeping `phone`. Write to both. Read from `phone`. Deploy. This release is safe to roll back.
10. **Migrate**: switch reads to `phone_number`. Still writing both. Deploy. Still safe to roll back.
11. **Contract**: stop writing `phone`, then drop it, in a later release, once you are certain you will not roll back past this point.
12. The rule underneath: every deployment must be compatible with the version before it and the version after it, because during any rolling update both are running at once.
13. That rule is why database changes are the hard part of continuous deployment, and why teams that ignore it get exactly one bad night and then adopt it.

■ 41.12.5 TECHNICAL - the engineer's version

1. Blue-green deployment as a named practice was popularized by Dan North and Jez Humble and set out in Continuous Delivery in 2010. Canary releasing takes its name from the caged birds carried into coal mines to detect gas.
2. Feature toggles were described in detail in Pete Hodgson's 2016 article Feature Toggles on Martin Fowler's site, which classifies them into release toggles, experiment toggles, ops toggles and permission toggles, with very different lifetimes.
3. Flickr's 2009 engineering post Flipping Out is the widely cited early description of running continuous deployment behind flags, with more than ten deploys a day.
4. Strategy comparison with the properties that actually decide the choice:

Strategy	Extra capacity	Both versions live
Recreate	none	no
Rolling	small surge	yes, briefly
Blue-green	100 percent	at the switch only
Canary	small	yes, for a long time

5. Canary requires observability that most teams underestimate: per-version metrics. If your error rate is not broken down by build, a canary tells you nothing you did not already know.
6. Typical canary schedule: 1 percent for 10 minutes, 5 percent for 20 minutes, 25 percent for 30 minutes, then 100 percent, with automatic rollback on any guard metric breach. Guard metrics are usually error rate, latency at the 99th percentile, and a business counter such as checkouts per minute.
7. Kubernetes supports Recreate and RollingUpdate natively; blue-green and canary are built on top with service selectors, ingress weights, or a service mesh. Argo Rollouts and Flagger are the common controllers for this.
8. Expand and contract is documented as the ParallelChange pattern by Danilo Sato in 2014. Its formal requirement is that every schema state supports both the previous and the next application version, sometimes called N-1 compatibility.
9. Migrations that are not reversible in practice: dropping a column, dropping a table, narrowing a type, adding a NOT NULL constraint without a default, and any data transformation that loses the original value. Plan these as separate releases, never bundled with a behaviour change.
10. GitHub Actions models targets as **environments**, which carry their own secrets and variables plus protection rules:

Protection	Effect
Required reviewers	up to 6 approvers
Wait timer	delay up to 43,200 min
Branch policy	only named branches deploy
Custom rules	third-party gates

11. 43,200 minutes is 30 days, the documented maximum wait timer. A wait timer is a real control: it is the mechanical version of “let it bake in staging overnight”.
12. A job binds to an environment with `environment: production`, at which point the environment’s secrets become available and its protection rules apply before the job starts, not during it.
13. GitLab expresses the same idea with `environment: plus when: manual`, and protected environments with approval rules. Argo CD expresses it as a sync policy, manual or automated, per application.
14. Observing tools: `gh api repos/OWNER/REPO/deployments`, `gh api repos/OWNER/REPO/environments`, `kubectl rollout status`, `kubectl rollout undo`, and per-version dashboards in your metrics system.

■ 41.12.6 WORDS – remember these

1. Recreate — stop the old, start the new — a strategy with a deliberate outage window and no version overlap.
2. Rolling update — replace instances a few at a time — controlled by surge and unavailability limits, with both versions live.
3. Blue-green — two full environments, one switch — instant cutover and instant rollback at double running cost.
4. Canary — a small slice of traffic first — progressive delivery gated on per-version guard metrics.
5. Feature flag — code shipped but switched off — runtime toggle decoupling deploy from release, reversible in milliseconds.
6. Expand and contract — change the schema in compatible steps — the ParallelChange pattern giving N-1 compatibility across releases.

7. Environment — a named deployment target with rules — a scope holding secrets, reviewers, wait timers and branch policies.

41.13 Secrets in CI

■ 41.13.1 PLAIN – in simple words

1. A pipeline needs passwords. It signs in to a registry, uploads to a server, talks to a cloud account.
2. Those values cannot live in the repository, because everyone who can read the code can read them, forever, in every clone.
3. So they are stored separately, in a box the service keeps locked, and handed to the machine only while the job runs.
4. Inside the job they appear as environment variables, which are just named values the running program can read.
5. When the job ends, the machine is destroyed and the values go with it.
6. Two rules matter more than everything else.
7. Rule one: never print a secret. Not to the log, not in an error message, not in a debug dump.
8. The service tries to help by blacking out anything matching a known secret, but that is text matching. Change the text even slightly and the blackout fails.
9. Encode a key in base64 and print it, and it goes to the log in full, because the blacked-out pattern no longer matches.
10. Rule two: a stranger’s pull request gets no secrets at all.
11. That is why a build from an outside contributor can test the code and cannot publish anything.
12. And the modern advice goes further: stop storing long-lived cloud passwords at all.
13. Instead the job proves who it is, and the cloud hands it a temporary key that expires in an hour.
14. Nothing long-lived exists to steal, which changes the shape of the whole problem.

■ 41.13.2 PLAIN – a picture in your head

1. Think of a building that contractors visit.
2. The old way: give each contractor a copy of the master key. It works forever, it can be copied, and if one is lost nobody knows.
3. The new way: the contractor shows an identity card at reception. Reception checks it against a list, and issues a visitor badge valid for one hour, for one floor.
4. The badge cannot be reused tomorrow. Losing it costs almost nothing.
5. Reception never gives out the master key, because there is no need.
6. That is the difference between a stored cloud password and identity federation.

Where this comparison breaks: the contractor’s identity card is a physical thing they carry. In continuous integration the identity is issued fresh by the platform for each job, and its trustworthiness rests entirely on the platform not lying about which repository is asking. So you must pin the trust rule down to an exact repository and branch, or any repository on the platform can walk in with a valid card.

■ 41.13.3 PLAIN – a worked example

1. The wrong way, and it is written like this in real repositories:

```
- run: |  
    echo "Token is ${ secrets.API_TOKEN }"  
    curl -H "Authorization: Bearer $TOKEN" ...
```

2. Two faults. The value is interpolated into the script text, so it can appear in an error trace. And it is printed on purpose.
3. The right way passes secrets through the environment, never into the script body:

```
- env:
  API_TOKEN: ${ secrets.API_TOKEN }
run: |
  curl -sS -H "Authorization: Bearer $API_TOKEN" \
    "$API_BASE/v1/publish"
```

4. Now the shell receives a variable name. The value never becomes part of the command text that appears in logs and traces.
5. The identity-federation version needs no stored password at all:

```
permissions:
  id-token: write
  contents: read
steps:
  - uses: aws-actions/configure-aws-credentials@v4
    with:
      role-to-assume: arn:aws:iam::111122223333:role/ci
      aws-region: ap-south-1
```

6. There is no `secrets.AWS_SECRET_ACCESS_KEY` anywhere. The `id-token: write` permission lets the job ask GitHub for a signed identity document, and the cloud exchanges it for credentials lasting about an hour.
7. Pinning a third-party action to an exact commit, with the version in a comment so tools can still update it:

```
- uses: actions/checkout@11bd719 # v4.2.2
```

8. That short form is for the page. In a real file, write the full 40-character hash, because a tag can be moved and a full hash cannot.

■ 41.13.4 PLAIN – what is really happening inside

1. A secret is encrypted before it leaves your browser, stored encrypted, and decrypted only at the moment a job that is allowed to see it starts.
2. The service also registers each secret value with the log processor, which scans every line and replaces exact matches with asterisks.
3. That masking is genuinely useful and genuinely shallow. It matches strings.
4. It does not know that `base64 -w0 key.pem` produces a different string containing the same secret. It does not know that printing a JSON object containing the key is the same disclosure.
5. This is exactly how the March 2025 attack on a popular third-party action worked: the malicious code read secrets out of the runner process and printed them in an encoded form, and masking did not catch it.
6. Fork pull requests get no secrets because the code being tested is not yours. Section 41.5 covered why, and this is the other half of the same rule.
7. For identity federation the sequence is short and worth knowing.
8. The job asks GitHub for a signed token describing itself: which repository, which branch, which environment, which workflow.
9. The cloud provider has been configured in advance to trust tokens signed by GitHub, but only when the description matches a rule you wrote.

10. If it matches, the cloud issues short-lived credentials. If it does not, the request is refused and nothing is leaked.
11. The whole exchange takes under a second and leaves nothing behind.

■ 41.13.5 TECHNICAL – the engineer’s version

1. GitHub encrypts secrets client-side with a libsodium sealed box before upload, stores them encrypted, and decrypts them only when injecting into a runner. Documented limits as of August 2026: 48 KB per secret, 1,000 organization secrets, 100 repository secrets, 100 environment secrets.
2. A value larger than 48 KB must be encrypted into a file committed to the repository, with only the passphrase stored as a secret. This is the documented workaround, not a hack.
3. Masking is implemented by registering values with the log service. You can register more at run-time with the workflow command `::add-mask::value`, which is how you protect a secret derived inside the job.
4. Masking limitations, all real: it does not survive transformation such as base64 or URL encoding, it does not apply to values split across lines, and it does not protect artifacts, only logs.
5. Scoping: repository secrets, organization secrets with repository allow lists, and environment secrets which are only available to jobs bound to that environment and only after its protection rules pass.
6. `GITHUB_TOKEN` is an automatically issued installation token, valid for the duration of the job, invalidated at the end. Declare `permissions:` at workflow or job level and grant the minimum. For newly created repositories and organizations GitHub changed the default to read-only in 2023; older ones may still default to write, so check rather than assume.
7. OIDC federation, precisely. GitHub Actions gained support in a changelog entry on 27 October 2021 and was declared generally available in November 2021.
8. The job requests a JSON Web Token from the endpoint in `ACTIONS_ID_TOKEN_REQUEST_URL`, authenticating with `ACTIONS_ID_TOKEN_REQUEST_TOKEN`, both injected only when `permissions: id-token: write` is set.
9. The issuer is `https://token.actions.githubusercontent.com`. The claim that matters is `sub`, whose shape is one of:


```
repo:OWNER/REPO:ref:refs/heads/main
repo:OWNER/REPO:environment:production
repo:OWNER/REPO:pull_request
```
10. On AWS the exchange is `sts:AssumeRoleWithWebIdentity` against an IAM OIDC identity provider, producing credentials of at most one hour by default. On Azure it is a federated credential on an app registration. On Google Cloud it is Workload Identity Federation.
11. The critical misconfiguration is a trust policy that checks the issuer but not the sub claim, or checks it with a wildcard. That configuration lets any repository on GitHub assume your role. Always match the full sub, including the ref or environment.
12. Third-party action supply chain risk, with a real case. In March 2025 the action `tj-actions/changed-files` was compromised: its version tags were retargeted to a malicious commit that read secrets out of runner memory and printed them into the workflow log.

Fact about that incident	Value
Advisory	CVE-2025-30066
Affected versions	up to 45.0.7

Fact about that incident	Value
Related compromise	CVE-2025-30154
United States CISA alert	18 March 2025

13. The mechanism deserves attention: nothing was pushed to your repository. A tag in someone else’s repository was moved. `uses: owner/action@v4` resolves that tag at run time, on every run, with no record of what it pointed at yesterday.
14. Pinning to a full 40-character commit SHA removes that entire class of attack, because a SHA names immutable content. Moving a tag cannot change what a SHA resolves to.

Reference form	Mutable	Verdict
@main	yes, any push	never
@v4	yes, tag moves	avoid
@v4.2.2	yes, tag moves	avoid
@11bd719... full SHA	no	correct

15. Pinning has a cost: you stop receiving security fixes automatically. Dependabot understands SHA pins with a trailing version comment and raises pull requests to move them, which restores updates while keeping the review step.
16. Earlier incidents in the same family, worth knowing as precedent: the Codecov bash uploader compromise in April 2021, where a modified script exfiltrated environment variables from customers’ CI jobs, and the SolarWinds Orion build system compromise disclosed in December 2020.
17. Other controls that matter: run untrusted work on ephemeral, isolated runners; never use self-hosted runners on public repositories; set `permissions: {}` and add back only what a job needs; and enable secret scanning with push protection so a committed key is rejected at push time.
18. Observing tools: `gh secret list`, `gh api repos/OWNER/REPO/actions/permissions`, `gh api repos/OWNER/REPO/actions/permissions/selected-actions`, and a repository policy allowing only actions from verified creators or a pinned allow list.

■ 41.13.6 WORDS – remember these

1. Secret — a value the pipeline needs and must not reveal — encrypted at rest, injected as an environment variable for one job.
2. Masking — blacking out secret values in logs — exact string replacement by the log processor, defeated by any encoding.
3. `::add-mask::` — register a value for masking at runtime — a workflow command for secrets derived inside a job.
4. OIDC federation — proving identity instead of holding a key — a signed JWT exchanged for short-lived cloud credentials.
5. sub claim — which repository, branch or environment is asking — the field the cloud trust policy must match exactly.
6. `id-token: write` — permission to request an identity token — required before the OIDC endpoint variables are injected.
7. SHA pinning — naming an action by immutable content — `uses: owner/act@<40 hex>`, immune to tag retargeting.
8. Supply chain attack — compromising something you depend on — as in the March 2025 `tj-actions/changed-files` incident, CVE-2025-30066.

41.14 Reading a failed run like an engineer

■ 41.14.1 PLAIN – in simple words

1. A run went red. Most people open it, read the last line, and guess.
2. The last line is almost never the fault. It is usually the summary of the fault, several steps downstream.
3. Do this instead, in order.
4. Find which **step** failed. Not which job, not which line. The step.
5. Read that step from its beginning, not from its end.
6. The real message is usually in the first few lines after the step starts, and everything below it is consequence.
7. Ask one question before anything else: does this fail on my machine too?
8. If it fails locally, this is a normal bug and continuous integration did its job. Fix it and stop reading.
9. If it passes locally and fails in the pipeline, you have a different problem, and guessing will waste an hour.
10. Then the difference between the two environments is the fault, and there is a short list of things that difference can be.
11. Work through that list in order rather than changing things at random.
12. And if you still cannot see it, re-run with debug output switched on, which prints what the machine was actually doing.

■ 41.14.2 PLAIN – a picture in your head

1. A car will not start in your driveway but starts fine at the garage.
2. A bad mechanic replaces parts until it works.
3. A good mechanic writes down every difference between the driveway and the garage: colder, on a slope, different fuel, different battery charge.
4. Then tests the differences one at a time.
5. The car is the same. The environment is not. The fault lives in the gap.

Where this comparison breaks: you cannot drive a build to the garage, but you can bring the garage to the build. A pipeline runs a published machine image, so you can start the identical image locally, which is a luxury no mechanic has.

■ 41.14.3 PLAIN – a worked example

1. Getting the logs without opening a browser:

```
# which runs failed recently
gh run list --status failure --limit 5

# the whole log, all jobs, all steps
gh run view 11947722301 --log

# only the failing steps, which is what you want first
gh run view 11947722301 --log-failed

# one job's log
gh run view --job 33417290011 --log
```

2. A real failure the reader would meet, coming from macOS locally and Ubuntu in the pipeline:

```
Run npm test
> jest
```

```
FAIL src/Cart/index.test.ts
  Cannot find module './cartTotal' from 'src/Cart/index.ts'
```

3. The file on disk is `src/Cart/CartTotal.ts` and the import says `./cartTotal`.
4. On macOS this works, because the default file system is case-insensitive. On the Linux runner it does not, because ext4 is case-sensitive.
5. Nothing is wrong with the test, the runner, or the cache. The two machines disagree about whether two file names are the same name.
6. Confirming it without guessing:

```
git ls-files src/Cart | head
# src/Cart/CartTotal.ts      <- what git actually stores
grep -rn "cartTotal" src/Cart/index.ts
```

7. Git stores the name it was given. Your local file system hid the mismatch. The runner did not.

■ 41.14.4 PLAIN – what is really happening inside

1. Debug output is not on by default because it is enormous and it can leak information.
2. Two switches turn it on, both set as repository or environment variables, or as secrets, with the value `true`.
3. `ACTIONS_STEP_DEBUG` makes each step print its internal decisions.
4. `ACTIONS_RUNNER_DEBUG` makes the runner agent itself print how it set up the job, which is what you need when the failure happens before your code runs.
5. With `gh, gh run rerun 11947722301 --debug` sets these for one re-run, which is safer than leaving them on.
6. Reproducing locally has three levels, in increasing fidelity.
7. Level one: run the same command with the same environment variable, since many tools change behaviour when `CI=true` is set.
8. Level two: run the same command inside the same container image the pipeline uses, so the operating system and installed versions match.
9. Level three: run the whole workflow locally with a tool such as `act`, which interprets the workflow file and runs the jobs in containers. It is an approximation, not the real service, and it will differ on some actions.
10. One more trick for the truly stuck: a step that opens an interactive session on the runner, so you can look around while the job is paused. Use it on private repositories only, and never on a job holding production secrets.

■ 41.14.5 TECHNICAL – the engineer’s version

1. Log structure. A run contains jobs; a job contains steps; each step is a log group. The interface collapses groups, so the failing step may be closed and invisible until you expand it. `--log-failed` filters to failing steps and is the correct first command.
2. Read the step from the top. A stack trace prints the deepest frame first in some languages and last in others, and the actual message is at the boundary between your code and the library.
3. Exit code 1 is a generic failure. Some codes are informative: 127 is command not found, 126 is found but not executable, 137 is SIGKILL which on a runner almost always means the out-of-memory killer, and 143 is SIGTERM which usually means a cancellation or a timeout.
4. Ordered checklist for a job that fails only in CI. Work down it; do not skip.

#	Check	Command or setting
1	Same commit?	gh run view --json headSha
2	Reproduces locally?	run the same command
3	OS differs?	uname -a, runner label
4	Tool versions differ?	node -v, python -V
5	Case sensitivity?	git ls-files vs disk
6	File missing?	check .gitignore
7	Env var missing?	env sort in a step
8	Secret missing?	fork PRs get none
9	Locale or timezone?	CI runs UTC, C locale
10	Network blocked?	egress rules, proxies
11	Test order or parallelism?	run with a fixed seed
12	Clean checkout?	no untracked local files
13	Stale cache?	change the cache key
14	Resource limits?	memory, disk, df -h

5. Rows 5, 6, 9 and 12 are the four that catch most people, and all four have the same root cause: your working copy contains state that the repository does not, or your machine normalizes something the runner does not.
6. Case sensitivity in detail. macOS APFS is case-insensitive but case-preserving by default; Linux ext4 and xfs are case-sensitive; Windows NTFS is case-insensitive. Git stores the exact name. A rename that only changes case may not even be recorded unless you use `git mv` explicitly or set `core.ignorecase false`.
7. Time and locale. Hosted runners run in UTC with a minimal locale. A test that formats a date, sorts strings, or parses a decimal comma will behave differently from a laptop set to Asia/Kolkata. Pin TZ and LC_ALL explicitly in the workflow rather than hoping.
8. Diagnostic step worth adding permanently to a slow-to-debug pipeline:

```
- name: Environment fingerprint
  if: always()
  run: |
    uname -a
    node -v || true
    python3 -V || true
    echo "TZ=$TZ LANG=$LANG"
    df -h /
    free -m || vm_stat || true
    git rev-parse HEAD
```

9. Always upload evidence on failure. A failing run that kept no test report, no screenshot and no server log costs a second run to learn anything:

```
- uses: actions/upload-artifact@v4
  if: failure()
  with:
    name: failure-evidence-${{ github.run_id }}
    path: |
      reports/
      logs/
      screenshots/
```

10. `if: failure()` and `if: always()` are the two conditions that matter here. Without one of them the upload step is skipped, because a failed previous step stops the job by default.

11. Retention and download: artifacts are downloadable with `gh run download 11947722301`, and logs with `gh run view --log > run.txt` for grepping.
12. Debug variables: `ACTIONS_STEP_DEBUG` and `ACTIONS_RUNNER_DEBUG`, both set to `true`, as repository variables or secrets. `gh run rerun ID --debug` enables them for one attempt. Debug logs are larger and are retained the same way as normal logs.
13. Local reproduction with `act` from the `nektos/act` project runs workflows in Docker containers. It approximates hosted runners; differences in preinstalled tooling, services and some actions are expected. Treat a green `act` run as encouraging, not as verification.
14. GitLab’s equivalents: the job log with `CI_DEBUG_TRACE: "true"` for shell tracing, artifacts with `when: always`, and `gitlab-runner exec` for local execution. Jenkins exposes the console output plus the pipeline steps view.
15. One caution on `CI_DEBUG_TRACE` and on `set -x`: shell tracing prints every expanded command, and every expanded command may contain a secret. Never enable shell tracing on a job that holds credentials.

■ 41.14.6 WORDS – remember these

1. Step log — the output of one command in a job — a collapsible log group, filterable with `gh run view --log-failed`.
2. `ACTIONS_STEP_DEBUG` — extra detail about each step — a variable set to `true`, enabling verbose action and expression logging.
3. `ACTIONS_RUNNER_DEBUG` — extra detail about job setup — runner-agent level logging, for failures before your code runs.
4. Exit code 137 — the process was killed — `SIGKILL`, on a runner almost always the out-of-memory killer.
5. Case sensitivity — whether two names are the same name — differs between APFS, NTFS and ext4; Git stores the exact stored name.
6. `if: always()` — run this step even after a failure — the condition that makes failure evidence get collected.
7. `act` — run workflows on your own machine — a local interpreter of workflow files using Docker, approximate rather than exact.

41.15 The wider landscape

■ 41.15.1 PLAIN – in simple words

1. Every idea in this chapter exists in every tool. Only the words change.
2. **GitHub Actions** lives inside GitHub. You write YAML in your repository and the machines are supplied for you.
3. **GitLab CI** lives inside GitLab and works the same way, except GitLab is also the issue tracker, the registry and the deployment tool, all one product.
4. **Jenkins** is a program you install on your own server. It is older than the others, it does anything, and you maintain all of it yourself.
5. **CircleCI** is a service that only does pipelines. It is not your code host, so you connect it to one.
6. **Buildkite** splits the job in an unusual way: they run the web interface and the queue, you run the machines.
7. **Argo CD** does something different from all of the above. It does not build. It watches a repository describing what should be running, and makes the real system match it.
8. That last idea has a name: **GitOps**.

9. In GitOps, the repository is the statement of what should be true, and a program inside your system continuously fixes any difference.
10. Nobody deploys by pushing. Everybody deploys by changing a file and merging it.
11. The difference matters most for keys. If nothing pushes into your system, your pipeline never needs credentials to it.

■ 41.15.2 PLAIN – a picture in your head

1. Two ways to keep a garden the way you want it.
2. Push: you walk in with tools whenever you decide, and change things. You need a key to the gate, and so does anyone helping you.
3. Pull: you pin a drawing of the intended garden on the gate. A gardener who already lives inside reads the drawing every few minutes and fixes anything that differs.
4. Nobody outside needs a key. The drawing is public and versioned. Every change is an edit to the drawing.
5. If a storm knocks a plant over, the gardener notices and rights it, without anyone asking.

Where this comparison breaks: a gardener can only fix what the drawing describes. GitOps only reconciles resources under its control, so anything created by hand outside that scope drifts unnoticed. And some changes, like data migrations, are not describable as a desired end state at all.

■ 41.15.3 PLAIN – a worked example

1. The same idea, “run the tests”, in four configuration languages.

```
# GitHub Actions: .github/workflows/ci.yml
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: npm ci && npm test
```

```
# GitLab CI: .gitlab-ci.yml
test:
  image: node:22
  script:
    - npm ci
    - npm test
```

```
// Jenkins: Jenkinsfile
pipeline {
  agent { docker { image 'node:22' } }
  stages {
    stage('test') {
      steps { sh 'npm ci && npm test' }
    }
  }
}
```

```
# CircleCI: .circleci/config.yml
version: 2.1
jobs:
  test:
    docker: [{ image: cimg/node:22.0 }]
    steps:
      - checkout
      - run: npm ci && npm test
```


2. Four files, one idea: check out the code, get a Node environment, run two commands.
3. The differences that actually matter are not syntax. They are who owns the machines, who owns the upgrades, and what happens when you need something unusual.

■ 41.15.4 PLAIN – what is really happening inside

1. Push deployment and pull deployment are the real architectural fork, and every tool sits on one side of it.
2. In push deployment, the pipeline holds credentials to production and reaches in to change it.
3. That is simple, immediate, and means a compromised pipeline is a compromised production system.
4. In pull deployment, an agent runs inside production, reads a repository, and applies the difference.
5. Nothing outside holds production credentials. The pipeline’s only job is to update a file in a repository.
6. The agent also loops forever, so it repairs drift: someone changing something by hand is silently undone within minutes.
7. That loop is the part people underestimate. It converts deployment from an event into a continuously enforced statement.
8. The cost is indirection. You merge a change and then wait for a program you are not watching to notice it, which is harder to debug than a command that either worked or did not.

■ 41.15.5 TECHNICAL – the engineer’s version

1. Comparison on the three axes that decide the choice.

Tool	Hosting	Config	Strength
GitHub Actions	hosted or self	YAML	ecosystem, no setup
GitLab CI	hosted or self	YAML	one integrated product
Jenkins	self-hosted	Groovy	plugins, total control
CircleCI	mainly hosted	YAML	caching, orbs, speed
Buildkite	hybrid	YAML	your hardware, their UI
Argo CD	in your cluster	Git manifests	reconciliation, drift

2. Jenkins. Started as Hudson by Kohsuke Kawaguchi at Sun Microsystems in 2004, first released 2005, renamed Jenkins after a community vote concluded on 29 January 2011. Jenkins 2.0, released in April 2016, introduced Pipeline as code with the Jenkinsfile in a Groovy-based domain-specific language.
3. Jenkins’s strength and weakness are the same fact: well over a thousand plugins. Anything is possible, and the plugin set is your responsibility to patch, test and upgrade forever.
4. GitLab CI. Began in 2012 as a separate application and was merged into GitLab in version 8.0 in September 2015. Uses .gitlab-ci.yml with stages, and a separate gitlab-runner binary with pluggable executors.
5. CircleCI. Launched 2011, config in .circleci/config.yml, with reusable packages called orbs introduced in November 2018. CircleCI disclosed a security incident in January 2023 and instructed customers to rotate all secrets stored in the platform, which is a useful reminder that a hosted CI provider is part of your trust boundary.
6. Buildkite. From Melbourne, Australia, starting in 2013. The distinguishing model is hybrid: Buildkite hosts the control plane and interface, and you run the agents on your own machines, so source code and secrets stay on your infrastructure.

7. Argo CD. Created at Applatix, which Intuit acquired in January 2018. Argo became a CNCF project and graduated, together with Flux, in December 2022.
8. GitOps as a term was coined by Alexis Richardson of Weaveworks in 2017. The OpenGitOps project states four principles: the desired state is declarative; it is versioned and immutable; agents pull it automatically; and agents continuously reconcile actual state towards it.
9. Flux is the other main implementation, also CNCF-graduated. Weaveworks, the company that created Flux and coined the term, ceased trading in early 2024, and the project continued under the foundation. That separation of project from vendor is exactly why foundation governance matters.
10. Others worth recognizing by name: Travis CI, which popularized in-repository configuration from 2011; TeamCity from JetBrains, 2006; Bamboo from Atlassian; Azure Pipelines; AWS CodeBuild and CodePipeline; Google Cloud Build; Tekton, a Kubernetes-native pipeline engine under the CD Foundation; and Concourse, which models everything as resources and tasks.
11. The push versus pull distinction, in security terms:

Model	Credentials live	Blast radius
Push from CI	in the CI system	CI owns production
Pull by agent	inside the cluster	CI owns a Git repo

12. Combining them is the common modern arrangement: continuous integration builds and tests and publishes an image, then updates an image tag in a configuration repository; a cluster agent notices and rolls it out. The pipeline never holds a cluster credential.
13. Portability, honestly stated: pipeline configuration is not portable between these tools, and every migration is a rewrite. What is portable is the shell script your steps call. Teams that keep the real work in scripts and use the YAML only as an orchestrator migrate in days rather than months.
14. Where experts disagree: self-hosted Jenkins for control, or a hosted service for speed. Control genuinely matters in regulated environments and on unusual hardware. Against it, maintaining Jenkins is a permanent part-time job nobody plans for. The deciding factor is whether you have staff to own it.
15. Observing tools across all of them: `gh run`, `glab ci`, the Jenkins `api/json` endpoint per build, `circleci config validate`, `buildkite-agent logs`, and `argocd app get` plus `argocd app diff` for reconciliation state.

■ 41.15.6 WORDS – remember these

1. GitOps — the repository states what should be running — declarative desired state, pulled and continuously reconciled by an in-cluster agent.
2. Reconciliation loop — keep fixing the difference forever — the controller pattern comparing desired state to actual state on a timer.
3. Drift — reality quietly diverging from the file — manual changes to a system under declarative control, reverted on the next reconcile.
4. Push deployment — the pipeline reaches into production — outbound deploys requiring production credentials in the CI system.
5. Pull deployment — production reaches out for its own config — an agent inside the boundary, so no inbound credentials exist.
6. Orb — CircleCI’s reusable configuration package — a versioned bundle of jobs, commands and executors, introduced in 2018.

7. `Jenkinsfile` — Jenkins pipeline as code — a Groovy-based declarative or scripted pipeline definition stored in the repository.

41.98 Common wrong ideas

1. Wrong: continuous integration is a tool you install. Right: it is a habit of merging to the mainline at least daily; the tool only enforces the habit, and a team merging quarterly with Jenkins running is not doing it.
2. Wrong: the green tick means this code is good. Right: it means the checks somebody wrote passed on some commit. It says nothing about the tests that were never written, and nothing about the code you are looking at now.
3. Wrong: a passing branch means the head commit passed. Right: a run is bound to one `head_sha`. A branch is a moving label, so a branch-level tick may belong to a commit that is no longer the head.
4. Wrong: CD means continuous deployment. Right: it usually means continuous delivery, which is “always releasable, released by a human decision”. Continuous deployment removes the human. The two are different practices with the same initials.
5. Wrong: `pull_request_target` is just `pull_request` with more permissions. Right: it runs your trusted workflow file with your secrets, so checking out the pull request head and executing anything from it hands a stranger your credentials.
6. Wrong: `mergeStateStatus: BLOCKED` means there is a merge conflict. Right: it means the branches merge cleanly and a protection rule is unmet. The conflict value is `DIRTY`.
7. Wrong: `UNKNOWN` means something is wrong. Right: it means GitHub has not finished computing mergeability. Re-query after a short delay and conclude nothing from the first answer.
8. Wrong: a skipped check is the same as a passed check. Right: a job skipped by an `if: condition` reports success, while a workflow skipped by a path filter never reports at all and blocks the merge indefinitely.
9. Wrong: pinning an action to `@v4` is safe because it is a version. Right: a tag is a movable pointer in someone else’s repository. Only a full commit SHA is immutable, as the March 2025 `tj-actions/changed-files` compromise showed.
10. Wrong: re-running until it goes green is a fix. Right: it converts a flaky test into a permanent lie, and once people stop believing red, the entire pipeline stops carrying information.

41.99 Chapter summary in 20 lines

1. Continuous integration is a practice, not a product: merge to the mainline at least daily, verify every merge automatically, and fix a broken mainline before starting new work.
2. The term came from Grady Booch in 1991, became an Extreme Programming practice in 1999, and got its working definition from Fowler and Foemmel in 2000.
3. Continuous delivery means every change is always releasable; continuous deployment means every change that passes goes live without a human. Same initials, different practices.
4. A runner is a machine that claims a job, checks out one exact commit, runs your steps, and reports an exit code. Hosted runners are destroyed after each job, which is why “it works on my laptop” fails there.
5. The workflow file lives in the repository, so the pipeline is versioned, reviewable and bisectable like any other code, and a token needs the `workflow` scope to change it.

6. Triggers decide what starts a run: `push`, `pull_request`, `schedule`, `workflow_dispatch`, `repository_dispatch`, tags and releases, narrowed by branch and path filters.
7. `pull_request` runs a fork's code with no secrets and a read-only token. `pull_request_target` runs your file with full secrets, so executing the pull request's code under it is the pwn request vulnerability.
8. The central idea of the chapter: a run is bound to one commit SHA, never to a branch. A branch is a moving pointer, so “CI passed on main” is a statement about whichever commit main pointed at then.
9. The correct verification is four steps: resolve the head SHA from the server, list runs filtered by that SHA, read the specific run by ID, and confirm the run's `head_sha` matches.
10. A branch-level tick can lie in at least six ways: older commit, merge result rather than head, skipped rather than passed, posted by another app, overwritten by a re-run, or attached to a merge commit that no longer exists.
11. Two APIs report results on a commit: the older commit status API with four states, and the newer checks API with check suites, check runs, timings and line annotations.
12. Conclusions are success, failure, neutral, cancelled, skipped, timed_out, action_required, stale and startup_failure, and a required check that never arrives stays Expected and blocks forever.
13. Merge queues solve the semantic conflict: two changes that each pass alone and break together. The queue builds the prospective merged state, tests it, and only then merges, serializing the landing order.
14. Speculative execution tests several queued entries at once assuming those ahead succeed, which is fast when all pass and wasteful when one fails. GitHub's merge queue became generally available on 12 July 2023.
15. `mergeStateStatus` has eight values. CLEAN and HAS_HOOKS mean go, BEHIND means update, DIRTY means real conflict, DRAFT means you marked it, UNSTABLE means a non-required check failed.
16. BLOCKED means a protection rule is unmet, not a conflict; UNKNOWN means GitHub is still computing mergeability, so re-query and conclude nothing.
17. A good pipeline runs cheapest first: lint, build, unit tests, integration tests, security scanning, artifact build, staging deploy, smoke tests, production, with the artifact built once and reused.
18. Deployment strategies trade downtime against cost: recreate, rolling, blue-green, canary, and feature flags, which separate deploying code from releasing it. Database migrations make rollback hard, so expand and contract.
19. Secrets are injected per job and masked by string matching only; forks get none; OIDC federation replaces long-lived cloud keys with one-hour credentials; and third-party actions must be pinned to a full commit SHA.
20. Read a failure by finding the failing step, reading it from the top, reproducing locally, and working an ordered checklist of environment differences before changing anything at random.

SSH, HTTPS, Keys, Tokens and Scopes

42.0 What this chapter gives you

1. You will be able to explain the two ways git talks to a server, SSH and HTTPS, and say exactly what is different and what is identical.
2. You will be able to say what SSH is, who wrote it, in what year, and what dangerous older tools it was built to replace.
3. You will be able to explain a key pair in plain words and then precisely, including what the private key can do that the public key cannot.
4. You will be able to read a real `ssh-keygen` run, decode the public key file field by field, and choose between RSA, ECDSA and Ed25519 with reasons.
5. You will be able to explain `authorized_keys` and `known_hosts`, including why wrong file permissions break logins silently.
6. You will be able to explain why the reader's SSH on port 22 hung while HTTPS on port 443 worked, and apply both fixes.
7. You will be able to explain personal access tokens, their expiry, their prefixes, and why a leaked token is an emergency rather than an annoyance.
8. You will be able to explain a scope precisely, and explain the exact reason a push touching a file under `.github/workflows/` was refused.
9. You will be able to tell a permission failure from a network failure from a rejected push, even though all three print "push failed".
10. You will be able to diagnose at least eight real git authentication error strings from the text alone, and act on each one correctly.

42.1 The two transports git uses, and why there are two

■ 42.1.1 PLAIN – in simple words

1. Git keeps your project history on your own machine. All of it.
2. Sometimes you want to send that history to a server, or fetch new history from one. That is what `push` and `fetch` do.
3. To send anything over a network, two programs need a pipe between them.
4. Git does not build its own pipe. It borrows one that already exists.
5. There are two pipes it commonly borrows, and you choose which one by the shape of the address you type.
6. An address like `git@github.com:owner/repo.git` uses the **SSH** pipe.
7. An address like `https://github.com/owner/repo.git` uses the **HTTPS** pipe, the same kind of pipe your web browser uses.

8. Here is the part people get wrong. The repository on the server is the same repository in both cases. One box, one set of files.
9. Only the road to it changes. It is like posting a letter versus handing it over in person. Same letter, same house, different delivery.
10. So switching from one to the other never loses a commit, never rewrites history, and never needs a re-clone. It is a one-line change.
11. What does change is how the server works out who you are.
12. Over SSH you prove yourself with a key file on your laptop.
13. Over HTTPS you prove yourself with a token, which is a long secret string that behaves like a password with rules attached.

■ 42.1.2 PLAIN – a picture in your head

1. Imagine one library building with one collection of books.
2. There are two doors. The staff door at the back and the public door at the front. Both lead to exactly the same shelves.
3. The staff door has a lock. You carry a physical key that fits it. If your key fits, you walk in. Nobody asks your name.
4. The public door has a desk. You show a card with your name on it and a list of what you are allowed to do. The clerk reads the card and decides.
5. The books do not care which door you used. A book you borrow through the back door is the same book you would have borrowed through the front.
6. Now suppose the alley behind the building is closed for roadworks. The staff door is fine. You simply cannot reach it.
7. You walk round to the front and use the public door. Nothing about the library changed. Your route changed.
8. That is exactly the reader's situation in this chapter, and exactly the fix.

Where this comparison breaks:

9. A real key and a real card are both things you hand over. A private SSH key is never handed over. It stays in your pocket and answers a question.
10. And a library clerk can see your whole card. A well-built server only sees that your token is valid and what it is allowed to do, not your password.

■ 42.1.3 PLAIN – a worked example

1. Here is the same repository addressed both ways, on a real machine.

```
$ git remote add t git@github.com:octocat/Hello-World.git
$ git remote add t https://github.com/octocat/Hello-World.git
$ git remote add t \
    ssh://git@ssh.github.com:443/octocat/Hello-World.git
```

2. All three name one repository: the account octocat, the repository Hello-World.
3. The first is the short SSH form. The colon after the host name is not a port. It separates the host from the path.
4. The second is HTTPS, and it is an ordinary web address with a .git suffix.
5. The third is the long SSH form, written as a proper address with a scheme, which is the only form where you can write a port number.
6. Changing transport on an existing clone is one command:

```
$ git remote set-url origin https://github.com/owner/repo.git
$ git remote -v
origin https://github.com/owner/repo.git (fetch)
origin https://github.com/owner/repo.git (push)
```

7. Nothing else changes. Your branches, your commits and your stash are all still local files under `.git`, untouched.
8. Here is a real clone of a real public repository over HTTPS, from the machine used to write this chapter:

```
$ git clone https://github.com/octocat/Hello-World.git
Cloning into 'Hello-World'...
$ git log --oneline -1
7fd1a60 Merge pull request #6 from Spaceghost/patch-1
```

9. Over SSH the commit `7fd1a60` would be byte-identical, because a commit identity is a hash of its content, not of how it travelled.

■ 42.1.4 PLAIN – what is really happening inside

1. Git itself does not know how to speak to a network. It shells out.
2. When you give it an HTTPS address it runs a small helper program called `git-remote-https`. When you give it an SSH address it runs `ssh`.
3. You can watch it choose. Here is a real trace on the machine used for this chapter, with the timestamps trimmed for width:

```
$ GIT_TRACE=1 git ls-remote \
    https://github.com/octocat/Hello-World.git
trace: built-in: git ls-remote https://github.com/...
trace: run_command: git remote-https https://github.com/...
trace: exec: git-remote-https https://github.com/...
```

4. And here is what git runs for an SSH address. This is the real command line git built, wrapped for width:

```
$ GIT_TRACE=1 git ls-remote \
    ssh://git@ssh.github.com:443/octocat/Hello-World.git
trace: run_command: GIT_PROTOCOL=version=2 ssh
-o SendEnv=GIT_PROTOCOL -p 443 git@ssh.github.com
'git-upload-pack '\'/octocat/Hello-World.git'\''
```

5. Read that last line carefully, because it explains everything about SSH and git in one go.
6. Git asked SSH to log in as the user `git` on the host `ssh.github.com`, and then to run one command there: `git-upload-pack`.
7. SSH is not doing anything git-specific. It is doing what it always does: logging in and running a command on a remote machine.
8. The remote command's input and output are piped straight back down the encrypted connection to your local git.
9. Over HTTPS there is no login shell and no remote command. Instead git makes ordinary web requests. Here are the real ones, captured live:

```
GET /octocat/Hello-World.git/info/refs?service=git-upload-pack
Git-Protocol: version=2
<= HTTP/1.1 200 OK
<= Content-Type: application/x-git-upload-pack-advertisement
```



```
POST /octocat/Hello-World.git/git-upload-pack
Content-Type: application/x-git-upload-pack-request
<= HTTP/1.1 200 OK
<= Content-Type: application/x-git-upload-pack-result
```

10. One GET to ask what branches exist, then one POST carrying the real conversation. That is the whole thing.
11. Underneath both, the same objects flow: the same commits, trees, blobs and the same packfile format. The payload is identical.

■ 42.1.5 TECHNICAL – the engineer’s version

1. Git supports four transports. Two matter today.

Transport	Port	Authentication
SSH	TCP 22	public key, user git
HTTPS smart	TCP 443	token in Basic auth
git:// (dumb)	TCP 9418	none, removed by GitHub
local file path	none	filesystem permissions

2. GitHub removed support for the unencrypted git:// protocol as part of a protocol hardening programme that became permanent on 15 March 2022.
3. Over SSH, git uses the SSH connection protocol to request a single command execution, not an interactive shell. The remote side runs git-upload-pack for fetches and git-receive-pack for pushes.
4. The environment variable GIT_PROTOCOL=version=2 is passed with -o SendEnv=GIT_PROTOCOL. Protocol v2 was introduced in Git 2.18 in June 2018 and made the default for fetches in Git 2.26 in March 2020.
5. Protocol v2 matters on large repositories because it lets the client ask for a filtered reference advertisement instead of receiving every ref.
6. Over HTTPS, git speaks the **smart HTTP** protocol, defined in the Git documentation file http-protocol.txt. Two endpoints per operation: GET \$URL/info/refs?service=git-upload-pack then POST \$URL/git-upload-pack.
7. The media types are application/x-git-upload-pack-request and application/x-git-upload-pack-result, and the push equivalents replace upload with receive.
8. Rewriting one transport into the other is a supported config feature, and it is the cleanest fix for a blocked port:

```
$ git config --global \
  url."https://github.com/".insteadOf "git@github.com:"
$ git config --global \
  url."https://github.com/".insteadOf "ssh://git@github.com/"
```

9. On the machine used for this chapter that rewrite is active, and it is visible in a real check:

```
$ git remote add t git@github.com:octocat/Hello-World.git
$ git remote get-url t
https://github.com/octocat/Hello-World.git
```

10. That is worth knowing as a trap. If a colleague swears their SSH URL works on a machine where port 22 is blocked, check insteadOf before believing that port 22 is open.

11. `pushInsteadOf` is the write-only variant, letting you fetch anonymously over HTTPS while pushing over SSH.

■ 42.1.6 WORDS – remember these

Transport — the road the data takes — the protocol carrying the Git wire protocol, SSH or HTTPS.

Remote — a nickname for a server copy — a named URL plus fetch and push refsspecs stored in `.git/config`.

`git-upload-pack` — the program that sends you history — the server-side process serving fetch and clone.

`git-receive-pack` — the program that accepts your history — the server-side process serving push, and the one that runs the hooks that can refuse you.

Smart HTTP — git talking over web requests — the `info/refs` plus `git-upload-pack` request pair using `x-git-*` media types.

`insteadOf` — a rule that rewrites addresses — a `url.<base>.insteadOf` config key that substitutes URL prefixes before the transport is chosen.

42.2 What SSH actually is

■ 42.2.1 PLAIN – in simple words

1. SSH stands for Secure Shell. A shell is the text prompt where you type commands.
2. SSH lets you get a shell on a computer that is somewhere else, over a network, safely.
3. “Safely” means two separate promises, and both matter.
4. First, nobody in the middle can read what you type or what comes back. It is scrambled on the wire.
5. Second, you can be sure the machine you reached is the machine you meant. Not an impostor sitting in the middle.
6. Before SSH, people used tools called `telnet`, `rlogin` and `rsh` to do the same job. Those tools sent your password as plain readable text.
7. Anyone able to watch the network could simply read it. Not crack it. Read it, like reading a postcard.
8. In 1995 a researcher at a Finnish university discovered exactly that happening on his own network and wrote SSH in response.
9. Today SSH does far more than shells. It carries file copies, tunnels, and the git pushes in this chapter.
10. But underneath it is still the same thing: a safe way to run one command on another machine.

■ 42.2.2 PLAIN – a picture in your head

1. Think of the old way as shouting your order across a crowded market square.
2. The stallholder hears you. So does everyone else, including the person who writes down what people order and comes back later pretending to be you.
3. Now think of SSH as a sealed pneumatic tube running from your desk to the stall.
4. You put a written note in a capsule, it travels inside the tube, and only the stall can open it. Nobody in the square sees anything.
5. But a tube alone is not enough. What if somebody quietly re-routed your tube to a different stall last night?

6. So SSH adds a second step. Before you send anything, the stall shows you a seal that only that stall could have made.
7. You compare it against the seal you wrote down the first time you visited. If it matches, it is really them.
8. If it does not match, everything stops with a very loud warning.

Where this comparison breaks:

9. A pneumatic tube is a physical private path. SSH runs over the same shared public cables as everything else. The privacy is mathematical, not physical.
10. And the “seal you wrote down the first time” is a real weakness. On that first visit you had no way to check. You simply trusted. We will look straight at that in section 42.5.

■ 42.2.3 PLAIN - a worked example

1. A connection has three visible stages. Here is a real one against a small server, with the client asked to explain itself.

```
$ ssh -v -p 2222 root@127.0.0.1 'echo OK'
debug1: Authenticating to 127.0.0.1:2222 as 'root'
debug1: kex: algorithm: sntrup761x25519-sha512@openssh.com
debug1: kex: host key algorithm: ssh-ed25519
debug1: kex: server->client cipher:
      chacha20-poly1305@openssh.com MAC: <implicit>
debug1: Server host key: ssh-ed25519
      SHA256:fwKYxD4JEmBcHPHj3WkgxSLrRh6tXIzpi/K0xz2yNrA
debug1: Host '[127.0.0.1]:2222' is known and matches the
      ED25519 host key.
debug1: Authentications that can continue: publickey
debug1: Offering public key: id_ed25519 ED25519
      SHA256:WH1E2gSLhG6FgMsha//GINoquuvX3fLxFRdMA3KMrSU
debug1: Server accepts key: id_ed25519 ED25519
Authenticated to 127.0.0.1 using "publickey".
OK
```

2. Line by line, in order of what happened.
3. kex is short for key exchange. Both sides agreed on the maths they would use to create a shared secret. That is stage one.
4. Server host key is the server proving who it is. Host is known and matches means the client checked its own notes and was satisfied.
5. Authentications that can continue: publickey is the server saying which ways of proving yourself it will accept. Here, only keys.
6. Offering public key then Server accepts key is stage two, you proving who you are.
7. OK at the end is stage three, the actual work happening inside the now trusted, now encrypted connection.
8. Notice the order. Encryption is set up first, then the server proves itself, then you prove yourself. Your credentials never travel in the open.

■ 42.2.4 PLAIN - what is really happening inside

1. SSH version 2 is built as three stacked layers. Each has one job.
2. **Layer one, transport.** It runs immediately after the TCP connection opens. It has three tasks.
3. It agrees the algorithms both sides will use, it performs a key exchange to create a shared secret, and it verifies the server's host key.

4. From that point on, everything is encrypted and every message carries an integrity check so tampering is detected.
5. Importantly, the shared secret is created fresh for this connection. It is never sent over the wire, by either side. Both sides compute it.
6. **Layer two, user authentication.** Now that the pipe is safe, you prove who you are. Public key, password, keyboard-interactive, or host-based.
7. This layer runs inside the encrypted tunnel, which is why a password sent over SSH is not exposed the way a telnet password was.
8. **Layer three, connection.** One SSH connection can carry many independent streams, called channels, at once.
9. A shell is a channel. A file copy is a channel. A forwarded port is a channel. They share the one encrypted pipe.
10. That is how you can be editing a file over SSH while a port forward runs in the background, on one connection and one login.
11. For git, exactly one channel is opened, of type “exec”, carrying one command: `git-upload-pack` or `git-receive-pack`.



■ 42.2.5 TECHNICAL - the engineer's version

1. SSH was written by Tatu Ylonen at Helsinki University of Technology in 1995, after a password-sniffing attack on the university network.
2. He released the first version in July 1995, and founded SSH Communications Security in December 1995 to commercialize it.
3. The protocol it replaced was the Berkeley r-commands: `rlogin` (TCP 513), `rsh` (TCP 514) and also `telnet` (TCP 23), all of which sent credentials in clear text.
4. Those tools also trusted a file called `.rhosts`, which authenticated by source IP address alone. Address spoofing therefore meant login.
5. **SSH-1** and **SSH-2** are different protocols, not versions of one protocol. SSH-1 has a structural integrity flaw.
6. SSH-1 used CRC-32 for integrity, which is a checksum, not a cryptographic authentication code. In 1998 this was shown to allow an insertion attack, and in 2001 the widely reported “SSH CRC-32 compensation attack detector” vulnerability made it worse. SSH-1 is dead and should never be enabled.
7. SSH-2 is specified in a set of IETF documents published in January 2006: RFC 4250 assigned numbers, RFC 4251 architecture, RFC 4252 authentication, RFC 4253 transport, RFC 4254 connection.
8. **OpenSSH** is a separate implementation, from the OpenBSD project. The history is precise and worth knowing.

Date	Event
July 1995	Ylonen releases ssh 1.x
early 1999	Gronvall's OSSH from 1.2.12
26 Sep 1999	OpenBSD forks OSSH
1 Dec 1999	OpenSSH 1.2.2 in OpenBSD 2.6
15 Jun 2000	SSH-2 support, OpenSSH 2.0

9. The fork happened because releases after ssh 1.2.12 carried increasingly restrictive licence terms. Bjorn Gronvall's OSSH revived the last freely licensed release, and the OpenBSD team took it from there.
10. SSH-2 support was written by Markus Friedl and shipped in OpenSSH 2.0 with OpenBSD 2.7 on 15 June 2000.
11. Port 22 is the IANA-assigned port for SSH. Ylonen requested it because it sat unused between ftp on 21 and telnet on 23.
12. Modern algorithm defaults on the OpenSSH 9.6 used for this chapter: key exchange `sntrup761x25519-sha512@openssh.com`, which combines the post-quantum lattice scheme NTRU Prime with X25519, and cipher `chacha20-poly1305@openssh.com`.
13. That hybrid key exchange became the OpenSSH default in version 9.0, in April 2022. The reason given was harvest-now-decrypt-later: an attacker recording traffic today and decrypting it after quantum computers arrive.

■ 42.2.6 WORDS – remember these

SSH — a safe way to use a distant computer — the Secure Shell protocol suite specified in RFC 4251 through RFC 4254.

Key exchange (kex) — the step where both sides build a shared secret — a Diffie-Hellman style agreement producing session keys without transmitting them.

Host key — the server's own identity key — the long-term key pair a server uses to sign the key exchange and prove it is itself.

Channel — one stream inside the connection — a multiplexed logical stream in the SSH connection protocol, with independent flow control.

telnet — an old unsafe remote login tool — a clear-text remote terminal protocol on TCP 23, RFC 854, with no encryption or integrity.

OpenSSH — the SSH program almost everyone runs — the OpenBSD implementation, first released 1 December 1999 with OpenBSD 2.6.

42.3 Public key cryptography for SSH

■ 42.3.1 PLAIN – in simple words

1. A key pair is two files made at the same time, mathematically joined.
2. One is called the **private key**. It stays on your machine and you never send it anywhere, ever.
3. The other is called the **public key**. It is meant to be handed out. You paste it into web forms and email it to people.
4. The private key can do one special thing: it can make a **signature** on a piece of data.
5. The public key cannot make a signature. It can only check one, and answer yes or no.

6. That asymmetry is the whole trick. Publishing the public key gives an attacker nothing, because checking signatures is not making them.
7. So when GitHub wants to know you are you, it does not ask for a password.
8. It sends your computer a piece of data and says: sign this.
9. Your computer signs it with the private key. GitHub checks the signature with the public key it already stored under your account.
10. If the check passes, only the holder of that private key could have produced it. You are in.
11. Nothing secret ever crossed the network. The private key stayed in a file on your laptop the entire time.
12. That is why a public key on a public web page is not a leak, and why a private key in a chat message is a catastrophe.

■ 42.3.2 PLAIN – a picture in your head

1. Imagine a very unusual rubber stamp.
2. Only you own the stamp. It presses a pattern so intricate that nobody can carve a copy from looking at an impression.
3. You give everyone a printed picture of the pattern. That is the public key.
4. Now anyone can send you a fresh sheet of paper with today's date and a random number on it and ask you to stamp it.
5. You stamp it and hand it back. They compare the impression with the printed picture. It matches, so it was you.
6. The random number matters more than it looks. If the paper always said the same thing, someone could keep an old stamped sheet and reuse it.
7. Because the paper is fresh every time, an old stamped sheet proves nothing about today.

Where this comparison breaks:

8. A rubber stamp makes the same mark every time. A real signature depends on both the key and the exact data, so it is different for every message.
9. And a rubber stamp can be physically stolen and used. A private key can be locked with a passphrase, so stealing the file is not always enough.
10. The honest version: some key types, RSA in particular, can also be used the other way round, to encrypt something that only the private key can open. In SSH user authentication, modern practice is signing, not encrypting.

■ 42.3.3 PLAIN – a worked example

1. Here is a real key being made on the machine used for this chapter. Nothing is edited except line width.

```
$ ssh-keygen -t ed25519 -C "kedbyte@laptop"
Generating public/private ed25519 key pair.
Enter file in which to save the key (~/ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in ~/ssh/id_ed25519
Your public key has been saved in ~/ssh/id_ed25519.pub
The key fingerprint is:
SHA256:WH1E2gSLhG6FgMsha//GINoquvX3fLxFRdMA3KMrSU
kedbyte@laptop
The key's randomart image is:
+--[ED25519 256]--+
```

```
| ... 0. .+B0.0 |
|. 0 0..0 E0+0 .|
| + 0 . .0 + * 0 |
|..0 00 0 .|
|. . . S . .|
|. 0 0 |
|... = . . .|
|+ .. = 0 .0 .|
|0+. . 0. .|
+----[SHA256]-----+
```

- Two files came out. `id_ed25519` is private, 411 bytes. `id_ed25519.pub` is public, 96 bytes.
- The fingerprint is a short hash of the public key, for humans to compare by eye. It is not a key and it is not secret.
- The randomart is the same fingerprint drawn as a picture, because humans spot a changed picture faster than a changed string of letters.
- Now let us prove what a private key can actually do. Sign a file:

```
$ echo "hello kedbyte" > msg.txt
$ ssh-keygen -Y sign -f ~/.ssh/id_ed25519 -n file msg.txt
Signing file msg.txt
Write signature to msg.txt.sig
```

- Check it with only the public key:

```
$ ssh-keygen -Y verify -f allowed_signers -I kedbyte@laptop \
-n file -s msg.txt.sig < msg.txt
Good "file" signature for kedbyte@laptop with ED25519 key
SHA256:WH1E2gSLhG6FgMsha//GINoquvX3fLxFRdMA3KMrSU
```

- Now change one word of the message and check the same signature again:

```
$ ssh-keygen -Y verify -f allowed_signers -I kedbyte@laptop \
-n file -s msg.txt.sig < msg2.txt
Signature verification failed: incorrect signature
Could not verify signature.
```

- That is the entire idea, demonstrated. The signature is bound to the exact bytes. Change one character and it stops matching.

■ 42.3.4 PLAIN – what is really happening inside

- Look at the public key file. It is one line with three fields separated by spaces. Here it is, wrapped for the page:

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIIpK1C+6wSQAKCH1
gza1Sb50MnBwsPXPd6FAeelzkUGS kedbyte@laptop
```

- Field one, `ssh-ed25519`, is the key type. Field three, `kedbyte@laptop`, is a comment. It is a label for humans and has no effect on anything.
- Field two is the key itself, encoded in base64 so it can be pasted into a text box. Decoded, it is 51 bytes with a strict internal structure.
- That structure is a repeating pattern: a four-byte length, then that many bytes of data. Here is the real decode of the key above:

```
total bytes: 51
field 1: length 11 -> 'ssh-ed25519'
```



```
field 2: length 32 -> 8a4a942fbac124009021e58336b549be
                        74327070b0f5cf0fa14079e973914192
```

5. So the type name appears twice: once in plain text outside the base64, and once inside it. The inside copy is the one that counts.
6. The 32 bytes are the actual Ed25519 public key. That is the whole secret-free half of the pair, and it really is only 32 bytes.
7. Now the login exchange, in order.
8. Your client tells the server: I would like to authenticate as this user with this public key. Would you accept it at all?
9. The server looks the key up in that user's `authorized_keys` file. If it is not there, the server says no and your client tries the next key.
10. If it is there, the server says yes, that key is acceptable. Note that nothing has been proven yet.
11. Your client now builds a block of data to sign. It contains the session identifier from the key exchange, the user name, the service name, and the public key.
12. The session identifier is the crucial part. It is unique to this one connection and could not have been known in advance.
13. Your client signs that block with the private key and sends the signature.
14. The server verifies the signature against the stored public key. If it verifies, you are authenticated.
15. Because the signed block includes the session identifier, a signature captured from an old session cannot be replayed into a new one.
16. And because the server also binds its own host key into the session identifier, a man-in-the-middle cannot forward your signature to the real server. That property is why host key checking is not optional.

■ 42.3.5 TECHNICAL - the engineer's version

1. Three key types are in real use, plus a dead one. Real measurements from the OpenSSH 9.6 machine used for this chapter.

Type	Bits	Public file size
Ed25519	256	96 bytes
ECDSA nistp256	256	182 bytes
RSA 4096	4096	744 bytes
DSA	1024	removed everywhere

2. Generation cost, averaged over five runs on the same machine:

Type	Average generation time
Ed25519	0.010 s
ECDSA nistp256	0.008 s
RSA 2048	0.229 s
RSA 4096	2.160 s

3. **RSA.** Published by Rivest, Shamir and Adleman in 1978. Security rests on the difficulty of factoring a large number into two primes.
4. 1024-bit RSA is broken in practice for well-funded attackers and is refused by modern software. 2048 bits is the floor, and is what NIST Special Publication 800-57 treats as acceptable to roughly 2030.

5. 4096 bits is common because it costs almost nothing for a key you generate once, and pushes the estimated strength well past the 2030 line.
6. Beyond 4096 the returns collapse. RSA key strength grows very slowly with size while signing cost grows fast, which is why 8192 is rare.
7. The critical modern detail is the signature algorithm, not the key size. The old `ssh-rsa` algorithm uses SHA-1, which has been considered unsafe since the 2017 SHAttered collision by CWI Amsterdam and Google.
8. RFC 8332, published in March 2018, defines `rsa-sha2-256` and `rsa-sha2-512` for the same RSA keys. OpenSSH disabled `ssh-rsa` by default in version 8.8 in September 2021.
9. **ECDSA**. The elliptic curve digital signature algorithm, standardized in FIPS 186. In SSH it is used with the NIST curves P-256, P-384 and P-521.
10. ECDSA works and is widely supported, but it has two known unhappy properties. It needs a fresh random value for every signature, and a repeated or biased value leaks the private key outright.
11. That is not theoretical. The 2010 Sony PlayStation 3 signing key recovery was exactly this bug, a fixed nonce.
12. Also, some cryptographers distrust the provenance of the NIST curve constants. This is a genuine expert disagreement. One side says the constants are unexplained and should be avoided, the other says no attack has ever followed from them and the curves are fine.
13. **Ed25519**. EdDSA over the twisted Edwards curve Curve25519, from the 2011 paper “High-speed high-security signatures” by Bernstein, Duif, Lange, Schwabe and Yang. Curve25519 itself is Bernstein’s, from 2006.
14. It is the current recommendation for SSH, for four reasons that are concrete rather than fashionable.
15. It is deterministic: the per-signature value is derived from the key and the message, so the ECDSA nonce failure mode cannot happen.
16. It is small and fast: 32-byte public keys, 64-byte signatures, and verification that is quick on tiny hardware.
17. Its parameters were chosen by a published rule with stated rationale, so there are no unexplained constants.
18. Its implementations are naturally constant-time, which resists timing side-channel attacks without special care.
19. Ed25519 in SSH is standardized in RFC 8709, February 2020, though OpenSSH shipped it in version 6.5 in January 2014, six years earlier.
20. **DSA**. `ssh-dss`, 1024 bits, roughly 80-bit security. GitHub removed support for DSA keys entirely, permanently enforced on 15 March 2022.
21. The complete list your OpenSSH understands is one command:

```
$ ssh -Q key
ssh-ed25519
sk-ssh-ed25519@openssh.com
ecdsa-sha2-nistp256
ecdsa-sha2-nistp384
ecdsa-sha2-nistp521
sk-ecdsa-sha2-nistp256@openssh.com
ssh-dss
ssh-rsa
```

22. The `sk-` prefixed types are hardware security key types, added in OpenSSH 8.2 in February 2020. The private key half lives on a physical device such as a YubiKey and cannot be copied off it.
23. Fingerprints default to SHA-256 in base64 since OpenSSH 6.8, March 2015. The older MD5 hex

form is still available and still seen in old documentation:

```
$ ssh-keygen -lf id_ed25519.pub
256 SHA256:WH1E2gSLhG6FgMsha//GINoquuvX3fLxFRdMA3KMrSU
kedbyte@laptop (ED25519)

$ ssh-keygen -lf id_ed25519.pub -E md5
256 MD5:74:36:b8:87:74:1d:49:9c:09:81:62:6b:89:eb:26:ad
kedbyte@laptop (ED25519)
```

24. Practical recommendation, and it is not controversial: `ssh-keygen -t ed25519 -C "you@machine"`. Use RSA 4096 only when you must talk to something old that cannot do Ed25519.

■ 42.3.6 WORDS – remember these

Key pair — two matched files, one secret one public — an asymmetric keypair where the public half is derivable from the private half but not the reverse.

Private key — the file you never share — the secret scalar or exponent, stored under `~/.ssh/`, optionally encrypted with a passphrase.

Public key — the file you paste everywhere — the shareable half, safe to publish, stored on the server in `authorized_keys`.

Fingerprint — a short code for comparing keys by eye — the base64 SHA-256 hash of the public key blob, printed by `ssh-keygen -l`.

Signature — proof that the holder of the key saw this data — the output of a signing algorithm over specified bytes, verifiable with the public key alone.

Ed25519 — the recommended modern key type — EdDSA over Curve25519, RFC 8709, 32-byte public keys, deterministic signatures.

Nonce — a number used once — a per-signature random or derived value whose reuse in ECDSA leaks the private key.

42.4 `authorized_keys` and how it works

■ 42.4.1 PLAIN – in simple words

1. On the server there is one plain text file that decides who may log in as a given user with a key.
2. It is called `authorized_keys` and it lives in that user's home directory, inside a folder called `.ssh`.
3. Each line of the file is one public key. If your public key is on a line, you may log in as that user.
4. That really is the entire mechanism. There is no database and no clever service. It is a text file that gets read.
5. To give someone access, you append their public key as a new line. To remove access, you delete that line.
6. There is one rule that catches everybody. The file and its folder must not be writable by anyone other than the owner.
7. If they are, the server refuses to use the file at all, and says nothing useful about why.
8. The reason is sound. If any other user could edit that file, they could add their own key and become you. So SSH refuses to trust it.
9. The failure is silent from the client's side. You get "permission denied" and no explanation.
10. GitHub has a variant of this idea for a single repository, called a **deploy key**. Same file format, narrower reach.

■ 42.4.2 PLAIN – a picture in your head

1. Think of a guest list on a clipboard at the door of a private room.
2. The doorman does not know anyone's face. He only checks the list.
3. To let someone in, you write their name on the list. To stop them, you cross it off. There is no other step.
4. Now add the rule that catches people out. The clipboard hangs on a hook in the public corridor.
5. If anyone walking past can write on it, the list is worthless. Anyone could add themselves.
6. So the doorman has a standing instruction: if the clipboard is not locked in its case, ignore the list entirely and let nobody in.
7. And he does not explain. He just says "not on the list", because telling you about the broken case would help an attacker more than it helps you.

Where this comparison breaks:

8. A name on a list can be claimed by anyone who knows the name. A public key on a line can only be used by whoever holds the matching private key.
9. So the list being publicly readable is fine. Only the list being publicly writable is fatal.

■ 42.4.3 PLAIN – a worked example

1. This is a real server, a real key, and a real failure, produced on the machine used for this chapter.
2. First, correct permissions. The connection works.

```
$ ls -l ~/.ssh/authorized_keys
-rw----- 1 root root 96 Aug 13 08:22 authorized_keys
$ ssh -i ~/.ssh/id_ed25519 -p 2222 root@127.0.0.1 'echo OK'
OK
```

3. Now make the file writable by everyone, which is exactly the mistake people make when they get frustrated and run `chmod 666`.

```
$ chmod 666 ~/.ssh/authorized_keys
$ ls -l ~/.ssh/authorized_keys
-rw-rw-rw- 1 root root 96 Aug 13 08:22 authorized_keys
$ ssh -i ~/.ssh/id_ed25519 -p 2222 root@127.0.0.1 'echo OK'
root@127.0.0.1: Permission denied (publickey).
```

4. The key did not change. The user did not change. Only the permission bits changed, and the login broke.
5. Nothing in the client message says so. It says `Permission denied (publickey)`, which is the same message you get for a completely wrong key.
6. Now the server's own log, from the same moment:

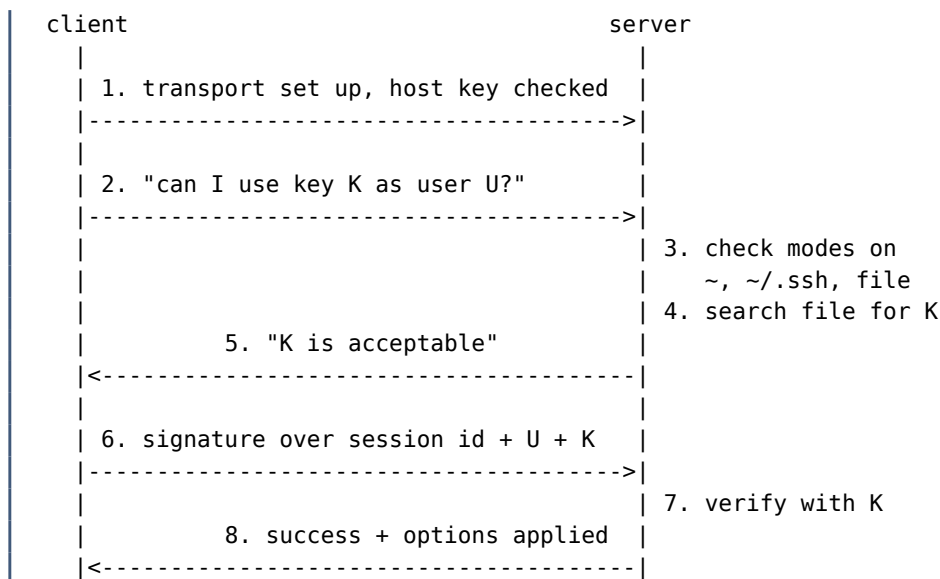
```
Connection from 127.0.0.1 port 48576 on 127.0.0.1 port 2222
Authentication refused: bad ownership or modes for file
/root/.ssh/authorized_keys
Failed publickey for root from 127.0.0.1 port 48576 ssh2:
ED25519 SHA256:WH1E2gSLhG6FgMsha//GINoquovX3fLxFRdMA3KMrSU
Connection closed by authenticating user root [preauth]
```

7. There it is: `bad ownership or modes`. The truth exists, but only on the server side.
8. This is the single most useful piece of knowledge in this section. When public key login fails and you control the server, read the server log.
9. The fix is two commands:

```
$ chmod 700 ~/.ssh
$ chmod 600 ~/.ssh/authorized_keys
```

■ 42.4.4 PLAIN - what is really happening inside

1. Here is the exchange as a numbered sequence, with a diagram after it.
2. The client opens TCP to port 22 and the two sides complete the encrypted transport layer, including the server proving its host key.
3. The client sends an authentication request naming the user, the service, the method `publickey`, and one public key. A flag says "this is only a query".
4. The server takes the user name and looks up that user's home directory.
5. It checks the ownership and permission bits on the home directory, on `~/.ssh`, and on `~/.ssh/authorized_keys`.
6. If any of them is writable by group or by others, the server abandons the file and behaves as if no key matched. This is the silent failure.
7. Otherwise it reads the file line by line, skipping blanks and comments, looking for a line whose key blob equals the one offered.
8. If none matches, the server replies "failure" and the client tries its next key, then finally gives up with `Permission denied (publickey)`.
9. If one matches, the server replies "this key is acceptable, now prove it".
10. The client builds the signature block, signs it, and sends it. The server verifies with the stored public key.
11. On success the server applies any options written in front of that key on that line, then starts the session.



12. Steps 3 and 4 are where nearly all real-world failures live, and neither of them tells the client anything.

■ 42.4.5 TECHNICAL - the engineer's version

1. Default path is `~/.ssh/authorized_keys`, set by the `AuthorizedKeysFile` directive in `sshd_config`. It accepts multiple paths and the tokens `%h` for home directory and `%u` for user name.
2. Required modes, enforced when `StrictModes yes` is set, which is the default:

Path	Required
~	not group/other writable
~/.ssh	700 recommended
~/.ssh/authorized_keys	600, owner only
~/.ssh/id_* private key	600, client side too

- The client enforces its own modes as well. A private key readable by others produces UNPROTECTED PRIVATE KEY FILE and is refused.
- Each line is: optional comma-separated options, then key type, then base64 key, then optional comment. No line continuation exists, so a wrapped paste is the second most common cause of failure after permissions.
- The options that matter in production:

Option	Effect
command="..."	force this command only
from="host,pattern"	restrict source addresses
no-port-forwarding	block tunnels
no-agent-forwarding	block agent reuse
no-pty	no interactive terminal
restrict	all restrictions at once
expiry-time="YYYYMMDD"	key stops working

- restrict was added in OpenSSH 7.2, February 2016, and is the safe default because it enables all current and future restrictions, letting you re-enable individually with permit* options.
- A real forced command, tested on the machine used for this chapter:

```
$ cat ~/.ssh/authorized_keys
command="echo forced: you may only run this",
no-port-forwarding,no-agent-forwarding,no-pty
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAA... kedbyte@laptop

$ ssh -i ~/.ssh/id_ed25519 -p 2222 root@127.0.0.1 \
  'rm -rf /important'
forced: you may only run this
```

- The destructive command was sent and simply not run. The original command is still available to the forced command in the environment variable SSH_ORIGINAL_COMMAND, which is how restricted git servers work.
- That is precisely the mechanism behind git-shell, the login shell that permits only git-upload-pack, git-receive-pack and git-upload-archive.
- Deploy keys** are GitHub's per-repository variant. The public key is attached to one repository, not to a user account.
- Deploy keys are read-only by default and can be granted write access when added. One key cannot be reused across repositories: GitHub rejects a duplicate.
- GitHub lists five alternatives to deploy keys: SSH agent forwarding, HTTPS with OAuth tokens, GitHub App installation access tokens which expire after one hour, machine users, and personal access tokens.
- On GitHub, all SSH access uses the single account name git. Your identity comes entirely from which key signed, not from the user name in the URL.

14. That is why `git@github.com` is the same for every person on earth, and why `ssh -T git@github.com` greets you by your own name.

■ 42.4.6 WORDS – remember these

`authorized_keys` — the list of keys allowed in — a per-user file of public key lines consulted by `sshd` during publickey authentication.

`StrictModes` — the rule that file permissions must be tight — an `sshd` directive making the server refuse key files writable by group or others.

`Forced command` — a key that can only do one thing — the `command="..."` option overriding whatever the client requested.

`SSH_ORIGINAL_COMMAND` — what the client actually asked for — the environment variable holding the requested command when a forced command is in effect.

`Deploy key` — a key for one repository, not one person — a public key attached to a single GitHub repository, read-only unless granted write.

`git-shell` — a login shell that only allows git — a restricted shell shipped with Git permitting only the three pack transfer commands.

42.5 known_hosts and host key verification

■ 42.5.1 PLAIN – in simple words

1. Everyone thinks about proving themselves to the server. Almost nobody thinks about the server proving itself to them.
2. But that half exists, it runs first, and it is the half that stops somebody sitting invisibly in the middle of your connection.
3. Every SSH server has its own key pair, separate from any user's. It is called the **host key**.
4. During setup, the server signs part of the conversation with its private host key. Only the real server could do that.
5. Your client checks that signature against a copy of the server's public key it saved earlier, in a file called `known_hosts`.
6. And here is the awkward truth. The first time you ever connect, your client has no saved copy, so it cannot check anything.
7. So it asks you. It shows you a fingerprint and asks whether to trust it. Most people type yes without looking.
8. That approach has a name: **trust on first use**. You are trusting that nobody was interfering during that one first moment.
9. After that first yes, the key is saved and every later connection is properly checked.
10. If the saved key ever stops matching, SSH stops everything and prints a very loud warning. That warning is doing its job. Read it, do not delete it.

■ 42.5.2 PLAIN – a picture in your head

1. You are meeting someone you have only spoken to by letter, and you must be sure it is really them.
2. The first meeting is the hard one. They show you a birthmark. You have no way to check it, so you write it down and decide to believe it.
3. Every meeting afterwards is easy. You look for the birthmark you wrote down. Match means it is them.

4. Now imagine you turn up and the birthmark is different.
5. Three explanations. They had it removed and told nobody. They have an identical twin who came instead. Or somebody is impersonating them.
6. You cannot tell which from looking. What you can do is stop, and go and check through a different channel before saying anything private.

Where this comparison breaks:

7. A birthmark cannot be changed at will. A host key can be legitimately replaced, and sometimes must be. So a mismatch is not automatically an attack.
8. And a person can be phoned to confirm. A server cannot. You have to find the fingerprint published somewhere trustworthy instead.

■ 42.5.3 PLAIN – a worked example

1. All of the following is real output from the machine used for this chapter, against a small server started for the purpose.
2. The first connection, with nothing saved yet:

```
$ ssh -p 2222 root@127.0.0.1
The authenticity of host '[127.0.0.1]:2222' can't be
established.
ED25519 key fingerprint is
SHA256:nu2laRCiSw3wMTbG6hoH8tKukC952d+/PbR0tgsD5vU.
This key is not known by any other names.
Are you sure you want to continue connecting
(yes/no/[fingerprint])?
```

3. Answer yes and the client writes one line into known_hosts. Here it is, truncated for page width:

```
$ cat ~/.ssh/known_hosts
[127.0.0.1]:2222 ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAA
IJJmXI2wR7keztPI78nlxu2ooGFa2+d/ke5qLMBiy5...
```

4. Three fields: which host, which key type, and the key. Note the host is in square brackets with the port, because it is not port 22.
5. Now the server's host key was deliberately replaced and the same connection tried again. This is the real, complete warning:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@  WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!  @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now
(man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the ED25519 key sent by the remote
host is
SHA256:fwKYxD4JEmBcHpHj3WkgxSLrRh6tXIzpi/K0xz2yNrA.
Please contact your system administrator.
Add correct host key in ~/.ssh/known_hosts to get rid of
this message.
Offending ED25519 key in ~/.ssh/known_hosts:1
    remove with:
    ssh-keygen -f '~/.ssh/known_hosts' -R '[127.0.0.1]:2222'
Host key for [127.0.0.1]:2222 has changed and you have
```

```
requested strict checking.
Host key verification failed.
```

6. Notice the message even tells you the exact line number and the exact command to remove it. That convenience is why people delete first and think never.
7. Notice also that the connection stopped. No password prompt appeared. SSH refused to send anything to a server it could not identify.

■ 42.5.4 PLAIN – what is really happening inside

1. The host key check happens during the key exchange, before any authentication method is even offered.
2. The two sides compute a shared secret, then a hash covering both version strings, both algorithm lists, both key exchange values and the shared secret. That hash is the **session identifier**.
3. The server signs the session identifier with its private host key and sends the signature plus its public host key.
4. The client verifies the signature using the public host key it was just sent. That proves the far end holds the matching private host key.
5. But it does not prove the far end is the right machine. Anyone can generate a key pair and sign with it.
6. So the client separately compares that public host key against `known_hosts`. That is the step that ties the maths to a name you care about.
7. Four outcomes. Not found means prompt the user. Found and matching means proceed. Found and different means abort loudly.
8. The fourth outcome is subtler: found for a different key type. If you have an Ed25519 entry and the server offers RSA, OpenSSH says “this key is not known by any other names” rather than warning of a change.
9. When a mismatch happens, SSH refuses to continue in the default configuration, because continuing means handing your credentials to something unverified.
10. If you use password authentication and click through such a warning, you have just typed your password into whatever is in the middle. With key authentication you would have leaked a signature, which is far less valuable, but still not nothing.

■ 42.5.5 TECHNICAL – the engineer’s version

1. `known_hosts` lives at `~/.ssh/known_hosts` for the user and `/etc/ssh/ssh_known_hosts` for the system. Controlled by `UserKnownHostsFile` and `GlobalKnownHostsFile`.
2. Line format: host patterns, key type, base64 key, optional comment. Host patterns may be comma-separated, may use `*` and `?`, and non-default ports are written `[host]:port`.
3. `HashKnownHosts yes` stores hosts as HMAC-SHA-1 hashes so a stolen file does not enumerate every server you use. Debian and Ubuntu enable this by default; upstream OpenSSH does not.
4. `StrictHostKeyChecking` has four values, and choosing correctly matters more than people think:

Value	Behaviour on new host
ask	prompt (upstream default)
accept-new	add, but warn on change
no	add silently, allow change
yes	refuse unless already known

5. `accept-new` was added in OpenSSH 7.6, October 2017, and is the correct choice for scripts. `no` is dangerous because it also tolerates a changed key, which is the case you actually care about.
6. Useful commands, all real:

```
$ ssh-keyscan -t ed25519 -p 2222 127.0.0.1 >> known_hosts
$ ssh-keygen -F '[127.0.0.1]:2222' -f known_hosts
# Host [127.0.0.1]:2222 found: line 1
$ ssh-keygen -R '[127.0.0.1]:2222' -f known_hosts
```

7. A warning that must be stated plainly: `ssh-keyscan` asks the server what its key is. If something is impersonating the server, `ssh-keyscan` will faithfully record the impostor's key.
8. So `ssh-keyscan` is a convenience for populating a file, not a verification tool. Verification means comparing against a fingerprint published by the operator through a different channel.
9. **The GitHub 2023 rotation is the textbook legitimate change.** At approximately 05:00 UTC on 24 March 2023, GitHub replaced its RSA SSH host key.
10. The cause was not a compromise of GitHub's systems. GitHub's RSA SSH private key had been briefly exposed in a public GitHub repository, and they rotated it as a precaution.
11. Every user with an RSA entry for `github.com` in `known_hosts` then saw the REMOTE HOST IDENTIFICATION HAS CHANGED warning. The remedy GitHub published was `ssh-keygen -R github.com` followed by re-adding the new key.
12. Users whose `known_hosts` held the Ed25519 or ECDSA entry saw nothing, because only the RSA key changed.
13. GitHub's current published fingerprints, which you should compare against rather than trusting a prompt:

Type	SHA-256 fingerprint
RSA	uNiVztksCsDhcc0u9e8Bu...
ECDSA	p2QAMXNICITJYWelOtr...
Ed25519	+DiY3wvvV6TuJJhbpZis...

14. The full Ed25519 fingerprint is SHA256:+DiY3wvvV6TuJJhbpZisF/zLDA0zPMSvHdkr4UvC0qU, and the full RSA one is SHA256:uNiVztksCsDhcc0u9e8BuJQXVUpKZIDTMczCvj3tD2s.
15. GitHub publishes these machine-readable at the `/meta` endpoint of its API, under the `ssh_keys` field, which is what automation should consume.
16. When is a changed host key benign? When the operator announced it, when the server was rebuilt or migrated, when you connected to a load-balanced pool with inconsistent keys, or when your DNS now resolves to a different machine because you changed networks.
17. When is it alarming? When nothing changed on your side, when it happens on an untrusted network such as public wifi, when only some connections show it, or when the operator's published fingerprint does not match.
18. The stronger long-term answer is **SSH certificates**. A certificate authority signs host keys, clients trust the CA once with a `@cert-authority` line, and individual host key changes stop mattering. OpenSSH has supported this since version 5.4, March 2010.

■ 42.5.6 WORDS - remember these

`known_hosts` — your notebook of server identities — a client-side file mapping host patterns to accepted host public keys.

Host key — the server's identity key — the long-term key pair `sshd` uses to sign the session identifier during key exchange.

Trust on first use — believing the first answer you get — the TOFU model, where the initial unverified key is pinned and all later ones checked against it.

`ssh-keyscan` — a tool that fetches a server's key — a scanner that records whatever key is offered, with no verification of authenticity.

Session identifier — a fingerprint of this one connection — the exchange hash from the first key exchange, bound into every authentication signature.

`@cert-authority` — trust a signer instead of each server — a `known_hosts` marker line naming a CA key that may vouch for host keys.

42.6 The SSH agent

■ 42.6.1 PLAIN - in simple words

1. A private key should be protected by a passphrase, so that stealing the file is not enough to steal the key.
2. But that means typing a long passphrase every single time you connect.
3. Git makes this unbearable. A fetch, a push, a status check against the server: each one is a fresh connection and a fresh prompt.
4. People solve this the worst possible way. They remove the passphrase.
5. The **SSH agent** is the right solution. It is a small program that runs in the background and holds your unlocked keys in memory.
6. You type the passphrase once. The agent keeps the unlocked key. Every later connection asks the agent instead of asking you.
7. Here is the important detail. The agent does not hand the key to `ssh`.
8. `ssh` sends the agent the data that needs signing. The agent signs it and sends back only the signature.
9. So the key never leaves the agent's memory, not even to another program on your own computer.
10. The agent forgets everything when it stops, which normally means when you log out or shut down.

■ 42.6.2 PLAIN - a picture in your head

1. Think of a notary who keeps your official seal in a locked drawer.
2. You unlock the drawer once in the morning with your combination.
3. All day, whenever you need a document sealed, you hand the document to the notary. He seals it and hands it back.
4. You never carry the seal around, and nobody else ever holds it. They only ever receive sealed documents.
5. At the end of the day the drawer locks itself and the combination is forgotten.
6. Now consider **agent forwarding**, which is the dangerous feature.
7. It means: when I visit another office, let that office phone my notary and ask for seals on their documents.

8. That is wonderfully convenient. It also means anyone with power over that office, while you are visiting, can phone your notary and get anything sealed.
9. They cannot steal the seal. They can use it, on whatever they like, for as long as you are connected.

Where this comparison breaks:

10. A notary would notice being asked to seal something outrageous. The agent signs opaque data and cannot tell a git push from a login to your bank's build server.

■ 42.6.3 PLAIN – a worked example

1. Real output, in order, from the machine used for this chapter.
2. Start an agent and look inside it. Empty.

```
$ eval "$(ssh-agent -s)"
Agent pid 14148
$ ssh-add -l
The agent has no identities.
```

3. Add a key that genuinely has a passphrase. The passphrase is typed once, here and nowhere else.

```
$ ssh-add ~/.ssh/id_pass
Enter passphrase for ~/.ssh/id_pass:
Identity added: ~/.ssh/id_pass (with-passphrase)
$ ssh-add -l
256 SHA256:cY2TWmZ8NY6LT0x9/BYw2D8XxqpmW4+M/C5K7i+nn0I
with-passphrase (ED25519)
```

4. Now connect with no `-i` flag naming a key, and with prompting explicitly disabled so it cannot secretly ask for anything.

```
$ ssh -o BatchMode=yes -p 2222 root@127.0.0.1 \
'echo CONNECTED USING AGENT KEY'
CONNECTED USING AGENT KEY
```

5. That worked with no passphrase typed and no key file named. The agent supplied the signature.
6. Add a key with an automatic expiry, which is a habit worth building:

```
$ ssh-add -t 300 ~/.ssh/id_pass
Identity added: ~/.ssh/id_pass (with-passphrase)
Lifetime set to 300 seconds
```

7. And empty the agent when you walk away:

```
$ ssh-add -D
All identities removed.
```

■ 42.6.4 PLAIN – what is really happening inside

1. The agent listens on a Unix domain socket, which is a file on disk used as a communication end-point between programs on the same machine.
2. Its path is put in the environment variable `SSH_AUTH_SOCK`. That variable is how every other program finds it. There is no other discovery mechanism.
3. When `ssh` needs to authenticate, it connects to that socket and asks: what identities do you hold? The agent replies with public keys only.
4. `ssh` offers those public keys to the server. If the server accepts one, `ssh` builds the data to be signed and sends it to the agent.

6. On macOS, the system agent is integrated with the login keychain. Since macOS 10.12.2 the correct configuration is explicit:

```
Host *
    UseKeychain yes
    AddKeysToAgent yes
    IdentityFile ~/.ssh/id_ed25519
```

7. `UseKeychain yes` stores the passphrase in the login keychain and retrieves it automatically, so the agent is repopulated after a reboot. It is a macOS-only option and will error on Linux.
8. On Linux, `gnome-keyring` and KDE's `ksshaskpass` provide similar integration. On Windows, OpenSSH ships an `ssh-agent` Windows service.
9. The safer alternative to agent forwarding is `ProxyJump`, added in OpenSSH 7.3, August 2016:

```
Host internal
    HostName 10.0.5.20
    ProxyJump bastion.example.com
```

10. `ProxyJump` tunnels a new SSH connection through the intermediate host. The intermediate host sees only encrypted bytes and never touches your agent.
11. A complete, realistic `~/.ssh/config`. This file must be mode 600 and OpenSSH applies the first matching value for each key, so specific hosts must come before `Host *`:

```
# GitHub over the HTTPS port, for blocked networks
Host github.com
    HostName ssh.github.com
    Port 443
    User git
    IdentityFile ~/.ssh/id_ed25519_github
    IdentitiesOnly yes

# A work server on an odd port, through a bastion
Host build
    HostName 10.0.5.20
    Port 2222
    User deploy
    ProxyJump bastion.example.com
    IdentityFile ~/.ssh/id_ed25519_work
    IdentitiesOnly yes

# Everything else
Host *
    AddKeysToAgent yes
    ServerAliveInterval 60
    ServerAliveCountMax 3
    HashKnownHosts yes
```

12. You never have to guess what that file computed. `ssh -G` prints the final resolved settings. Real output from this chapter's machine:

```
$ ssh -G demo
user root
hostname 127.0.0.1
port 2222
identitiesonly yes
identityfile /tmp/sshdemo/.ssh/id_ed25519
```


13. `ServerAliveInterval 60` with `ServerAliveCountMax 3` sends a keepalive every 60 seconds and gives up after three unanswered ones. This is the fix for connections silently dying behind a NAT device that expires idle entries.

■ 42.6.6 WORDS - remember these

SSH agent — a background helper holding unlocked keys — a daemon exposing a signing oracle over a Unix socket, never releasing key material.

`SSH_AUTH_SOCK` — the address of your agent — the environment variable naming the agent's Unix domain socket path.

Agent forwarding — letting a remote host use your agent — an SSH channel exposing your local agent socket on the remote machine, usable by its root.

ProxyJump — hop through a middle machine safely — a client directive tunnelling a fresh SSH session through an intermediate host without agent exposure.

`IdentitiesOnly` — only try the key I named — a directive suppressing the agent's other keys, avoiding Too many authentication failures.

`UseKeychain` — let macOS remember the passphrase — an Apple-specific `ssh_config` option storing and retrieving passphrases from the login keychain.

42.7 Why SSH on port 22 can be blocked when 443 is not

■ 42.7.1 PLAIN - in simple words

1. A port is a number attached to a connection saying which service you want on that machine.
2. Web traffic over HTTPS uses port 443. SSH uses port 22.
3. Many networks do not let you reach every port on the outside world. They allow a short list and block the rest.
4. Port 443 is always on that list. If you block 443, the web stops working, and somebody complains within seconds.
5. Port 22 is often not on the list. Blocking it upsets developers, and only developers, and usually not loudly enough to matter.
6. There are also two real reasons a network operator gives for filtering 22.
7. First, port 22 is scanned constantly by automated programs looking for servers with weak passwords. Filtering it reduces that noise.
8. Second, an outbound SSH connection can carry anything at all inside it, in a form nobody can inspect. It is a perfect way to move data out unseen.
9. So the block is not aimed at you. You are collateral.
10. And the way it fails is cruel. A blocked port 22 usually does not say "no". It says nothing at all, and your command hangs.
11. This is exactly the shape of the reader's own fault in this book, and the fix is one of two lines of configuration.

■ 42.7.2 PLAIN - a picture in your head

1. A building with many numbered delivery bays. Bay 443 is the main goods entrance, open all day, huge queues, everyone uses it.
2. Bay 22 is a small side entrance used by maintenance staff.

3. Management decides to weld bay 22 shut. Almost nobody notices. Bay 443 cannot be welded shut, because then the business stops.
4. Now, welding shut has two possible styles.
5. The polite style is a sign on the door: “closed, use the main entrance”. You read it, turn around, and lose ten seconds.
6. The unhelpful style is no sign at all. You stand there. You knock. You wait. You knock again. Eventually you give up, knowing nothing.
7. Networks overwhelmingly choose the second style, on purpose, because a sign tells a stranger which doors exist.

Where this comparison breaks:

8. A welded door is permanent and obvious on inspection. A network filter can be selective by time, by source address or by destination, so the same command can work for your colleague and hang for you.

■ 42.7.3 PLAIN – a worked example

1. This is real, measured on the machine used to write this chapter, which happens to sit behind exactly this kind of filtering.
2. Port 22 to GitHub, with a generous timeout:

```
$ time ssh -o ConnectTimeout=30 -T git@github.com
ssh: connect to host github.com port 22: Connection timed out

real    0m30.057s
```

3. Thirty seconds, and then a timeout. Nothing came back at any point.
4. Port 443 to the same host name, in the same shell, one second later:

```
$ time bash -c 'exec 3<>/dev/tcp/github.com/443 && echo OPEN'
OPEN

real    0m0.021s
```

5. Twenty-one milliseconds. The network is fine. DNS is fine. GitHub is fine. Only port 22 is unreachable.
6. Now compare that with a port where something actively refuses. Nothing is listening on port 2200 on the local machine:

```
$ time ssh -o ConnectTimeout=10 -p 2200 root@127.0.0.1
ssh: connect to host 127.0.0.1 port 2200: Connection refused

real    0m0.015s
```

7. Fifteen milliseconds and a clear refusal. That is what a healthy “no” looks like.
8. Put the three side by side and the diagnosis writes itself.

Case	Message	Time
Nothing listening	Connection refused	0.015 s
Port 443 reachable	connects	0.021 s
Port 22 filtered	Connection timed out	30.057 s

9. The timing is the tell. Milliseconds means somebody answered. Tens of seconds means silence.

■ 42.7.4 PLAIN – what is really happening inside

- Chapter 34 of this book covered two ways to block a packet, and the difference decides your whole afternoon. Here it is again in one line each.
- REJECT** means refuse and say so. The filter sends back a TCP reset or an ICMP error. Your program fails instantly with “connection refused”.
- DROP** means refuse and say nothing. The packet is deleted. No reply is generated at all.
- With DROP, your computer does not know anything happened. It assumes the packet was lost, so it does what TCP always does: it sends it again.
- The retries are spaced out with doubling gaps. One second, two, four, eight, sixteen. Then it gives up and reports a timeout.
- That is why a blocked port 22 hangs for tens of seconds rather than failing instantly. Every second of that wait is your machine politely retrying.
- Firewall administrators choose DROP deliberately. A REJECT confirms that a host exists at that address, which helps someone mapping your network.
- So silence is a security choice that costs you diagnosis time.
- Applying it to the reader’s own session: `curl -v https://github.com` printed Trying 20.207.73.82:443... and then produced nothing for fifteen seconds. That is the same signature at a different port.
- What that proves is narrow and must be stated carefully. It proves no response arrived. It is consistent with a silent drop somewhere on the path. It does not identify who dropped it, and it does not prove the server was down.
- What made the reader’s case conclusive was the same request succeeding over mobile data seconds later. Same destination, different path, different result.

■ 42.7.5 TECHNICAL – the engineer’s version

- The exact test to determine whether port 22 is blocked, in order, and each step distinguishes something specific.

```
# 1. Does the name resolve? (rules out DNS)
$ dig +short github.com
20.207.73.82

# 2. Can TCP reach port 22 at all? (the real question)
$ nc -vz -w 5 github.com 22

# 3. Same for 443, as a control
$ nc -vz -w 5 github.com 443

# 4. If nc is absent, bash can do it
$ time bash -c 'exec 3<>/dev/tcp/github.com/22'

# 5. GitHub's own SSH check, most direct of all
$ ssh -vT -o ConnectTimeout=10 git@github.com
```

- Read the results this way:

Observation	Conclusion
22 refused fast	reachable, nothing there
22 silent, 443 ok	port 22 filtered
both silent	path or host problem
both fast, auth fails	network fine, key wrong

Observation	Conclusion

3. **Fix one: switch the remote to HTTPS.** One command per repository, or a global rewrite for all of them.

```
$ git remote set-url origin \
    https://github.com/owner/repo.git

# or globally, for every GitHub remote at once
$ git config --global \
    url."https://github.com/".insteadOf "git@github.com:"
```

4. **Fix two: use GitHub's SSH over the HTTPS port.** GitHub runs an SSH endpoint on port 443 at a different host name.
5. The host name is `ssh.github.com`, not `github.com`. This matters. GitHub's documentation states it explicitly because people get it wrong.
6. Test it first:

```
$ ssh -T -p 443 git@ssh.github.com
Hi USERNAME! You've successfully authenticated, but GitHub
does not provide shell access.
```

7. Then make it permanent with the exact block GitHub publishes, in `~/.ssh/config`:

```
Host github.com
    Hostname ssh.github.com
    Port 443
    User git
```

8. After that block, your existing `git@github.com:owner/repo.git` remotes need no change at all. `ssh` rewrites the destination before connecting.
9. This is real SSH, not SSH wrapped in TLS. It is the SSH protocol speaking on TCP port 443. A firewall filtering by port number is satisfied.
10. Therefore it does not help against a proxy that inspects protocols, only against a port-number filter. GitHub's own documentation notes that proxy servers may interfere.
11. Comparison of the two fixes, since they are not equivalent:

Aspect	HTTPS remote	SSH on 443
Credential	token	SSH key
Works via HTTP proxy	yes	usually not
Survives deep inspection	yes	no
Config scope	per repo or global	one config block

12. A third option exists and is worth knowing: `ProxyCommand` with a tool such as `corkscrew` or `nc` tunnels SSH through an HTTP proxy using the `CONNECT` method. It is fiddly and usually not needed if fix one works.
13. On the honest side: none of this is a way around a policy that forbids the traffic. Using SSH on 443 to evade a corporate control is a conversation with your employer, not a technical question.

■ 42.7.6 WORDS – remember these

Port — a number naming a service on a machine — a 16-bit TCP or UDP endpoint identifier; 22 for SSH, 443 for HTTPS by IANA assignment.

DROP — block and say nothing — a filter action discarding the packet with no reply, producing a client-side timeout after retries.

REJECT — block and say so — a filter action returning a TCP RST or ICMP unreachable, producing an immediate Connection refused.

Egress filtering — controlling what may leave a network — outbound firewall policy, commonly permitting only 80, 443 and DNS.

ssh.github.com — GitHub's SSH server on the web port — the host name serving the SSH protocol on TCP 443 for networks blocking 22.

ProxyCommand — a custom way to reach a host — an ssh_config directive running an external program to establish the transport.

42.8 HTTPS authentication for git

■ 42.8.1 PLAIN – in simple words

1. When git talks over HTTPS, it makes ordinary web requests, and it proves who you are the way web requests always have: a user name and a secret.
2. For many years that secret was your actual GitHub account password.
3. That stopped. On 13 August 2021, GitHub stopped accepting account passwords for git operations entirely.
4. The reason is straightforward. Your account password unlocks everything you own, cannot be limited, cannot be given to one script, and cannot be cancelled without changing it everywhere.
5. In its place you use a **token**, which is a long random string that acts like a password but has rules attached to it.
6. A token can be limited to certain actions. It can expire. It can be cancelled on its own without touching anything else.
7. Now the practical problem. You cannot type a forty-character random string before every push.
8. So git has **credential helpers**. A credential helper is a small program that stores your token and hands it back when git asks.
9. Where it stores it depends on the helper, and the differences matter. Some put it in your operating system's encrypted vault. One writes it to a plain text file.
10. Knowing which helper you use, and being able to look inside it and clear it, is a basic skill.

■ 42.8.2 PLAIN – a picture in your head

1. Think of a building where the receptionist checks a pass every time you go through a door.
2. Carrying the pass in your hand and showing it forty times a day is tiring.
3. So you give it to a locker at the entrance. Whenever you approach a door, the locker sends the pass details for you.
4. There are different lockers. A steel one with a combination that only opens when you are logged in. And a cardboard box on the counter with your pass sitting in it, visible to anyone who looks.
5. Both work. Both save you the same effort. Only one is a good idea.
6. And when you lose your pass, or you suspect somebody copied it, you need to do two separate things.

7. Cancel the pass at the security office, so the old one stops working. And empty the locker, so your own building stops trying to use the dead one.

Where this comparison breaks:

8. A physical pass is one object. A token can be copied silently and used from anywhere in the world at the same time as you are using it, with no sign at the door.

■ 42.8.3 PLAIN – a worked example

1. Real output from the machine used for this chapter, using git's own `credential` commands. These are the exact operations a helper performs.
2. Choose a helper. Here the insecure store helper is chosen deliberately, so we can look at what it writes:

```
$ git config --global credential.helper \  
    'store --file=/tmp/demo/.git-credentials'
```

3. Save a credential. This is what happens automatically the first time you push and type a token:

```
$ printf 'protocol=https\nhost=github.com\nusername=reader  
password=ghp_EXAMPLEexampleEXAMPLEexample1234\n\n' \  
| git credential approve
```

4. Now look at what landed on disk:

```
$ ls -l /tmp/demo/.git-credentials  
-rw----- 1 root root 63 Aug 13 08:24 .git-credentials  
$ cat /tmp/demo/.git-credentials  
https://reader:ghp_EXAMPLEexampleEXAMPLE...@github.com
```

5. That is your token, in a plain readable file. The permissions are 600, owner only, but there is no encryption at all.
6. Ask git what it would send for a given host. This is the command to run when you suspect a stale credential:

```
$ printf 'protocol=https\nhost=github.com\n\n' \  
| git credential fill  
protocol=https  
host=github.com  
username=reader  
password=ghp_EXAMPLEexampleEXAMPLEexample1234
```

7. And clear it, which is what you do the moment a token is rotated or leaked:

```
$ printf 'protocol=https\nhost=github.com\nusername=reader  
password=ghp_EXAMPLEexampleEXAMPLEexample1234\n\n' \  
| git credential reject  
$ wc -c < /tmp/demo/.git-credentials  
0
```

8. The file is now empty. Note carefully: this cleared the local copy. It did nothing whatsoever to the token on GitHub's side.

■ 42.8.4 PLAIN – what is really happening inside

1. Git never asks a helper for a credential until it needs one.
2. It first makes the request without any credential. If the server is happy, nothing else happens. This is why cloning a public repository never prompts.
3. If the server responds with HTTP status 401 and a WWW-Authenticate header, git knows a credential is required.
4. Git then runs the configured helper with the subcommand `get`, feeding it the protocol, host and path on standard input.
5. The helper replies with `username=` and `password=` lines, or says nothing if it has none stored.
6. If no helper has one, git prompts you on the terminal.
7. Git then retries the request with an `Authorization: Basic` header. That header is the user name and token joined with a colon and base64-encoded.
8. Base64 is an encoding, not encryption. It hides nothing. What protects the token is that the whole request travels inside TLS.
9. If the retried request succeeds, git runs the helper again with `store`, so a working credential is remembered.
10. If it fails with 401, git runs the helper with `erase`, so a wrong credential is not offered forever.
11. That last step is why a bad cached token often self-corrects, and why a token that is valid but lacks a permission does not: that is a 403, not a 401, so git does not erase it.

```
git ---> GET /repo.git/info/refs      ---> server
git <--- 401 + WWW-Authenticate      <--- server
git ---> helper "get" (host, protocol)
git <--- helper "username=..., password=..."
git ---> GET again + Authorization: Basic ...
git <--- 200 OK
git ---> helper "store" (remember this one)
```

■ 42.8.5 TECHNICAL – the engineer's version

1. The credential helper protocol is a documented text protocol. A helper is any executable named `git-credential-NAME`, invoked with one of `get`, `store` or `erase`.
2. The helpers you will actually meet:

Helper	Storage	Encrypted
osxkeychain	macOS keychain	yes
manager (GCM)	OS vault per platform	yes
libsecret	GNOME keyring	yes
wincred	Windows Cred Manager	yes
cache	memory, 900 s default	n/a
store	~/.git-credentials	no

3. `store` is documented by git itself as insecure. Its own manual page says the credentials are stored unencrypted. Use it only on a machine where the whole disk is encrypted and single-user, and preferably not then.
4. `cache` keeps credentials in memory in a background daemon with a default timeout of 900 seconds, adjustable with `--timeout`.
5. Git Credential Manager, formerly Git Credential Manager Core, is the cross-platform helper maintained by GitHub. It handles OAuth device flow, so you get a browser login instead of pasting a token.

6. Configuration is layered, and this catches people out:

```
$ git config --show-origin --get-all credential.helper
file:/etc/gitconfig      manager
file:~/.gitconfig       osxkeychain
```

7. Multiple helpers are tried in order until one returns a credential. An empty value, `credential.helper=`, resets the list, which is the way to disable an inherited system-level helper.
8. Per-host configuration is possible and is the right way to keep work and personal accounts separate:

```
[credential "https://github.com"]
    username = work-account
    helper = osxkeychain
```

9. Inspecting and clearing on macOS, which is the reader's platform:

```
# see it
$ printf "protocol=https\nhost=github.com\n\n" \
| git credential-osxkeychain get

# delete it
$ printf "protocol=https\nhost=github.com\n\n" \
| git credential-osxkeychain erase
```

10. The graphical route on macOS is the Keychain Access application, searching for `github.com` and deleting the internet password entry.
11. On Windows: `cmdkey /list` in a command prompt, or Credential Manager in Control Panel, under Windows Credentials.
12. On Linux with libsecret: `secret-tool search server github.com`.
13. The HTTP conversation is worth seeing once. Real capture from this chapter's machine against a public repository, so no credential was needed:

```
GET /octocat/Hello-World.git/info/refs?service=git-upload-pack
Git-Protocol: version=2
<= HTTP/1.1 200 OK
<= Content-Type: application/x-git-upload-pack-advertisement
```

14. Against a repository the requester cannot see, unauthenticated, the same endpoint returns 401 rather than 404. Real result:

```
$ curl -s -o /dev/null -w "status=%{http_code}\n" \
"https://github.com/octocat/not-a-real-repo.git/info/refs\
?service=git-upload-pack"
status=401
```

15. That 401 for a repository that does not exist is deliberate. GitHub will not confirm or deny the existence of a private repository to an anonymous requester. Remember this when we reach Repository not found.
16. The user name in an HTTPS git URL is almost always ignored by GitHub when a token is supplied. `x-access-token`, your login, or anything else usually works. The token identifies you.
17. Never put a token in the remote URL itself. It lands in `.git/config` in plain text, in your shell history, and in the output of `git remote -v`, which people paste into bug reports.

■ 42.8.6 WORDS - remember these

Credential helper — a program that remembers your token — an executable implementing git's get/store/erase credential protocol.

Basic authentication — user name and secret sent in a header — HTTP authentication, RFC 7617, sending base64 of user:secret, safe only inside TLS.

401 versus 403 — “who are you” versus “not allowed” — 401 means no or invalid credential and triggers a prompt; 403 means the credential is valid but insufficient.

store helper — the one that writes plain text — a helper persisting credentials unencrypted in ~/.git-credentials.

Keychain — the operating system's encrypted vault — macOS Keychain, Windows Credential Manager or a libsecret provider, unlocked with your login.

git credential fill — ask git what it would send — a plumbing command running the helper chain and printing the resolved credential.

42.9 Personal access tokens

■ 42.9.1 PLAIN - in simple words

1. A token is a long random string that stands in for you when a program talks to a service.
2. It is not derived from your password. Cancelling it does not affect your password, and changing your password does not necessarily cancel it.
3. That independence is the point. A token is a separate identity you can hand to one script, on one machine, for one purpose.
4. A token has three properties a password does not have, and each one matters.
5. It can be **scoped**, meaning limited to a specific list of things it is allowed to do.
6. It can **expire**, meaning it stops working on a date you chose, whether or not you remember it exists.
7. It can be **revoked individually**, meaning you cancel this one without disturbing the other nine tokens on your other nine machines.
8. GitHub offers two kinds. The older **classic** tokens, which use a list of broad permission names. And **fine-grained** tokens, which pick individual repositories and individual permissions.
9. A leaked token is an emergency, not an inconvenience. Anyone holding it is you, for everything that token can reach, immediately.
10. There is no clever recovery. You revoke it, you create a new one, you replace it everywhere, and you check what was done while it was out.

■ 42.9.2 PLAIN - a picture in your head

1. Think of a hotel with two kinds of key.
2. The master key opens every room, the safe, the office and the roof. That is your account password.
3. The other is a plastic card cut for room 412 only, valid until Friday, and the front desk can cancel it in one click without recutting anyone else's.
4. If a cleaner needs into room 412 on Tuesday, you give them the card, not the master key.
5. If the card is lost, you cancel it. Nothing else changes. Nobody else is inconvenienced.
6. If the master key is lost, the entire hotel has to be rekeyed.
7. Now add the part that has no hotel equivalent. A copied card is invisible. The lock cannot tell the difference and neither can you.

8. So the card also has a printed expiry date, so that a copy you never learned about dies on its own.

Where this comparison breaks:

9. A hotel card must be physically present at the door. A token can be used from another continent, by a program, thousands of times a minute.
10. And a hotel lock logs the card number. Token use is logged too, but you have to go and look, and almost nobody does until something breaks.

■ 42.9.3 PLAIN – a worked example

1. GitHub tokens begin with a short prefix that tells you what kind they are. This is a deliberate design so that scanners can find them.

Prefix	What it is
ghp_	classic personal token
github_pat_	fine-grained personal token
gho_	OAuth app user token
ghu_	GitHub App user token
ghs_	GitHub App server token
ghr_	refresh token

2. Using one at the command line, once, without saving it anywhere:

```
| $ git clone https://TOKEN@github.com/owner/repo.git
```

3. That works, and it is a bad habit, because the token is now in the shell history and in `.git/config`.
4. The better single-use form pipes it in and never stores it:

```
| $ echo "TOKEN" | gh auth login --with-token
```

5. Rotating a token is a fixed procedure. Do these in order, because the order prevents an outage:
6. One: create the new token with the same scopes, and note the expiry.
7. Two: update every place that uses it. Your keychain, your CI secrets, the `.netrc` on the build box, the deployment config.
8. Three: verify the new one works, with an actual operation, not a guess.
9. Four: only now, revoke the old one.
10. Five: watch for failures for a day, because there is always one forgotten machine.
11. If instead the token has leaked, that order inverts. Revoke first, immediately, and accept the outage. A live leaked token is worse than downtime.

■ 42.9.4 PLAIN – what is really happening inside

1. A token is a record in GitHub's database. It has an owner, a creation time, an expiry time, a list of permissions and a last-used timestamp.
2. What you are shown once, at creation, is the token string. GitHub does not store that string in a readable form; it stores a hash of it.
3. That is why the page says you will never see it again. They genuinely cannot show it to you.
4. When you use it, the server hashes what you sent and looks up the record. If the record exists, is not expired and is not revoked, you are authenticated.
5. Then, and only then, does it check whether the permissions on that record allow the specific operation you asked for. Those are two separate steps.

6. Keeping them separate explains the whole of section 42.10. Being authenticated does not mean being allowed.
7. **Secret scanning** is GitHub scanning content pushed to it, looking for strings that match known credential patterns.
8. This is why the prefixes exist. `ghp_` followed by the right number of base62 characters with a checksum is machine-recognisable with almost no false positives.
9. GitHub runs the same scanning for over five hundred partner patterns: cloud provider keys, payment provider keys, and so on.
10. When GitHub finds its own token in a public repository, it revokes it automatically. When it finds a partner's, it notifies the partner, who typically revokes it too.
11. **Push protection** is the same detection moved earlier. Instead of finding the secret after it is pushed, the push itself is refused.
12. That distinction is enormous. A secret that reached the server must be treated as leaked forever, even after you rewrite history, because the object may have been fetched or cached.
13. A secret blocked by push protection never arrived. Nothing leaked. You fix your commit and move on.
14. Push protection has been enabled by default for public repositories since 2024. For private repositories it is part of GitHub's paid secret protection product.

■ 42.9.5 TECHNICAL - the engineer's version

1. Classic versus fine-grained, precisely:

Property	Classic	Fine-grained
Scope model	broad named scopes	per-permission
Repository limit	all you can access	chosen list
Owner limit	all orgs	one user or org
Org approval	no	can be required
Expiry	optional	policy-controlled

2. GitHub's own documentation states the case bluntly: a classic token "will grant access to all repositories within the organizations that you have access to, as well as all personal repositories".
3. That sentence is the whole argument for fine-grained tokens. A classic token on a laptop is a key to everything that laptop's owner can reach.
4. Fine-grained tokens are limited to resources owned by a single user or organization, can be limited to specific repositories, and are granted specific permissions rather than broad scopes.
5. Organization owners can require approval before a fine-grained token may act on organization resources, which is a control classic tokens never had.
6. Expiry: GitHub supports preset periods and a custom date. Fine-grained tokens gained optional non-expiring lifetimes and rotation policies in a change announced on 18 October 2024, and enterprises can enforce maximum lifetimes.
7. GitHub automatically removes personal access tokens that have not been used for a year. That is a useful backstop, not a strategy.
8. Practical expiry guidance, and this is a convention rather than a rule:

Use	Suggested lifetime
Interactive laptop use	90 days

Use	Suggested lifetime
A one-off script	7 days
A long-running service	do not; use an App
CI pipeline	do not; use OIDC

9. Token format, since GitHub's April 2021 redesign: a fixed prefix, an underscore, then base62 characters, ending in a checksum. The checksum lets a scanner reject a random-looking string cheaply before doing any lookup.
10. Auditing what a token can do, without guessing. The response headers of an authenticated API call name the scopes:

```
$ curl -sI -H "Authorization: Bearer $TOKEN" \
  https://api.github.com/user | grep -i x-oauth
x-oauth-scopes: repo, read:org
x-accepted-oauth-scopes:
```

11. x-oauth-scopes is what your token has. x-accepted-oauth-scopes is what that particular endpoint would accept. Comparing the two diagnoses most permission failures in one command.
12. With the GitHub CLI the equivalent is `gh auth status`, which prints the scopes of the stored token.
13. Leak response, in order, and treat this as a checklist:
14. Revoke the token in GitHub settings. This is instant and unconditional.
15. Rotate anything the token could have reached: deploy keys, downstream secrets, database passwords stored in repositories it could read.
16. Review the account's security log and, for organizations, the audit log, filtering by the period the token was live.
17. Check for new SSH keys, new deploy keys, new webhooks, new collaborators and changed workflow files. Those are the standard persistence mechanisms.
18. Rewriting git history does not undo a leak. Assume the secret is public from the moment it was pushed.
19. Never commit a token, even to a private repository. Private repositories get forked, made public, and cloned onto laptops that later get lost.

■ 42.9.6 WORDS - remember these

Personal access token (PAT) — a password substitute you can limit and cancel — a server-side credential record with scopes, expiry and independent revocation.

Classic token — the older broad kind — a PAT carrying OAuth-style scopes across every repository and organization the user can reach.

Fine-grained token — the newer narrow kind — a PAT limited to one owner, a chosen repository list and individual permissions.

Revocation — cancelling one credential — deleting the server-side token record, taking effect immediately and independently of other credentials.

Secret scanning — the service that hunts for leaked credentials — pattern matching over pushed content, with automatic revocation for GitHub's own tokens.

Push protection — refusing the push instead of cleaning up after it — pre-receive secret detection that blocks the write, so nothing ever leaks.

42.10 Scopes, properly

■ 42.10.1 PLAIN – in simple words

1. A **scope** is a named permission attached to a credential.
2. The name is a short string, like `repo` or `workflow`. It sits in the token's record on the server, not in anything you hold.
3. The server checks it at the exact moment you try to do something. Not when you log in. At the moment of the action.
4. That timing is the thing to understand. You can be perfectly, provably authenticated and still be refused.
5. Authentication answers "who are you". Authorization answers "may you do this". They are separate questions asked at different times.
6. So a token with no `workflow` scope will happily let you clone, fetch, and push ordinary changes. It will refuse one specific kind of push.
7. The reader hit exactly this. A push that changed a file under `.github/workflows/` was refused, while everything else worked.
8. The reason is not arbitrary. A file under `.github/workflows/` is not ordinary content. It is instructions for GitHub's own machines.
9. Those machines run with access to the repository's secrets. So changing that file is changing what code runs with those secrets.
10. GitHub therefore treats it as a separate, higher permission that you must ask for on purpose.

■ 42.10.2 PLAIN – a picture in your head

1. Imagine an office pass that opens the front door, the stairs and your own floor.
2. You walk in, you go up, you work. Everything is fine. You are definitely a recognised employee.
3. Then you try the server room. The reader flashes red. Not because your pass is fake, but because your pass does not include that room.
4. Nobody has questioned your identity. They know exactly who you are. The answer is still no.
5. Now the important part: why is the server room separate at all?
6. Because inside it you could reconfigure the machines that everyone else's work runs on, and give yourself access to everything they touch.
7. That is precisely the relationship between a normal file and a CI workflow file. One is a document. The other is a change to the factory.

Where this comparison breaks:

8. A door either opens or it does not, and it decides in one instant. A scope check can depend on the content of what you are pushing, which path it touches, and even whether an identical file already exists on another branch.

■ 42.10.3 PLAIN – a worked example

1. The scopes that actually matter in daily work. These are the exact names GitHub uses.

Scope	What it grants
<code>repo</code>	full read/write, private too
<code>public_repo</code>	same, public repos only
<code>workflow</code>	add/update workflow files
<code>read:org</code>	read org and team membership

Scope	What it grants
write:packages	publish packages
read:packages	download packages
admin:repo_hook	manage repository webhooks
gist	write access to gists

2. GitHub's own words for the two that cause the most confusion.
3. repo "grants full access to public and private repositories including read and write access to code, commit statuses, repository invitations, collaborators, deployment statuses, and repository webhooks".
4. workflow "grants the ability to add and update GitHub Actions workflow files". And then a detail almost nobody knows, quoted exactly.
5. "Workflow files can be committed without this scope if the same file (with both the same path and contents) exists on another branch in the same repository."
6. That exception explains a confusing symptom. Merging a branch that already contains the workflow file can succeed while the original push failed.
7. Note that repo does not include workflow. It is not a hierarchy. The broadest normal scope still stops at the workflow directory.
8. read:org is the one people forget. Without it, a token cannot list the organizations you belong to, so tools report you as having no organizations at all.

■ 42.10.4 PLAIN – what is really happening inside

1. Here is the reader's exact failure, and the message is worth reading character by character.

```
$ git push origin main
Enumerating objects: 7, done.
Writing objects: 100% (4/4), 412 bytes, done.
To https://github.com/owner/repo.git
! [remote rejected] main -> main (refusing to allow a
  Personal Access Token to create or update workflow
  `.github/workflows/ci.yml` without `workflow` scope)
error: failed to push some refs to
'https://github.com/owner/repo.git'
```

2. Read what already succeeded before the refusal, because it tells you where the failure was not.
3. Enumerating objects and Writing objects both completed. The network worked. TLS worked. Authentication worked.
4. Your commits were transmitted to GitHub. All of them. The bytes arrived.
5. The refusal came from git-receive-pack on GitHub's side, after receiving everything, when it inspected what was about to be applied.
6. Then it declined to update the branch reference. Nothing was written. The objects were transmitted and discarded.
7. That is what ! [remote rejected] means, and it is why this is not a transport failure. A transport failure cannot produce a sentence about scopes, because a broken transport cannot produce sentences at all.
8. The phrasing refusing to allow is also deliberate. GitHub is not saying it failed. It is saying it decided.
9. Here is the same shape reproduced locally on the machine used for this chapter, with a server-side hook standing in for GitHub's own check. Real output:


```
$ git push origin main
remote: error: refusing to allow a Personal Access Token
remote: to create or update workflow
remote: `.github/workflows/ci.yml` without `workflow` scope
To /tmp/gitgate/srv/repo.git
! [remote rejected] main -> main (pre-receive hook declined)
error: failed to push some refs to '/tmp/gitgate/srv/repo.git'
```

10. And the push immediately before it, which touched only README.md, succeeded:

```
$ git push origin main
To /tmp/gitgate/srv/repo.git
* [new branch]      main -> main
```

11. Same server, same credential, same network, same second. The only variable was which file the commit touched.
12. That comparison is the proof that this is a content-dependent authorization decision, taken on the server.

■ 42.10.5 TECHNICAL – the engineer’s version

1. The fix, in order, and it is short:
2. Open GitHub, Settings, Developer settings, Personal access tokens, Tokens (classic). Select the token in use.
3. Tick workflow. Save. The scope is added to the existing token; you do not have to create a new one for classic tokens.
4. If the token was already cached, git will keep using the same string, and it now has the scope. No local change is needed.
5. If you create a new token instead, clear the cached one first, or git will keep sending the old one:

```
$ printf "protocol=https\nhost=github.com\n\n" \
| git credential reject
```

6. With the GitHub CLI, re-authorize with the extra scope in one command:

```
$ gh auth refresh -h github.com -s workflow
```

7. For fine-grained tokens, the equivalent is the repository permission “Workflows: Read and write”.
8. For a GitHub App, it is the workflows repository permission, and the app must be installed on the repository.
9. Verify before retrying, rather than pushing and hoping:

```
$ curl -sI -H "Authorization: Bearer $TOKEN" \
https://api.github.com/user | grep -i x-oauth-scopes
x-oauth-scopes: repo, workflow
```

10. **Now generalise, because all three failures print “push failed”.** This table is the most useful thing in the chapter.

Layer	Symptom	Proof
Network	hangs then times out	no bytes sent
Auth	asks again, or 401	Invalid username
Authorization	refused with a reason	[remote rejected]
Non-fast-forward	refused, no credential talk	[rejected]

11. Distinguish them by four observations, in this order.
12. Did objects transfer? Writing objects: 100% means the network and the credential both worked. Everything after that is a policy decision.
13. Was a credential requested? A prompt or a 401 means authentication failed, not authorization.
14. Is there a reason in parentheses after ! [remote rejected]? A sentence means a server rule. A bare [rejected] with non-fast-forward means your branch is behind and no permission is involved.
15. How long did it take? Instant means somebody answered. Fifteen to thirty seconds of silence means the network.
16. A worked triage, applying the table to the four cases:

```
Case A: 30 s, "Connection timed out"
-> network. Port filtered. See 42.7.

Case B: "remote: Invalid username or password"
-> authentication. Token wrong, expired or revoked.

Case C: "! [remote rejected] ... without `workflow` scope"
-> authorization. Credential fine. Add the scope.

Case D: "! [rejected] ... (non-fast-forward)"
-> neither. Fetch and rebase or merge.
```

17. One further separation the reader met in the same session. Plain git kept working while the GitHub API did not.
18. That is the same split at a larger scale: git the tool, running locally against local objects, versus GitHub the service, reachable only over the network and enforcing its own rules.
19. The honest version: the workflow scope is not a complete defence. Someone with repo scope can still change a file that a workflow executes, such as a build script the workflow calls. The scope protects the workflow definition, not everything the workflow does.
20. That is why serious repositories also use branch protection, required reviews on the .github/ path with a CODEOWNERS file, and environment protection rules on the jobs that hold real secrets.

■ 42.10.6 WORDS - remember these

Scope — a named permission on a credential — a string in the token record that the server checks per operation, such as repo or workflow.

Authentication — establishing who you are — verifying a credential against a stored record; failure produces 401.

Authorization — establishing what you may do — checking permissions for a specific operation; failure produces 403 or a rejection message.

! [remote rejected] — the server chose not to apply this — a per-ref failure from git-receive-pack, always accompanied by a reason.

Non-fast-forward — your branch is behind, not forbidden — a rejection because the update would discard commits, unrelated to any permission.

x-oauth-scopes — the header that lists what your token has — a GitHub API response header naming the token's granted scopes.

42.11 OAuth, tokens, SSH keys and GitHub Apps

■ 42.11.1 PLAIN – in simple words

1. There are four different ways a program or a person can be identified to GitHub, and people mix them up constantly.
2. An **SSH key** identifies a machine you sit at. It lives in a file on that machine and never moves.
3. A **personal access token** identifies you as a person, to a program you gave it to. It is a string, so it can be copied.
4. An **OAuth authorization** is what happens when you click “sign in with GitHub” on someone else’s website. You grant that site permission to act as you.
5. A **GitHub App** is a separate identity of its own. It is not you. It is installed on specific repositories and acts in its own name.
6. The difference that matters most is **blast radius**: if this credential leaks, how much can the finder reach, and for how long.
7. An SSH key with no passphrase, sitting on a stolen laptop, reaches every repository you can reach, forever, until you notice.
8. A classic token reaches the same, from anywhere in the world, with no laptop required.
9. A GitHub App installation token reaches only the repositories the app is installed on, only with the permissions granted, and it dies after one hour.
10. For automated systems there is a fifth and better option that involves no stored secret at all. We come to it at the end of this section.

■ 42.11.2 PLAIN – a picture in your head

1. Four ways to get a package delivered into a building.
2. The SSH key is a key cut for your own front door. It works only when you are standing there. Losing it is bad but local.
3. The personal token is your signature on a delivery authorization. Anyone with a photocopy can use it anywhere, until you cancel it.
4. The OAuth grant is you signing a standing instruction that a courier company may collect on your behalf. You can withdraw it, but while it stands, they act as you.
5. The GitHub App is a contractor with their own uniform, their own badge, and a written list of which rooms they may enter. When they do something, the log says the contractor did it, not you.
6. That last property is underrated. When a person leaves the company, their key, token and OAuth grants all die with their account. The contractor keeps working.

Where this comparison breaks:

7. A contractor can be phoned and asked what they are doing. An app acts in milliseconds, thousands of times, and the only record is the audit log.

■ 42.11.3 PLAIN – a worked example

1. The four identities compared on the properties that decide which to use.

Identity	Acts as	Typical lifetime
SSH key	you	years, until removed
Classic PAT	you	30 to 90 days
OAuth grant	you, via an app	until revoked
App install token	the app	1 hour

2. And the blast radius if each one leaks:

Identity	Reach if leaked
SSH key	all repos you can access
Classic PAT	all repos, plus scopes
Fine-grained PAT	listed repos only
App install token	installed repos, 1 hour

3. Choosing, as a short decision list:

4. A human on their own laptop, working daily: an SSH key with a passphrase, held in the agent.
5. A human running a script occasionally: a fine-grained token, short expiry, limited to the repositories the script touches.
6. A server that must pull one repository: a deploy key, read-only, on that server only.
7. A product integrating with GitHub for many customers: a GitHub App.
8. A CI pipeline talking to a cloud provider: neither of these. Use OIDC, which is the next block.

■ 42.11.4 PLAIN - what is really happening inside

1. The problem with a long-lived credential in a CI system is that it must be stored somewhere the CI system can read.
2. Which means anyone who can make the CI system print things, or run arbitrary code, can read it. And CI systems exist to run code that people wrote today.
3. The modern answer removes the stored secret entirely. It is called **OIDC federation**, and the idea is simple even if the acronyms are not.
4. When a job starts, GitHub creates a short-lived signed statement describing that exact job. Which repository, which branch, which workflow, which run.
5. That statement is signed by GitHub, and anyone can check the signature against GitHub's published verification keys.
6. The job hands that statement to the cloud provider and says: I am this workflow in this repository, please give me credentials.
7. The cloud provider has been told in advance which statements to accept. For example, only jobs from owner/repo on the main branch.
8. It verifies the signature, checks the description matches its rule, and issues its own short-lived credential, valid for perhaps an hour.
9. Nothing long-lived was ever stored. There is no secret in the repository, no secret in the CI settings, and nothing to rotate.
10. If someone steals the statement, it expires in minutes and only works for that one narrow rule.

```

GitHub Actions job
| 1. request token for audience "aws"
v
GitHub OIDC issuer
| 2. signed JWT: repo, ref, workflow, run_id
v
the job
| 3. present JWT to the cloud provider
v
cloud provider
| 4. verify signature, match trust policy
| 5. issue short-lived credentials
v
the job now has 1-hour credentials, no stored secret

```

■ 42.11.5 TECHNICAL – the engineer’s version

1. **SSH keys** authenticate a client to a server using public key cryptography. On GitHub they are attached to a user account or, as deploy keys, to a repository. They carry no scopes: an account SSH key can do whatever that account can do over git.
2. That is a genuine limitation. There is no such thing as a read-only account SSH key on GitHub. If you need read-only, you need a deploy key or a fine-grained token.
3. **Personal access tokens** are bearer credentials. Possession is authorization. They carry scopes and expiry as covered in 42.9 and 42.10.
4. **OAuth** is the delegation framework in RFC 6749, October 2012, with the current best-practice profile in RFC 9700, January 2025. Three parties: you the resource owner, the app, and GitHub the authorization server.
5. The user-facing difference from a token is that you never hand the app your credential. You are redirected to GitHub, you approve a named list of scopes, and the app receives a token bound to that grant.
6. OAuth App tokens act entirely as you. Every action appears in the audit log as your action. Revoking the grant kills the token.
7. **GitHub Apps** are first-class actors. Key properties:

Property	Value
Identity	the app, not a user
Install scope	chosen repositories
Permissions	per-resource read/write
Install token life	1 hour
Rate limit	scales with installation

8. A GitHub App authenticates in two stages. It signs a JSON Web Token with its own private key to prove it is the app, then exchanges that for an installation access token scoped to one installation.
9. That installation token expires after one hour, which is stated in GitHub’s own documentation as one of the reasons to prefer apps over deploy keys.
10. Apps also survive staff turnover. A token or key tied to a departing employee dies when the account is deprovisioned, usually at three in the morning, in the middle of a deployment.
11. **OIDC federation for CI.** OpenID Connect is an identity layer on top of OAuth 2.0. GitHub Actions runs an OIDC provider that issues a JSON Web Token per job.
12. Claims in that token include sub, which encodes the workflow context such as repo:octo-org/octo-repo:environment:prod, plus aud, repository, actor, ref, workflow and run_id.
13. The cloud side is configured with a trust policy matching those claims. A correct policy pins at least the repository and usually the branch or environment.
14. GitHub states three benefits: no cloud secrets duplicated as long-lived GitHub secrets, granular control through the provider’s own access tools, and automatic rotation because each token expires when the job ends.
15. A minimal workflow needs one permission block and one action:

```
permissions:  
  id-token: write  
  contents: read
```

16. Without id-token: write the job cannot request an OIDC token at all, and the failure message is unhelpful, so this is worth memorizing.

17. `GITHUB_TOKEN`, the automatic token inside every Actions run, is itself an installation token for a GitHub App. Its permissions are set by the `permissions: key`, and any action in the job can read it via `github.token` even if you never pass it.
18. A common trap follows from that. The workflow scope discussion in 42.10 applies here too: `GITHUB_TOKEN` deliberately cannot push changes to workflow files, and cannot trigger further workflow runs, to prevent infinite loops and privilege escalation.
19. The honest version, and it is a real trade-off. OIDC federation is more setup than pasting a token. For a one-person hobby project a short-lived fine-grained PAT is a reasonable choice. For anything with real secrets, the setup cost is paid once and removes a whole class of incident.

■ 42.11.6 WORDS – remember these

Bearer credential — whoever holds it, is you — a token requiring no proof of possession beyond presenting it.

OAuth — letting an app act for you without your password — the delegation framework of RFC 6749, exchanging a user approval for a scoped token.

GitHub App — a robot with its own identity and badge — an installable actor with per-repository permissions and one-hour installation tokens.

Installation access token — an app's short-lived key to one installation — a token valid for 1 hour, scoped to the repositories the app is installed on.

OIDC federation — proving who you are instead of holding a secret — exchanging a signed job description for short-lived cloud credentials.

Blast radius — how much damage one leak causes — the set of resources and the time window a compromised credential can affect.

42.12 Practical hardening for a developer's own machine

■ 42.12.1 PLAIN – in simple words

1. Most security advice is too long to follow. Here is the short version that actually pays for itself.
2. Put a passphrase on every private key. It is the difference between a stolen laptop being embarrassing and being catastrophic.
3. Use the SSH agent so the passphrase costs you one typing per day, not one per push.
4. Make a different key on every machine. Never copy a private key between machines, not even your own.
5. The reason is simple. If you lose one machine, you remove one key. If you copied one key everywhere, you have to replace it everywhere at once, in a hurry, while panicking.
6. Look at your list of keys and tokens once or twice a year and delete the ones you cannot account for.
7. On any server you run yourself, turn password logins off entirely. Keys only.
8. Install a small program that bans addresses which repeatedly fail to log in. It is not clever, but it removes most of the noise.
9. Moving SSH to a port other than 22 stops the automated scanning almost completely. Be honest that this hides the door, it does not lock it.
10. And know in advance what you will do when a key or token leaks, because the first ten minutes decide how bad it gets.

■ 42.12.2 PLAIN – a picture in your head

1. Think of the difference between a bicycle lock and a bicycle in a locked shed.
2. A passphrase on your key is the lock. Someone who takes the bicycle still cannot ride away immediately.
3. Separate keys per machine is owning three cheap bicycles instead of one expensive one. Losing one is annoying, not ruinous.
4. Disabling password login on your server is removing the letterbox that people keep pushing lock-picks through.
5. Banning repeat offenders is a doorman who remembers faces.
6. Changing the port is painting the shed to look like a hedge. Any determined person who walks the whole street still finds it.
7. That last one is worth doing anyway, not because it stops attackers, but because it stops your log file filling with thousands of pointless entries and hiding the one that matters.

Where this comparison breaks:

8. A thief must be physically present. An attacker scans the entire internet's address space in hours, so "nobody would bother with my little server" is simply false. Everything is scanned, all the time, automatically.

■ 42.12.3 PLAIN – a worked example

1. A complete setup for a new machine, in order, with real commands.

```
# 1. one key, this machine, with a passphrase
$ ssh-keygen -t ed25519 -C "you@laptop-2026"

# 2. keep it in the agent so you type it once
$ ssh-add -t 8h ~/.ssh/id_ed25519

# 3. check what the agent holds
$ ssh-add -l
256 SHA256:WH1E2gSLhG6FgMsha//GINoq... (ED25519)

# 4. add the public half to GitHub, then test
$ ssh -T git@github.com
```

2. A minimal safe ~/.ssh/config for that machine:

```
Host *
    AddKeysToAgent yes
    IdentitiesOnly yes
    ServerAliveInterval 60
    HashKnownHosts yes
```

3. On a server you own, the three lines in /etc/ssh/sshd_config that matter most:

```
PasswordAuthentication no
KbdInteractiveAuthentication no
PermitRootLogin prohibit-password
```

4. Always test in a second terminal before closing the first one. Locking yourself out of a remote machine is a rite of passage that is best skipped.
5. Adding a passphrase to a key that has none, without regenerating it:


```
$ ssh-keygen -p -f ~/.ssh/id_ed25519
Enter old passphrase:
Enter new passphrase:
```

6. Checking whether a key file is encrypted at all. An unencrypted OpenSSH private key contains the string `none` twice near the start of the base64, for the cipher and the key derivation function.

■ 42.12.4 PLAIN – what is really happening inside

1. A passphrase does not scramble the whole key file with the passphrase directly. That would be weak, because people choose short passphrases.
2. Instead the passphrase is fed through a slow, deliberately expensive function to derive an encryption key. Then that derived key encrypts the private key.
3. Slow is the point. If each guess costs a fraction of a second instead of a microsecond, a guessing attack becomes hundreds of thousands of times more expensive.
4. When you type the passphrase, the same derivation runs, the file is decrypted in memory, and the plain key is handed to the agent.
5. From then until the agent is emptied or stopped, the unlocked key exists in that process's memory and nowhere else on disk.
6. A ban tool such as `fail2ban` works by reading the authentication log, matching failure lines with a pattern, counting them per source address, and inserting a firewall rule when a threshold is crossed.
7. It is not intelligent and it does not need to be. Automated scanners do not adapt; they move to the next address.
8. Changing the port works for the same unglamorous reason. Mass scanners prioritise port 22 because that is where the yield is. A different port is still found by a full port scan, just far less often.

■ 42.12.5 TECHNICAL – the engineer's version

1. Key encryption in the OpenSSH format uses `bcrypt_pbkdf`, a password-based key derivation function built on `bcrypt`, introduced with the new private key format in OpenSSH 6.5, January 2014.
2. The work factor defaults to 16 rounds and is adjustable at generation time:

```
$ ssh-keygen -t ed25519 -a 100 -C "you@laptop"
```

3. `-a 100` sets 100 KDF rounds. It makes unlocking take noticeably longer for you, and proportionally longer for anyone brute-forcing the file.
4. Since OpenSSH 7.8, August 2018, the OpenSSH format is the default for all key types, replacing the old PEM format which used a weak MD5-based derivation.
5. Server hardening, the directives that matter and what each removes:

Directive	Value	Removes
<code>PasswordAuthentication</code>	<code>no</code>	guessing attacks
<code>PermitRootLogin</code>	<code>prohibit-password</code>	direct root by password
<code>MaxAuthTries</code>	<code>3</code>	long guessing sessions
<code>AllowUsers</code>	<code>list</code>	logins by other accounts
<code>PermitEmptyPasswords</code>	<code>no</code>	the worst case
<code>LoginGraceTime</code>	<code>30</code>	held-open connections

6. Validate before restarting, every time:

```
$ sudo sshd -t
$ sudo systemctl reload ssh
```

7. fail2ban reads `/var/log/auth.log` on Debian-family systems or the journal on others. A typical jail:

```
[sshd]
enabled = true
port    = 22
maxretry = 4
findtime = 600
bantime  = 3600
```

8. That bans an address for one hour after four failures within ten minutes. With PasswordAuthentication no already set, the practical benefit is log volume rather than security, which is still a real benefit.
9. sshguard and CrowdSec are alternatives; CrowdSec additionally shares reputation data across participants.
10. Changing the port. State the honest position clearly, because both exaggerations are common.
11. It is **security through obscurity**. It provides no protection against anyone who scans your ports, which any real attacker does.
12. It does reliably remove the overwhelming majority of automated noise, because mass scanners target 22 by default. Measured on typical internet-facing hosts, failed login attempts commonly drop by more than 95 percent.
13. So the correct framing is: it is a log hygiene measure with a security side effect, and it must never be the only measure.
14. Note also that ports below 1024 require root to bind, so a high port such as 2222 is bindable by an unprivileged process, which is a small argument against picking a high port on a shared machine.
15. Key rotation, as a routine rather than an emergency:

Credential	Suggested cycle
SSH key, laptop	yearly, or on loss
SSH key, server	on rebuild
Classic PAT	90 days
Deploy key	yearly
App private key	yearly

16. **Leak response**, and this is the part to rehearse. The order matters more than the speed.
17. For an SSH key: remove the public key from every account and server that has it. GitHub's SSH key page shows a last-used date for each key, which is the fastest way to spot one you do not recognise.
18. For a token: revoke it in settings. Instant.
19. Then, in both cases, review what was done. Check the account security log, the organization audit log, and the repository's recent pushes.
20. Look specifically for the standard persistence tricks: a newly added SSH key, a new deploy key, a new webhook, a new collaborator, a changed workflow file, or a new GitHub App installation.

21. Then rotate everything the compromised credential could read. A token with repo scope could read every secret ever committed to any repository it reached, including ones you deleted from the working tree but not from history.
22. Finally, write down what happened and how the credential got out. Almost every leak is one of four things: committed to a repository, pasted into a chat or ticket, printed in CI logs, or left on a machine that was lost.

■ 42.12.6 WORDS – remember these

Passphrase — a secret that unlocks your key file — a user secret fed through a KDF to derive the key that encrypts the private key at rest.

`bcrypt_pbkdf` — the slow function guarding your key file — the KDF used by the OpenSSH private key format, with an adjustable rounds parameter.

`fail2ban` — a doorman that remembers repeat offenders — a log-scanning daemon inserting temporary firewall bans after repeated authentication failures.

Security through obscurity — hiding rather than locking — a measure that raises the cost of finding a service without raising the cost of attacking it.

Key rotation — replacing credentials on a schedule — periodic reissue and retirement, limiting the useful lifetime of an undetected compromise.

Persistence — what an attacker leaves behind — added keys, hooks, collaborators or workflow changes that survive the original credential being revoked.

42.13 A diagnosis table for git authentication failures

■ 42.13.1 PLAIN – in simple words

1. Almost every git authentication problem announces itself with one specific sentence.
2. If you learn to read those sentences, you stop guessing and start fixing.
3. There are three questions to ask in order, and they narrow it down fast.
4. First: how long did it take? Instant means somebody replied. Half a minute of silence means the network.
5. Second: did it ask for a user name or password? That means it wanted a credential and either had none or was told the one it had was wrong.
6. Third: did it mention SSH keys, or a repository name, or a scope? Each of those points at a different layer.
7. The most misleading message of all is `Repository not found`, because it usually does not mean the repository is missing.
8. It usually means your credential is not allowed to see it, and the server is refusing to confirm that it exists.

■ 42.13.2 PLAIN – a picture in your head

1. Think of a doctor listening to a description of symptoms.
2. “It hurts when I press here” narrows things down enormously, far more than “I feel unwell”.
3. Error messages are the same. `Permission denied (publickey)` is “it hurts when I press here”. `Push failed` is “I feel unwell”.
4. So the first job is always to find the most specific line in the output, not the last line.

- Git prints the summary last and the cause first. People read the last line and miss the answer three lines above it.

Where this comparison breaks:

- A patient can be asked follow-up questions. A one-shot error message cannot, which is why the verbose flags exist.

■ 42.13.3 PLAIN – a worked example

- Get more information before guessing. Two flags do almost everything.

```
# for SSH remotes, ask ssh to narrate
$ GIT_SSH_COMMAND="ssh -vvv" git push origin main

# for HTTPS remotes, ask git to narrate
$ GIT_TRACE=1 GIT_CURL_VERBOSE=1 git push origin main
```

- And the single most useful check for SSH, which tests only the connection and touches no repository:

```
$ ssh -T git@github.com
Hi USERNAME! You've successfully authenticated, but GitHub
does not provide shell access.
```

- That message is a success. The words “does not provide shell access” alarm people every time. GitHub is not a shell server; it only runs git commands.
- If that greets you by name, your key and your network are both fine, and any remaining problem is about a specific repository or a specific push.

■ 42.13.4 PLAIN – what is really happening inside

- The reason one problem produces several different messages is that several independent programs are involved, each with its own vocabulary.
- Your kernel produces `Connection refused` and `Connection timed out`. These are TCP-level outcomes and mention no names.
- ssh produces `Permission denied (publickey)` and `Host key verification failed`. These are SSH-level and mention keys.
- curl inside git produces HTTP status codes and TLS errors.
- GitHub’s own server produces every line beginning with `remote:.` Those are the only messages that reflect a decision by GitHub rather than a failure of plumbing.
- So the prefix tells you the layer. A line starting with `remote:` came from the far end and means you got there. A line starting with `ssh:` means you did not.

■ 42.13.5 TECHNICAL – the engineer’s version

- The table. Each row is an exact string you will see.

Error text	Meaning
Permission denied (publickey)	key not accepted
Invalid username or password	token wrong or gone
without workflow scope	scope missing
Host key verification failed	known_hosts mismatch
Connection timed out	packets dropped

Error text	Meaning
port 22: Connection refused	reached, nothing there
Repository not found	usually no permission
non-fast-forward	branch behind, not auth

2. Now each one in full, with the action to take.
3. **git@github.com: Permission denied (publickey)**. SSH connected and the server rejected every key offered. Causes, in order of likelihood: the public key was never added to the account; the agent is empty; the wrong key is being offered; IdentityFile points at the public key by mistake; on your own server, wrong permissions on ~/.ssh or authorized_keys. Check with `ssh -T git@github.com` and `ssh-add -l`. On a server you control, read the server log, which is the only place bad ownership or modes appears.
4. **remote: Invalid username or password.** or **remote: Support for password authentication was removed on August 13, 2021.** The HTTPS credential is not valid. Either it is an account password, which has not worked since that date, or the token expired, was revoked, or was for a different account. Clear the cached credential with `git credential reject` or the platform keychain command, then retry so you are prompted afresh.
5. **! [remote rejected] main -> main (refusing to allow a Personal Access Token to create or update workflow ... without workflow scope)** The credential is valid. The push touched .github/workflows/. Add the workflow scope to the token, or the Workflows write permission to a fine-grained token, then push again. Nothing was written; the branch is unchanged.
6. **Host key verification failed.** The server's key does not match the saved one, or you declined the prompt. Compare the printed fingerprint against the operator's published one before doing anything else. If and only if it matches, remove the stale entry with `ssh-keygen -R github.com` and reconnect. Never delete the file wholesale to make a warning go away.
7. **ssh: connect to host github.com port 22: Connection timed out** Nothing replied. Port 22 is filtered on this network, or the path is broken. Test 443 as a control. Fix with an HTTPS remote or the `ssh.github.com` port 443 configuration in 42.7. Expect roughly 15 to 130 seconds before this message appears, depending on the system.
8. **ssh: connect to host X port 22: Connection refused** The opposite of the above and far friendlier. Something answered and said no. The host is reachable and nothing is listening on 22. Either sshd is not running, or it listens on another port. Measured in milliseconds, not seconds.
9. **remote: Repository not found.** followed by **fatal: repository ... not found**. Read this as "not found for you". Causes: a typo in the owner or repository name; the repository is private and your credential lacks access; you are authenticated as a different account than you think; a classic token without repo scope, which cannot see private repositories at all; an organization with SAML single sign-on where the token has not been authorized for that organization.
10. Why 404 rather than 403? Because returning 403 would confirm the repository exists. GitHub deliberately returns "not found" to anyone not permitted to know. We measured this directly in 42.8: an anonymous request for a nonexistent repository returned 401, not 404.
11. **! [rejected] main -> main (non-fast-forward) or (fetch first)**. Not an authentication problem at all. Your branch is behind the remote. `git fetch` then rebase or merge. If you push after a rebase, use `--force-with-lease`, never plain `--force`.
12. **send-pack: unexpected disconnect while reading sideband packet** The connection died while git was reading the server's response. This is the error the reader saw twice during a network outage.

13. It deserves a paragraph of its own because it is the most dangerous message in this list, and for a reason that is not obvious.
14. It tells you the response was lost. It tells you nothing about whether the write landed. The server may have applied your push completely and then failed to tell you.
15. So never assume either outcome. Re-query the actual state: `git ls-remote origin refs/heads/main` and compare the hash against your local commit.
16. Do not trust `origin/main` in your local repository either. That is a remote-tracking reference, which is a cached copy of what the remote said last time you successfully spoke to it. During an outage it is stale by definition.
17. Two more you will meet:
18. **Too many authentication failures** The agent offered more keys than the server's `MaxAuthTries` allows before you got to the right one. Fix with `IdentitiesOnly yes` and an explicit `IdentityFile` for that host.
19. **sign_and_send_pubkey: signing failed: agent refused operation** The agent has the public key but cannot use the private key. Usually a key added from a file that has since changed, or a hardware key that needs a touch. Run `ssh-add -D` then re-add.
20. A general ordering rule that resolves most confusion: read the output from the top, find the first line that names a cause, and ignore the summary at the bottom. `error: failed to push some refs` is never the reason.

■ 42.13.6 WORDS - remember these

`remote:` prefix — a line that came from the server — output relayed from the far end, proving the connection and authentication both worked.

Sideband — the side channel carrying server messages — the multiplexed stream in the pack protocol used for progress and error text.

Remote-tracking reference — your cached note of what the server said — `refs/remotes/origin/main`, updated only on a successful fetch or push.

`--force-with-lease` — force push, but safely — a push that refuses if the remote moved since your last fetch, unlike plain `--force`.

`ls-remote` — ask the server what it actually has — a command querying live references without touching your working tree.

SAML SSO authorization — an extra approval step for org access — an organization requirement that a token be separately authorized before it can reach that organization's repositories.

42.98 Common wrong ideas

1. Wrong: SSH and HTTPS reach different copies of the repository, so switching loses work. Right: it is one repository. Only the road changes. Switching is one `git remote set-url` and nothing is lost.
2. Wrong: the private key is sent to the server, so publishing the public key is risky too. Right: the private key never moves. The server sends data, your machine signs it, and only the signature crosses the network. The public key is designed to be pasted into web forms on purpose.
3. Wrong: `Permission denied (publickey)` means the key is wrong. Right: it also appears when `~/.ssh` or `authorized_keys` has loose permissions, when the agent is empty, and when too many wrong keys were offered first. Only the server log distinguishes them.

4. Wrong: the host key warning is a nuisance, so delete `known_hosts`, and anyway a change always means an attack. Right: both halves are wrong. It is the one warning that detects an interceptor, so compare the fingerprint against a published one and remove only the stale entry. And GitHub legitimately rotated its RSA host key at about 05:00 UTC on 24 March 2023.
5. Wrong: port 22 being blocked means GitHub is down, and a blocked port fails immediately. Right: on the machine used for this chapter, port 22 timed out after 30.057 seconds while port 443 connected in 0.021 seconds to the same host. A DROP sends nothing back, so your machine retries with doubling gaps and hangs. Only a REJECT fails fast, with `Connection refused`.
6. Wrong: a token is just a password with extra steps. Right: a token can be scoped to specific actions, expire on a date, and be revoked alone. A password can do none of those things, which is why GitHub stopped accepting passwords for git on 13 August 2021.
7. Wrong: the repo scope includes everything, so a push cannot be refused for permissions. Right: repo does not include workflow. A push touching `.github/workflows/` is refused without it, even though authentication succeeded and every object transferred.
8. Wrong: `remote rejected` means the network dropped the push. Right: the word `remote:` proves you reached the server and it answered in sentences. That is a policy decision, not a transport failure.
9. Wrong: `Repository not found` means the repository does not exist. Right: it usually means your credential may not see it. GitHub returns “not found” rather than “forbidden” so it does not confirm that a private repository exists.
10. Wrong: changing the SSH port secures a server, and agent forwarding is just a convenience toggle. Right: a different port only removes automated scanning noise, since a real attacker scans all ports. And root on the machine you forwarded to can use your agent silently, against every server your keys open, for as long as you stay connected. Prefer ProxyJump.

42.99 Chapter summary in 20 lines

1. Git borrows two transports: SSH on TCP 22 and HTTPS on TCP 443. The repository is identical either way; only the road and the credential differ.
2. Over SSH, git logs in as the user `git` and runs one command remotely, `git-upload-pack` or `git-receive-pack`. Over HTTPS it makes one GET and one POST.
3. SSH was written by Tatu Ylonen at Helsinki University of Technology in 1995 after a password-sniffing attack, replacing telnet, rlogin and rsh, which sent passwords in clear text.
4. SSH-2 replaced the structurally flawed SSH-1 and is specified in RFC 4251 to RFC 4254. OpenSSH forked from Bjorn Gronvall’s OSSSH on 26 September 1999 and shipped with OpenBSD 2.6 on 1 December 1999.
5. SSH-2 has three layers: transport for encryption and host key verification, user authentication, and connection for multiplexed channels.
6. A key pair splits into a private key that signs and a public key that only verifies. Publishing the public half is safe; the private half never leaves your machine.
7. Authentication is challenge-response: the client signs a block containing the session identifier, so an old signature cannot be replayed and a man-in-the-middle cannot forward yours.
8. Ed25519 is the current recommendation: deterministic signatures, 32-byte public keys, published parameter rationale, RFC 8709. RSA 2048 is the floor and 4096 is common. DSA is removed everywhere.
9. `authorized_keys` is a plain text file of public keys. Wrong permissions make the server ignore it and return `Permission denied (publickey)` with no explanation to the client.

10. Options in front of a key restrict it: `command=`, `from=`, `no-port-forwarding`, and `restrict` for everything at once. Deploy keys are the per-repository variant.
11. `known_hosts` is the other half nobody reads. The server proves itself first, and the first connection is trust on first use, which is the model's honest weakness.
12. GitHub rotated its RSA host key at about 05:00 UTC on 24 March 2023 after the private key was briefly exposed publicly. That is what a legitimate change looks like.
13. The agent holds unlocked keys and signs on request, so the key never leaves its memory. Agent forwarding lets root on the remote host use your keys and should be replaced by ProxyJump.
14. Port 22 is commonly filtered outbound because it is heavily scanned and useful for exfiltration, while 443 must stay open or the web breaks. A DROP hangs for tens of seconds; a REJECT refuses in milliseconds.
15. The two fixes are switching the remote to HTTPS, or using GitHub's SSH endpoint `ssh.github.com` on port 443 with a four-line `~/.ssh/config` block.
16. GitHub removed password authentication for git operations on 13 August 2021. Credential helpers now store tokens, in the OS keychain if you are careful and in plain text if you use `store`.
17. A token can be scoped, can expire and can be revoked alone. Its prefix, `ghp_` or `github_pat_`, exists so that secret scanning can find it, and push protection blocks it before it ever leaks.
18. A scope is a named permission checked at the moment of the operation. Pushing a file under `.github/workflows/` needs `workflow`, because changing CI changes what runs on GitHub's runners with the repository's secrets.
19. Distinguish the three "push failed" cases by evidence: silence and a long wait is network, a credential prompt or `Invalid username is authentication`, and `! [remote rejected]` with a sentence is authorization.
20. Four identities with four blast radii: SSH keys, personal access tokens, OAuth grants and GitHub Apps. CI should hold no long-lived secret at all and use short-lived OIDC-federated credentials instead.

Cloud Computing, the Bezos Mandate, AWS and Azure

43.0 What this chapter gives you

1. You will be able to say what “the cloud” actually is in one plain sentence, without any mystery, and say exactly what it changed and what it did not.
2. You will be able to describe a real datacentre from the outside wall to the individual server, including power, cooling and what a rack unit is.
3. You will be able to explain regions, availability zones and edge locations precisely, and name the real regions the big three run inside India.
4. You will be able to explain what a hypervisor does, why virtual machines became cheap, and how special hardware removed the last of the overhead.
5. You will be able to state clearly that a container is a process, not a small virtual machine, and name the exact Linux features that make it work.
6. You will be able to read a Dockerfile, a Kubernetes manifest and a Terraform file line by line, and say what each line causes to happen.
7. You will be able to quote the 2002 Bezos API mandate, explain why it forced Amazon to build a platform, and say honestly what it cost.
8. You will be able to tell the true story of how AWS started, and correct the popular myth about spare retail capacity with evidence.
9. You will be able to name the eighteen AWS services you meet in real work, say what problem each solves, and map each to Azure and Google Cloud.
10. You will be able to read a cloud bill, predict where it will explode, and argue both sides of the buy-versus-rent question with real numbers.

43.1 What the cloud actually is

■ 43.1.1 PLAIN – in simple words

1. The cloud is somebody else’s computers, sitting in somebody else’s building.
2. You rent them by the hour or by the second, instead of buying them.
3. You reach them over the internet, and you never touch them physically.
4. That is the whole idea. There is nothing floating in the sky.
5. The word “cloud” came from old network drawings, where people drew a fuzzy cloud shape to mean “the internet, and we are not going to draw the details”.
6. So “in the cloud” originally just meant “over there, past our own wires”.
7. Four things genuinely changed when this became a business you could buy.

8. First, **elasticity**: you can ask for 200 machines at 9am and give them back at 11am, and pay only for those two hours.
9. Second, **pay per use**: no big purchase up front, no machine sitting idle.
10. Third, **self-service**: you get the machines by making a request from a program, in seconds, with nobody's approval and no phone call.
11. Fourth, somebody else deals with the building, the power, the cooling, the broken disks and the security guard at the door.
12. Here is what did not change. These are still real physical computers.
13. They still overheat, lose power, lose disks and lose network cables.
14. Your program can still be slow, still be wrong, and still fall over.
15. The cloud moved the pain. It did not delete the pain.

■ 43.1.2 PLAIN - a picture in your head

1. Think about how you get electricity in your home.
2. You do not own a generator. You do not buy fuel or service a machine.
3. You plug in, you use what you need, and once a month a meter reading turns into a bill.
4. If you buy an air conditioner, you do not have to also buy a bigger generator. You just use more, and the bill goes up.
5. If you go away for a month, you use almost nothing, and you pay almost nothing.
6. Somebody else runs enormous power stations so that you can avoid running a small one badly.
7. Renting computers works the same way. Amazon, Microsoft and Google run the enormous machine rooms so that you can avoid running a small one badly.
8. This comparison is not new. The writer Nicholas Carr made it the centre of his 2008 book *The Big Switch*, comparing cloud computing with the way factories stopped running their own power plants a century earlier.

Where this comparison breaks: electricity is one product. One volt is like any other volt, and you can change supplier without rewiring your house. Cloud services are all different from each other, and moving from one supplier to another means rewriting real code. Also, electricity is metered on one dimension. A cloud bill meters compute time, storage, requests, data movement, addresses and dozens of other things at once, and the surprise is almost always in a dimension you were not watching.

■ 43.1.3 PLAIN - a worked example

1. You are two students launching a website for a college festival.
2. Old way: you buy one server. Say it costs 90,000 rupees, plus a rack space rental, plus a person to look after it.
3. You must guess the busiest moment months in advance and buy for that.
4. On the festival night 40,000 people arrive at once. Your one server dies.
5. For the other 360 days of the year, the server is 3 per cent busy, and you have paid for all of it anyway.
6. Cloud way: you rent two small machines for the quiet months. On festival night you rent forty machines for six hours, then give them back.

Approach	Quiet month	Festival night
Own one server	Paid in full	Site falls over
Rent, fixed size	Pay 2 machines	Site falls over

Approach	Quiet month	Festival night
Rent, elastic	Pay 2 machines	Pay 40 for 6 hours

7. Real prices, from the AWS public price list for the North Virginia region in August 2026: a small t3.medium machine costs 0.0416 US dollars per hour.
8. Forty of them for six hours is $40 \times 6 \times 0.0416 = 9.98$ US dollars.
9. That is the actual thing that changed. Ten dollars, decided at 7pm, with no purchase order and no meeting.

■ 43.1.4 PLAIN – what is really happening inside

1. You send a small message over the internet to the provider’s control system. It says, in effect, “give me one machine of this size, in this place”.
2. That message is a normal web request, signed with your secret key so the provider knows it is really you.
3. A piece of software called the **control plane** takes the request. Its job is to decide and to record, not to run your program.
4. It finds a physical server in the right building that has room, and picks it.
5. On that physical server, a program called a **hypervisor** carves out a slice: some processor time, some memory, some disk, one network address.
6. It boots an operating system into that slice from a saved disk image.
7. Within about 30 to 60 seconds, that slice answers to a network address, and you can log in as if it were a computer on your desk.
8. Everything after that is ordinary computing. The slice does not know or care that it is a slice.
9. A separate meter records that this slice existed, second by second, and that record eventually becomes a line on your bill.
10. When you say “delete it”, the hypervisor throws the slice away and the physical server is offered to the next customer.

■ 43.1.5 TECHNICAL – the engineer’s version

1. The working definition of cloud computing is not marketing. It is a standard document: NIST Special Publication 800-145, published September 2011 by the United States National Institute of Standards and Technology.
2. It names five essential characteristics. A service is only cloud computing if it has all five.
3. On-demand self-service: the consumer provisions capacity unilaterally, with no human interaction with the provider.
4. Broad network access: capabilities are available over the network through standard mechanisms.
5. Resource pooling: physical and virtual resources are pooled and assigned to many consumers, with location independence at some level of abstraction.
6. Rapid elasticity: capabilities can be scaled outward and inward, in some cases automatically, and appear unlimited to the consumer.
7. Measured service: use is metered and reported, giving transparency to both sides.
8. The same document names three service models, IaaS, PaaS and SaaS, and four deployment models: private, community, public and hybrid.
9. That test is genuinely useful. A rented server with a two-week ticket queue fails point three and point six, and is therefore hosting, not cloud.
10. Scale of the industry, from published company filings. Amazon reported AWS net sales of 128,725 million US dollars for the full year 2025, with segment operating income of 45,606 million.

11. In the quarter ending June 2026 AWS reported 42.2 billion US dollars, up 37 per cent year on year, an annualized run rate near 169 billion.
12. Microsoft first disclosed Azure's own revenue for its fiscal year 2025, at 75 billion US dollars, and reported that Azure passed 100 billion for fiscal 2026, growing 41 per cent.
13. Alphabet reported Google Cloud revenue of 17.664 billion US dollars for the quarter ending December 2025, up 48 per cent, and said the business ended 2025 at an annual run rate above 70 billion.

Provider	Latest disclosed figure	Period
AWS	128.7 bn USD	Full year 2025
Azure	Over 100 bn USD	Fiscal year 2026
Google Cloud	17.7 bn USD	Q4 2025 quarter

14. Commands that make the abstraction visible: `aws ec2 describe-instances`, `az vm list`, `gcloud compute instances list`. Each is a signed web request to a control plane, nothing more.

■ 43.1.6 WORDS – remember these

1. Cloud — renting computers you reach over the internet — on-demand, self-service, metered, pooled and elastic computing per NIST SP 800-145.
2. Elasticity — getting more and giving it back quickly — automatic or manual scaling of provisioned capacity in both directions.
3. Control plane — the part that decides and records — the API and orchestration layer that provisions and tracks resources.
4. Data plane — the part that does the actual work — the running instances, packets and stored bytes themselves.
5. Provisioning — asking for and receiving a resource — the allocation of capacity to a tenant in response to an API call.
6. Tenant — one paying customer sharing the hardware — an isolated account whose workloads share pooled physical resources.

43.2 Inside a real datacentre

■ 43.2.1 PLAIN – in simple words

1. A datacentre is a warehouse full of computers, built so they never stop.
2. Inside, computers are not on desks. They are bolted into tall metal frames called **racks**, like books on a shelf but lying flat.
3. Each computer is a flat metal tray about as wide as a shelf and as thick as a few textbooks. Rows of them fill each rack from floor to head height.
4. Every rack has a small switch at the top that all its computers plug into.
5. Those switches plug into bigger switches, which plug into the internet.
6. Power comes in from the electricity grid, on two separate paths, so that one failure does not stop the room.
7. Between the grid and the computers sit big batteries, which cover the few seconds of a cut, and diesel generators, which cover the hours.
8. All that electricity turns into heat, so a large part of the building is machinery for taking heat away.
9. Cold air is pushed up through the floor into one aisle, sucked through the computers, and comes out hot into the next aisle.

10. Getting in is hard on purpose: fences, cameras, guards, card and fingerprint doors, and a locked cage around your own rack.
11. These buildings are enormous. A big one holds tens of thousands of computers and uses as much electricity as a small city.

■ 43.2.2 PLAIN – a picture in your head

1. Think of a very large library.
2. The racks are the shelves, standing in long parallel rows.
3. Each server is one book lying flat on the shelf, and the shelf spacing is fixed so books of standard thickness always fit.
4. Every row of shelves has a librarian at the end of it, the top-of-rack switch, who handles all messages in and out of that row.
5. The librarians all report to a small group of head librarians, and any librarian can reach any other through any head librarian.
6. The reading rooms alternate: one cold room, one warm room, one cold, one warm, because readers give off heat and you keep the cold air away from the warm exhaust.
7. The building has its own power station in the yard, in case the city cuts supply, and a battery room that keeps the lights on while it starts.

Where this comparison breaks: a library's books are the point of the library. Here, the servers are interchangeable and expected to die. Roughly two to five per cent of servers fail in a year, and staff replace them without anybody noticing. Also, a library is quiet. A datacentre hall is loud enough that long-stay staff wear ear protection, because thousands of fans are running.

■ 43.2.3 PLAIN – a worked example

1. Take one standard rack. The width of the mounting rails is 19 inches, which is 482.6 millimetres. This is the EIA-310 standard, and it dates from telephone equipment in the 1920s.
2. Height is measured in **rack units**, written U. One U is 1.75 inches, which is 44.45 millimetres, exactly.
3. A common rack is 42U tall, about 1.87 metres of usable height.
4. A basic web server is 1U thick. So 42 of them fit, minus space for switches and cable management. In practice you plan for about 38 servers plus 2 switches.
5. Now the power. Suppose each server draws 350 watts on average.
6. $38 \times 350 \text{ W} = 13,300 \text{ W}$, which is 13.3 kilowatts for one rack.
7. An older datacentre was designed for 5 to 10 kilowatts per rack, so that rack would not fit its power budget. You would fill the rack only half full.
8. A modern rack of graphics processors for artificial intelligence work is a different animal. An NVIDIA GB200 NVL72 rack draws up to about 120 kilowatts, which is roughly what forty Indian homes use at peak.
9. Air cannot carry that much heat away. Those racks are water-cooled, with cold liquid piped directly onto the chips.

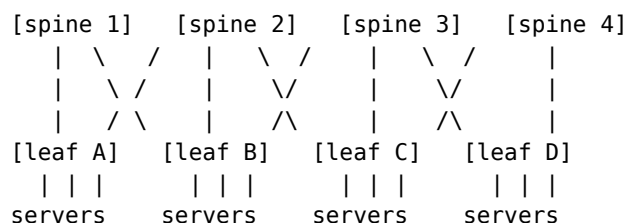
Rack type	Power per rack	Cooling
Legacy enterprise	5 to 10 kW	Air
Modern cloud	10 to 20 kW	Air, contained
AI training rack	Up to 120 kW	Direct liquid

■ 43.2.4 PLAIN – what is really happening inside

1. Grid electricity arrives at high voltage and is stepped down by transformers outside the building.
2. It passes through switchgear that can move the whole load from one source to another in a fraction of a second.
3. It then reaches the uninterruptible power supply, the UPS. This is a bank of batteries or a spinning flywheel.
4. The UPS is not there to run the building for hours. It is there to hold the load for the 10 to 30 seconds a diesel generator needs to start and take over.
5. Generators keep enough fuel on site for a day or more, with contracts for refuelling after that.
6. From the UPS, power goes to rack-level power strips, called PDUs. Servers have two power supplies, plugged into two independent strips, fed from two independent paths.
7. Cooling runs in a loop. Computer room air handlers blow cold air under a raised floor or through overhead ducts into the cold aisle.
8. Server fans pull that air front to back, over the hot parts, and blow it into the hot aisle behind.
9. Physical barriers, doors and roofs, seal the aisles so cold and hot air never mix. Mixing is the single biggest waste in a machine room.
10. The hot air returns to the coolers, gives its heat to a water loop, and the water carries the heat to chillers or to outdoor cooling towers.
11. For very hot racks, air is skipped entirely. A cold plate sits directly on the chip and water takes the heat out of the rack.
12. Meanwhile the network: each server has one or two cables to the switch at the top of its own rack, and those switches connect upwards to a layer of larger switches, so that any server can reach any other in a small, fixed number of hops.

■ 43.2.5 TECHNICAL – the engineer's version

1. Rack mechanics follow EIA-310-D. Rail spacing 19 inches, 1U equals 44.45 mm, common heights 42U, 45U, 48U and 52U, common depths 1000 to 1200 mm.
2. Rackmount servers are 1U, 2U or 4U. A 1U dual-socket server today typically holds 2 CPUs of 32 to 128 cores each, 12 to 32 memory slots and a handful of NVMe drives.
3. Blade servers put many thinner server modules into a shared chassis, sharing power supplies, fans and network fabric. They win on density and cabling and lose on vendor lock-in. Hyperscale operators mostly use custom rackmount designs instead, following the Open Compute Project, which Facebook founded in 2011.
4. Network topology is **spine-leaf**, also called a folded Clos network after Charles Clos's 1953 telephone switching paper. Every leaf connects to every spine, and no leaf connects to another leaf.



Any server to any server: leaf -> spine -> leaf. Two hops.
Add a spine to add bandwidth. Add a leaf to add racks.

5. Leaf switches are the top-of-rack switches, typically 48 ports of 25 or 100 gigabits per second down to servers, and 4 to 8 uplinks of 100 to 400 gigabits per second up to the spine.

6. The ratio of downlink to uplink bandwidth is the **oversubscription ratio**. 3:1 is common in general fleets. Storage and AI clusters are built at 1:1, meaning non-blocking.
7. Power redundancy is described as N, N+1 or 2N. The Uptime Institute Tier classification runs Tier I to Tier IV, where Tier III allows concurrent maintenance and Tier IV is fault tolerant.
8. ASHRAE Thermal Guidelines class A1 recommends an inlet temperature of 18 to 27 degrees Celsius. Operators run near the top of that band deliberately, because warmer air needs less chilling.
9. **Power usage effectiveness**, PUE, is total facility power divided by IT equipment power. A PUE of 2.0 means one watt of overhead for every watt of computing. The theoretical floor is 1.0.

Operator or group	PUE	Source year
Global average	1.54	Uptime 2025 survey
Hyperscale range	1.10 to 1.15	Uptime 2025 survey
Google fleet, TTM	1.09	Reported Q1 2026
Colocation and enterprise	1.58 to 1.80	Uptime 2025 survey

10. Google's published history shows fleet PUE falling from about 1.23 in 2008 to 1.09 today. The global average has been stuck near 1.54 since 2019, which is an honest and unflattering industry fact.
11. Physical security is audited, not asserted. The relevant reports are SOC 2 Type II, ISO/IEC 27001 and, for cardholder data, PCI DSS. Controls include perimeter fencing, vehicle barriers, two-factor mantrap entry, biometric readers, per-page access lists and asset destruction on decommission.
12. Scale and cost, from 2026 construction cost surveys: roughly 11.3 million US dollars per megawatt of IT capacity for a standard facility, and 20 million or more per megawatt for an AI-optimized build.
13. On that basis a 500 megawatt campus costs about 5.5 to 6 billion US dollars standard, or over 10 billion for a dense AI build. Electrical systems alone are 40 to 50 per cent of construction cost.
14. Microsoft stated that it deployed about 2 gigawatts of new datacentre capacity during its fiscal year 2025. That single year of one company is comparable to the entire generating capacity of a large power station.

■ 43.2.6 WORDS - remember these

1. Rack unit (U) — one shelf slot — 44.45 mm of vertical mounting height under EIA-310-D.
2. Top-of-rack switch — the small switch every server in a rack plugs into — the leaf layer of a Clos fabric, typically 48 x 25G with 100G uplinks.
3. Spine-leaf — a wiring plan where any machine is two hops from any other — a folded Clos topology with full leaf-to-spine meshing.
4. UPS — the battery that covers a power cut — an uninterruptible power supply sized to bridge to generator start, typically 10 to 30 seconds.
5. Hot and cold aisle — keeping cold air and hot exhaust apart — containment of supply and return airflow to prevent recirculation.
6. PUE — how much extra power the building wastes — total facility power divided by IT load, global average 1.54, hyperscale 1.10 to 1.15.
7. CRAC and CRAH — the room's air conditioners — computer room air conditioner with its own compressor, versus air handler fed by chilled water.

43.3 Regions, availability zones and edge locations

■ 43.3.1 PLAIN – in simple words

1. A **region** is a place on the map where a provider has built capacity. Mumbai is a region. Frankfurt is a region. Ohio is a region.
2. Regions are far apart, and they are run as almost separate businesses. Prices differ, and some services exist in one region and not another.
3. Inside a region there are **availability zones**. A zone is one or more buildings with its own power, its own cooling and its own network links.
4. Two zones in one region are close enough that talking between them is fast, but far enough apart that one flood, fire or substation failure cannot take out both.
5. That is the whole point. A zone is a **failure domain**: a boundary that damage is not supposed to cross.
6. So you never put your only copy of anything in one zone. You run at least two copies, in two zones, and something in front that sends traffic to whichever is alive.
7. An **edge location** is a third thing and much smaller. It is a small set of machines in many cities that hold copies of files and terminate connections close to users.
8. Edge locations make things feel fast. They do not run your application. Do not confuse them with regions.
9. Rule of thumb: zones protect you from a building. Regions protect you from a city or a country. Edge locations protect you from distance.

■ 43.3.2 PLAIN – a picture in your head

1. Think of a bank with branches.
2. A region is a city where the bank operates.
3. Availability zones are three separate branch buildings in that city, on different streets, on different electrical supplies.
4. If one branch burns down, your money is still in the other two, because the bank writes every transaction to all three before confirming it.
5. An edge location is an ATM. There are hundreds of them, in many towns, and they are fast to reach.
6. But an ATM cannot open an account or approve a loan. It holds cash and handles simple requests. For anything real it talks back to a branch.
7. Countries also have rules: some money must stay inside the country. That is data residency.

Where this comparison breaks: bank branches in a city are genuinely independent businesses that can run alone. Cloud availability zones share a regional control plane, and some regional services have components that live in only one zone. In the AWS event of 20 October 2025, one subsystem in one region failed and the effect was felt across the whole region and by customers worldwide, even though the individual zones had power and cooling the entire time.

■ 43.3.3 PLAIN – a worked example

1. You run a shop website for Indian customers. You pick the AWS Mumbai region, whose code name is `ap-south-1`.
2. That region has three availability zones, named `ap-south-1a`, `ap-south-1b` and `ap-south-1c`.
3. You place two web servers in zone a and two in zone b, with a load balancer in front that spreads traffic across both.
4. You place your database as a primary in zone a and a standby copy in zone b, with automatic promotion if the primary is lost.
5. Now zone a loses power. The load balancer stops sending traffic to the two dead servers within about 30 seconds. The database standby is promoted in one to two minutes. Customers see a short blip.
6. Cost of that safety: you run four servers where two would do, and you pay for traffic that crosses between zones, at 0.01 US dollars per gigabyte in each direction.
7. Now suppose the whole Mumbai region fails. Nothing you have deployed helps. Surviving that needs a second region, which needs a second copy of your data kept up to date, which is a much larger project.

■ 43.3.4 PLAIN – what is really happening inside

1. Zone names are shuffled per account on purpose. Your `ap-south-1a` and my `ap-south-1a` are usually different physical zones, so that customers do not all crowd into the alphabetically first one.
2. The stable identifier is a zone ID such as `aps1-az1`, which is the same for everyone. That is what you compare when you need to know two accounts are truly in the same building.
3. Between zones, the provider runs its own dedicated fibre, usually many redundant paths. Round trip time is typically well under two milliseconds.
4. Between regions, traffic crosses the provider's private long-distance backbone. Mumbai to Singapore is roughly 60 to 70 milliseconds round trip. Mumbai to North Virginia is roughly 190 to 220 milliseconds.
5. Those numbers come from the speed of light in glass and cannot be improved by any amount of money. Light travels about 200,000 kilometres per second in fibre, so 1,000 km of fibre costs about 5 milliseconds one way, before any switching delay.
6. A region failure, in practice, is rarely the building burning. It is far more often a software or configuration fault in a service that everything else in the region depends on, such as identity, DNS or the metadata store.
7. When that happens, machines that are already running usually keep running, but you cannot start new ones, cannot log in, and cannot change anything.

■ 43.3.5 TECHNICAL – the engineer’s version

1. Current published footprints, checked in August 2026. AWS states 123 availability zones across 39 geographic regions, with plans announced for the Kingdom of Saudi Arabia and Chile. Microsoft states over 70 Azure regions. Google states 43 regions and 130 zones.
2. AWS defines an availability zone as one or more discrete datacentres with redundant power, networking and connectivity, physically separated by a meaningful distance, up to 100 km apart, with single-digit millisecond latency between them.
3. Azure calls the equivalent an availability zone too, and pairs most regions into region pairs for platform-managed geo-replication. You cannot choose your pair.
4. Google Cloud calls them zones, and a zone is a deployment area within a region, isolated for failure purposes.
5. The India footprint, with names as the providers publish them:

Provider	Region name	Live from
AWS	Mumbai, ap-south-1	June 2016
AWS	Hyderabad, ap-south-2	21 Nov 2022
Azure	Central, South, West India	Sept 2015
Azure	India South Central	6 Aug 2026
Google	Mumbai, asia-south1	Nov 2017
Google	Delhi NCR, asia-south2	15 July 2021

6. Azure also offers Jio India West and Jio India Central, operated in partnership with Reliance Jio. Azure’s three 2015 regions are in Pune, Chennai and Mumbai; the 2026 India South Central region is in Hyderabad and opened with three availability zones.
7. AWS announced an investment of 4.4 billion US dollars in India by 2030 at the Hyderabad launch. Microsoft’s stated India commitment reached 20.5 billion US dollars by 2026, combining announcements from January 2025 and December 2025.
8. Data residency is a legal question, not a technical one. In India the relevant instruments are the Digital Personal Data Protection Act, 2023, and the Reserve Bank of India’s April 2018 directive requiring payment system data to be stored in India.
9. In the European Union it is the General Data Protection Regulation of 2016, applied from May 2018, plus sector rules and sovereign cloud offerings.
10. Choosing a region therefore has three inputs: latency to your users, price, and law. They frequently disagree.
11. The honest note on multi-region. Marketing implies you tick a box. The reality is that you must solve write conflicts, replication lag, failover orchestration, DNS behaviour under partial failure, cost of duplicated capacity, and testing.
12. Almost nobody tests region failover under load. A failover path that has never been exercised is a hypothesis, not a capability.
13. Real example to anchor this. On 20 October 2025 AWS suffered a major disruption in us-east-1, caused by a latent race condition in the automated DNS management of DynamoDB. The trigger occurred around 06:49 UTC, DNS was restored by 09:25 UTC, and full recovery of dependent systems such as EC2 instance launches continued until about 20:50 UTC the same day, with some Redshift work finishing the next morning.
14. Many companies that believed they were multi-region discovered that their identity provider, their build system or their DNS was single-region.

15. Commands to inspect this: `aws ec2 describe-availability-zones --region ap-south-1` shows both the name and the stable zone ID.

■ 43.3.6 WORDS – remember these

1. Region — a city-sized place where the provider has capacity — an independent deployment area with its own service catalogue, pricing and endpoints.
2. Availability zone — one building group with its own power — an isolated failure domain within a region, single-digit millisecond latency to peers.
3. Failure domain — the boundary damage should not cross — the blast radius of a single correlated fault.
4. Edge location — a small cache near users — a point of presence for content delivery and connection termination, not general compute.
5. Data residency — the rule that data must stay in a country — a legal obligation on storage location, distinct from encryption or access control.
6. Region pair — Azure's fixed partner region — a Microsoft-assigned pairing used for platform geo-replication and staged updates.

43.4 Virtualization, the technology that made cloud possible

■ 43.4.1 PLAIN – in simple words

1. One physical computer used to run one operating system, and that was that.
2. Most of the time it was mostly idle, waiting for something to do.
3. Virtualization is a trick that lets one physical computer pretend to be many separate computers at once.
4. The program that does the pretending is called a **hypervisor**.
5. It sits between the real hardware and the pretend computers, and it hands out slices of processor, memory, disk and network.
6. Each pretend computer is called a **virtual machine**. It boots its own operating system and believes it is alone.
7. It is not alone. Ten or forty of its neighbours are on the same metal, completely unaware of each other.
8. This is what made renting computers by the hour possible. Without it, giving a customer a machine would mean physically plugging one in.
9. With it, giving a customer a machine means writing a few numbers into a table and starting a process.
10. It also lets the provider move a running customer off a server that needs repair, without switching that customer off. That is called live migration.

■ 43.4.2 PLAIN – a picture in your head

1. Think of a large old house converted into flats.
2. The house is the physical server. The flats are the virtual machines.
3. Each family has its own front door, own kitchen, own furniture, and never sees the others.
4. There is one water main, one electricity supply and one roof, shared. A caretaker, the hypervisor, splits them fairly and stops one family from flooding another.
5. If one family plays loud music, the others may hear it a little. That is resource contention, the noisy neighbour problem.
6. If the caretaker needs to fix the roof over one flat, a very good caretaker can move the whole family, mid-dinner, to an identical flat next door, so fast they barely notice. That is live migration.

Where this comparison breaks: a family in a flat knows it is in a flat. A well built virtual machine genuinely cannot tell, by any ordinary means, that it is virtual. That indistinguishability is a formal design goal, stated by Gerald Popek and Robert Goldberg in their 1974 paper, and it is stronger than any housing analogy suggests. Also, the caretaker in this story has absolute power: the hypervisor can read every byte of every flat, which is why the trust model of the provider matters so much.

■ 43.4.3 PLAIN – a worked example

1. Take a physical server with 2 processors of 64 cores each, so 128 physical cores, plus 1,024 gigabytes of memory.
2. With hyper-threading on, the operating system sees 256 logical processors.
3. You want to sell virtual machines with 4 virtual processors and 16 gigabytes of memory each.
4. On memory alone, 1,024 divided by 16 gives 64 virtual machines, minus about 5 per cent kept for the hypervisor itself. Call it 60.
5. On processors, 60 virtual machines x 4 virtual processors = 240 virtual processors against 256 logical ones. That is roughly 1:1, no oversubscription.
6. If instead you sell to customers whose machines are usually idle, you might place 120 virtual machines, giving 480 virtual processors on 256 logical ones, an oversubscription ratio of nearly 2:1.
7. That is a business decision, not a technical one, and it is exactly why cheap virtual servers feel inconsistent and expensive ones do not.

Setting	Ratio	Effect on tenant
No oversubscription	1:1	Steady, predictable
Light, 2:1	2:1	Usually fine, sometimes slow
Aggressive, 8:1	8:1	Frequent stealing of time

8. You can measure this from inside a Linux guest. Run `top` and look at the `st` column, short for steal time. It is the percentage of time your virtual processor was ready to run but the hypervisor gave the physical core to somebody else. Anything consistently above 5 is a crowded host.

■ 43.4.4 PLAIN – what is really happening inside

1. A processor has at least two privilege levels: a privileged one for the operating system and an unprivileged one for applications.
2. Normally the operating system sits in the privileged level and controls the hardware directly.
3. Under virtualization, the hypervisor takes the privileged level, and the guest operating system is pushed down a level, even though it still thinks it is privileged.
4. When the guest executes an instruction that only a truly privileged program may run, the processor stops and hands control to the hypervisor. This is called a **trap**.
5. The hypervisor looks at what the guest was trying to do, does something safe and equivalent, and resumes the guest.
6. Memory is handled with two levels of translation. The guest maps its own pages as usual; the hardware then maps those again to real physical pages, using a second table the hypervisor controls.
7. Disks are usually just large files on the host, or slices of a network storage system. A write inside the guest becomes a write to that file.
8. Network cards are emulated or, better, partly real: modern cards can present many independent copies of themselves, and the hypervisor gives one copy directly to each guest.

9. Live migration works by copying memory pages to the destination host while the guest keeps running, repeatedly re-copying pages that changed, until the set of changed pages is small enough to copy in a brief pause.
10. That final pause is typically well under a second, and the guest's network connections survive because the address moves with it.

■ 43.4.5 TECHNICAL – the engineer's version

1. Gerald Popek and Robert Goldberg, in Formal Requirements for Virtualizable Third Generation Architectures, Communications of the ACM, July 1974, gave three requirements: equivalence, resource control and efficiency.
2. Their theorem: an architecture is classically virtualizable if the set of sensitive instructions is a subset of the privileged instructions.
3. x86 failed that test. John Robin and Cynthia Irvine identified 17 problematic instructions in a 2000 USENIX Security paper. Instructions such as P0PF silently did the wrong thing instead of trapping.
4. Two workarounds followed. VMware, founded in 1998, shipped binary translation: rewrite the guest's privileged instruction stream on the fly.
5. Xen, from the University of Cambridge, presented at SOSP in 2003, shipped **paravirtualization**: modify the guest kernel so it calls the hypervisor deliberately, through hypercalls, instead of executing offending instructions. Faster, but needs a modified guest.
6. Hardware assistance ended the argument. Intel VT-x shipped in November 2005; AMD-V, code-named Pacifica, shipped in May 2006. Both add a new guest mode, so an unmodified guest kernel can run at its expected privilege level and trap cleanly to the hypervisor.
7. Memory virtualization was still expensive until second level address translation arrived: AMD Nested Page Tables in Barcelona in 2007, Intel Extended Page Tables in Nehalem in 2008. Before that, hypervisors maintained shadow page tables in software.
8. Device passthrough needs an IOMMU: Intel VT-d and AMD-Vi. PCI-SIG's Single Root I/O Virtualization specification, published 2007, lets one physical card present many virtual functions, each assignable to one guest.

Hypervisor type	Runs on	Examples
Type 1, bare metal	The hardware	ESXi, Xen, Hyper-V, KVM
Type 2, hosted	An operating system	VirtualBox, Workstation

9. The type 1 versus type 2 split is a useful teaching device and a slightly leaky one. KVM, merged into Linux 2.6.20 in February 2007, turns the Linux kernel itself into a type 1 hypervisor while Linux remains a normal operating system. Experts differ on which box it belongs in.
10. From the host's point of view a KVM virtual machine is an ordinary process. You can see it in `ps`, one thread per virtual processor, its guest memory is an anonymous mapping, and you can kill it with `kill`.
11. Live migration in QEMU/KVM is iterative pre-copy: transfer all pages, then repeatedly transfer the pages dirtied during the previous pass, then stop the guest, transfer the remainder and the device state, and resume on the target. Typical stop time is tens to a few hundred milliseconds.
12. The **hypervisor tax** is the fraction of a machine consumed by virtualization rather than by customers. Werner Vogels has written that on older AWS designs this was around 30 per cent of the resources.

13. AWS attacked it with hardware. The path: C3 in 2013 offloaded network processing, C4 in 2014 offloaded storage, Amazon acquired the Israeli chip company Annapurna Labs in early 2015, and the full **Nitro System** shipped with C5 instances in 2017.
14. Nitro has three parts: Nitro Cards, which are dedicated hardware running VPC networking, EBS storage and instance storage; the Nitro Security Chip, which verifies firmware; and the Nitro Hypervisor, a thin KVM-based component that now mainly assigns memory and CPU.
15. Because almost nothing is left in software, AWS can also sell `.metal` instances, where a customer gets the whole physical server with no hypervisor at all, yet still gets normal EBS volumes and VPC networking.
16. Microsoft's equivalent story runs through programmable hardware. The Catapult project put FPGAs in Azure servers from 2015, Accelerated Networking used those FPGA-based SmartNICs to move virtual switching off the CPU, and Azure Boost, announced in 2023, offloads storage and networking to a dedicated card in the same spirit as Nitro.
17. Observation commands: `lscpu` shows the hypervisor vendor line inside a guest; `systemd-detect-virt` names the hypervisor; `dmidecode -s` `system-manufacturer` often reveals it; `virsh list` and `virsh domstats` inspect KVM guests from the host.

■ 43.4.6 WORDS – remember these

1. Hypervisor — the program that splits one computer into many — a virtual machine monitor providing equivalence, resource control and efficiency.
2. Type 1 versus type 2 — running on bare metal versus on top of an existing operating system — native versus hosted VMM.
3. Paravirtualization — the guest knows it is a guest and cooperates — replacing sensitive instructions with explicit hypercalls, as in Xen.
4. VT-x and AMD-V — processor features that make the trick cheap — hardware virtualization extensions adding a distinct guest execution mode, 2005 and 2006.
5. Oversubscription — selling more slices than you physically have — allocating virtual resources beyond physical capacity, relying on idle time.
6. Steal time — time your machine was ready but not scheduled — the `st` counter in Linux CPU accounting, the direct measure of host contention.
7. Live migration — moving a running machine without switching it off — iterative pre-copy of memory and device state to another host.
8. Nitro — AWS's offload hardware — dedicated cards and a minimal hypervisor that remove virtualization overhead and enable bare-metal instances, from 2017.

43.5 Containers

■ 43.5.1 PLAIN – in simple words

1. A virtual machine is a whole pretend computer, with its own operating system inside. That costs memory and takes tens of seconds to start.
2. Most of the time you did not want a computer. You wanted to run one program, with the exact files and libraries it expects.
3. A **container** gives you that. It is one ordinary program, running on the normal operating system, but wrapped so that it can only see its own files, its own processes and its own network.
4. Say this clearly, because almost every beginner gets it wrong: a container is not a small virtual machine. There is no second operating system inside it.

5. From the host's point of view it is just a process, visible in the process list, killable, schedulable, like any other.
6. The wrapping is done by features built into the Linux kernel. The container asks for a private view of the world, and the kernel gives it one.
7. Because there is nothing to boot, a container starts in tens of milliseconds instead of tens of seconds.
8. Because there is no second operating system, it uses megabytes of extra memory instead of hundreds of megabytes.
9. The other half of the idea is the **image**: a frozen, shareable copy of all the files the program needs, built once and run anywhere.
10. That is why containers solved the oldest complaint in software: "it works on my machine". Now the machine travels with the program.
11. The trade is isolation. Virtual machines are separated by hardware enforcement. Containers are separated by kernel bookkeeping, which is thinner, and a kernel bug can cross it.

■ 43.5.2 PLAIN – a picture in your head

1. Imagine a big office building with one security system, one power supply and one set of stairs. That is the operating system kernel.
2. A virtual machine is like building a whole separate house in the car park: its own foundations, its own wiring, its own everything.
3. A container is like giving one team a locked room inside the existing building, plus a rule at reception that they may not ask about any other team, plus a cap on how much electricity and how many desks they may use.
4. The team genuinely cannot see the other teams. Ask them who else is in the building and they will answer, honestly, "nobody".
5. Setting up a room takes minutes. Building a house in the car park takes months.
6. But the room is inside a building that somebody else controls. If the building's fire alarm has a flaw, everybody in it is affected.

Where this comparison breaks: the locked room suggests a physical wall. There is no wall. There is only the receptionist consistently refusing to answer certain questions. If you find a way to make the receptionist answer, you are out. That is exactly what a container escape is, and it is why you do not run untrusted code in a plain container without further sandboxing.

■ 43.5.3 PLAIN – a worked example

1. Here is a real Dockerfile for a small Python web service, with every line explained afterwards.

```
FROM python:3.12-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
RUN useradd -m appuser && chown -R appuser /app
USER appuser
EXPOSE 8000
ENV PORT=8000
CMD ["gunicorn", "-b", "0.0.0.0:8000", "app:application"]
```

2. FROM python:3.12-slim starts from an existing image that already contains a minimal Linux userland and Python 3.12. You are not building from nothing.
3. WORKDIR /app creates and enters a directory. Later commands run there.
4. COPY requirements.txt . copies just the list of dependencies in first.

5. `RUN pip install ...` installs those dependencies. Because this is a separate layer, and because only the requirements file was copied so far, this expensive step is reused from cache whenever your source changes but your dependency list does not. This ordering is the single most valuable trick in the file.
6. `COPY . .` now copies your actual source code.
7. `RUN useradd ...` creates a non-root user and gives it ownership.
8. `USER appuser` means every later instruction, and the running container, operate as that user. Running as root inside a container is a common and bad default.
9. `EXPOSE 8000` is documentation plus metadata. It does not open a port by itself.
10. `ENV PORT=8000` sets an environment variable inside the image.
11. `CMD [...]` is the command run when the container starts. The list form avoids a shell, so signals reach your process directly.
12. The commands to build and run it:

```
docker build -t festival-site:1.4 .
docker run -d --name web -p 8080:8000 \
  --memory 512m --cpus 1.5 festival-site:1.4
docker ps
docker logs -f web
docker exec -it web /bin/sh
docker stop web && docker rm web
```

13. `-p 8080:8000` maps port 8080 on the host to 8000 in the container.
14. `--memory 512m --cpus 1.5` are the limits. Without them, one container can consume the whole machine.

■ 43.5.4 PLAIN - what is really happening inside

1. When you run a container, the runtime asks the Linux kernel to start a process with a private view of six or seven kinds of thing. Each private view is called a **namespace**.
2. The PID namespace gives the process its own process numbering. The first process inside sees itself as PID 1 and cannot see the host's processes.
3. The network namespace gives it its own network interfaces, its own routing table and its own firewall rules. Usually a virtual cable is created, one end inside, one end on a bridge in the host.
4. The mount namespace gives it its own filesystem tree, so `/` inside is not `/` outside.
5. The user namespace lets an identity that is root inside map to an ordinary unprivileged identity outside.
6. The IPC namespace separates shared memory and message queues. The UTS namespace lets the container have its own hostname.
7. Separately, **cgroups** put ceilings on what the process may consume: so much memory, so much processor time, so much disk bandwidth. If it exceeds the memory ceiling, the kernel kills it.
8. The filesystem is assembled from stacked read-only layers plus one writable layer on top. Reads fall through the stack to the first layer that has the file. Writes go to the top layer only, copying the file up first.
9. Because the read-only layers are shared, ten containers from the same image store one copy of the common files, not ten.
10. The image itself is just those layers plus a small JSON document describing the command, environment, user and layer order.
11. A **registry** is a web service that stores and serves those layers and documents, addressed by content hash.

■ 43.5.5 TECHNICAL – the engineer’s version

1. The idea is old. chroot appeared in Version 7 Unix in 1979: change the apparent root directory of a process. It isolates the filesystem view and nothing else, and it was never a security boundary.
2. FreeBSD jails, by Poul-Henning Kamp, arrived in FreeBSD 4.0 in March 2000, adding process, network and user isolation to the chroot idea.
3. Solaris Zones shipped in Solaris 10 in 2005, with resource controls and a full virtualized environment per zone.
4. On Linux the pieces arrived separately. The mount namespace landed in kernel 2.4.19 in 2002. Process containers, developed at Google by Paul Menage and Rohit Seth, were renamed control groups and merged in kernel 2.6.24 in January 2008. The user namespace was completed in kernel 3.8 in 2013.
5. LXC, released in 2008, was the first tool to assemble those pieces into something usable. It required real Linux skill.
6. Docker changed the packaging, not the kernel. Solomon Hykes demonstrated it at PyCon in Santa Clara and it was open sourced in March 2013, out of a company then called dotCloud, renamed Docker Inc. later that year.
7. Docker’s contribution was the image format, the layer cache, the Dockerfile and a public registry: a developer workflow, not an isolation mechanism. Early versions used LXC underneath, replaced by its own libcontainer in version 0.9 in 2014.
8. Standardization followed. The **Open Container Initiative** was founded in June 2015 under the Linux Foundation, by Docker, CoreOS and others. Its runtime and image specifications reached version 1.0 in July 2017, and the distribution specification followed in 2021.
9. runc is the OCI reference runtime. containerd and CRI-O are the higher level runtimes that Kubernetes actually calls. Docker Engine today is a developer-facing layer over containerd.
10. Namespaces in the kernel, with their clone flags:

Namespace	Isolates	Flag
PID	Process numbering	CLONE_NEWPID
Network	Interfaces, routes	CLONE_NEWNET
Mount	Filesystem tree	CLONE_NEWNS
User	UID and GID mapping	CLONE_NEWUSER
IPC	Shared memory, queues	CLONE_NEWIPC
UTS	Hostname, domain	CLONE_NEWUTS

11. There are also cgroup and time namespaces, added later. cgroup v2, the unified hierarchy, is the current standard on modern distributions.
12. Overlay filesystems: overlayfs was merged in kernel 3.18 in 2014 and is the default storage driver. It presents a merged view of lower read-only directories and one upper writable directory, with copy-up on first write.
13. Image layers are content-addressed by SHA-256 digest. The image manifest lists layer digests; the config blob holds the command, environment and history. Pulling an image you already partly have transfers only the missing digests.
14. Additional hardening beyond namespaces and cgroups: seccomp-bpf to restrict system calls, AppArmor or SELinux for mandatory access control, dropping Linux capabilities, and read-only root filesystems.
15. For untrusted workloads, use a stronger boundary: gVisor, which intercepts system calls in userspace, or Kata Containers and AWS Firecracker, which put a genuine lightweight virtual ma-

chine around each container. Firecracker, open sourced by AWS in 2018, boots a micro virtual machine in about 125 milliseconds and is what runs AWS Lambda and Fargate.

16. Inspection commands: `lsns` lists namespaces, `nsenter` enters one, `systemd-cgls` and `systemd-cgtop` show cgroup trees, `docker image history` shows the layers and their sizes, `crictl ps` lists containers on a Kubernetes node.

■ 43.5.6 WORDS – remember these

1. Container — an isolated process with its own files and limits — a process group running under restricting namespaces and cgroups on a shared kernel.
2. Namespace — a private view of one kind of system resource — a kernel mechanism partitioning PID, network, mount, user, IPC, UTS and cgroup views.
3. cgroup — a ceiling on how much a process may consume — control group hierarchy enforcing CPU, memory and I/O limits, cgroup v2 unified.
4. Image — a frozen copy of everything the program needs — an ordered set of content-addressed layers plus a config manifest, per the OCI image spec.
5. Layer — one saved change on top of the previous ones — a tar archive of filesystem differences, shared and cached by digest.
6. Registry — the shop where images live — an OCI distribution-spec service storing manifests and blobs, such as Docker Hub or ECR.
7. OCI — the agreed rules so tools interoperate — the Open Container Initiative runtime, image and distribution specifications, founded June 2015.
8. Container escape — breaking out into the host — exploitation of a kernel or configuration flaw to cross the namespace boundary.

43.6 Kubernetes and orchestration

■ 43.6.1 PLAIN – in simple words

1. Containers are easy to run one at a time on your laptop.
2. Now imagine 400 of them, across 30 machines, each needing to find the others, restart when they die, and move when a machine is taken away.
3. Doing that by hand is impossible. You need a program that does it for you. That program is called an **orchestrator**.
4. **Kubernetes** is the one that won. The name is Greek for helmsman, and it is often shortened to K8s, because there are eight letters between K and s.
5. The idea at its heart is simple and worth memorizing: you describe what you want, and the system continuously works to make reality match.
6. You do not say “start three copies”. You say “there should be three copies”.
7. If one dies at 3am, nobody is paged, because nothing has been violated except temporarily. The system notices the gap and starts another.
8. If you edit the number to five, the system notices the gap and starts two more. The same machinery handles both cases.
9. This is called **declarative reconciliation**: state the desired end, let a control loop close the difference, forever.
10. Everything else in Kubernetes is detail on top of that one idea.

■ 43.6.2 PLAIN – a picture in your head

1. Think of a thermostat on a wall.
2. You do not tell the heater to switch on for eleven minutes. You set 22 degrees.
3. The thermostat measures, compares with your setting, and acts. Then it does it again, forever.
4. Open a window and the temperature drops; the thermostat responds without being told about the window.
5. Kubernetes is a building full of thermostats, one for each kind of thing.
6. One watches “how many copies of this program are running” against “how many should be”. Another watches “which machines are alive”. Another watches “what traffic goes where”.
7. Each loop is small, dull and independent. The intelligence is in the fact that they never stop running.

Where this comparison breaks: a thermostat has one input and one output. Kubernetes controllers act on each other’s outputs, so an error can propagate. A wrong setting does not sit still and look wrong; it is enthusiastically and repeatedly enforced across your entire cluster in seconds. That is the same property that makes it powerful and makes mistakes fast.

■ 43.6.3 PLAIN – a worked example

1. Here is a minimal but real pair of Kubernetes objects.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: festival-web
spec:
  replicas: 3
  selector:
    matchLabels:
      app: festival-web
  template:
    metadata:
      labels:
        app: festival-web
    spec:
      containers:
      - name: web
        image: registry.example.com/festival-site:1.4
        ports:
        - containerPort: 8000
        resources:
          requests:
            cpu: "250m"
            memory: "256Mi"
          limits:
            memory: "512Mi"
        readinessProbe:
          httpGet:
            path: /healthz
            port: 8000
---
apiVersion: v1
kind: Service
metadata:
  name: festival-web
spec:

```

```

selector:
  app: festival-web
ports:
- port: 80
  targetPort: 8000

```

2. `kind: Deployment` says this is a managed set of identical copies.
3. `replicas: 3` is the desired state. This one number is the whole contract.
4. `selector.matchLabels` says which pods this deployment considers its own. The labels in the template must match, or nothing works.
5. `image:` names the exact image, including a version tag. Using `latest` here is the classic mistake, because two nodes can then run different code.
6. `requests` is what the scheduler uses to decide where the pod fits. `limits` is what the kernel enforces at runtime.
7. Note there is a memory limit but no CPU limit. That is a deliberate and widely recommended pattern: CPU throttling hurts latency, while unlimited memory risks killing the node.
8. `readinessProbe` tells Kubernetes when this copy may receive traffic. Without it, traffic arrives before the program is ready and users see errors during every deployment.
9. The Service gives the three copies one stable internal name and one stable internal address, and spreads connections across whichever copies are ready.
10. Apply and observe:

```

kubectl apply -f festival.yaml
kubectl get deploy,rs,pods -l app=festival-web
kubectl describe pod festival-web-6c9f7d8b4-x2ktr
kubectl logs -f deploy/festival-web
kubectl rollout status deploy/festival-web
kubectl rollout undo deploy/festival-web

```

■ 43.6.4 PLAIN - what is really happening inside

1. `kubectl apply` sends your document to one program, the API server, which validates it and writes it into a database.
2. Nothing has run yet. All that exists is a written wish.
3. The deployment controller notices a Deployment with no matching ReplicaSet and creates one.
4. The ReplicaSet controller notices it should have three pods and has zero, so it creates three pod records. Still nothing is running.
5. The scheduler notices three pods with no node assigned. For each, it filters nodes that could fit, scores the survivors, picks the best, and writes the node name into the pod record.
6. On that node, an agent called the kubelet is watching for pods assigned to it. It sees one, pulls the image, and asks the container runtime to start it.
7. The kubelet then reports status back: starting, running, ready, or failed.
8. The endpoints controller notices a ready pod matching a Service's label selector, and adds its address to that Service's endpoint list.
9. On every node, a component programs the local packet rules so that traffic to the Service address is redirected to one of the current endpoint addresses.
10. If a pod dies, the kubelet reports it, the ReplicaSet controller sees two where three should be, and creates one more. The loop closes.
11. Nobody issued a command. Every step was a controller observing a difference and reducing it.

■ 43.6.5 TECHNICAL – the engineer’s version

1. Kubernetes descends from Borg, Google’s internal cluster manager, described publicly in the 2015 EuroSys paper Large-scale cluster management at Google with Borg, and from its successor experiment Omega.
2. The first Kubernetes commit was pushed to GitHub on 6 June 2014: 250 files and 47,501 lines of Go, shell and Markdown. Joe Beda, Brendan Burns and Craig McLuckie are the usually named originators, with Borg engineers including Brian Grant and Tim Hockin joining early.
3. Version 1.0 was released in July 2015, and Google donated the project to the newly formed Cloud Native Computing Foundation under the Linux Foundation at the same time. That donation is the reason competitors adopted it.
4. Core objects you must know:

Object	What it is for
Pod	Co-located containers, one unit
ReplicaSet	Keeps N identical pods up
Deployment	Rolling updates of ReplicaSets
Service	Stable name and virtual IP
Ingress	HTTP routing from outside
ConfigMap	Non-secret configuration data
Secret	Base64-encoded sensitive data
Namespace	A naming and policy boundary
Node	One worker machine

5. A pod is the smallest schedulable unit. Containers in a pod share a network namespace, so they reach each other on `localhost`, and can share volumes.
6. Control plane components: the API server, the only component that talks to the datastore; the scheduler, which assigns pods to nodes; the controller manager, which runs the built-in control loops; and `etcd`, a distributed key-value store using the Raft consensus algorithm.
7. On each worker: the kubelet, which owns pod lifecycle on that node, a network proxy component, and a container runtime speaking the Container Runtime Interface, usually `containerd` or `CRI-O`.
8. Be honest about Secrets. By default they are base64-encoded, not encrypted, in `etcd`. Encryption at rest must be enabled explicitly, and anyone with read access to the namespace can read them.
9. Ingress is being superseded. The Gateway API reached general availability for its core resources in October 2023 and is the direction of travel; Ingress remains extremely common and is not going away soon.
10. Scale limits, from the official documentation: no more than 5,000 nodes, 150,000 pods and 300,000 containers in a single cluster, with at most 110 pods per node.
11. Managed offerings remove the control plane work but not the concepts: EKS on AWS from June 2018, AKS on Azure from June 2018, and GKE on Google Cloud from 2015, which was the first.
12. Now the honest section, which matters more than the object list.
13. Most teams do not need Kubernetes. This is not a fashionable opinion; it is what the cost structure says.
14. Kubernetes solves problems that appear at a certain size: many services, many teams, high change rate, bin-packing pressure across many machines, and a genuine need for self-healing across node failures.

15. Below that size it adds a permanent, non-trivial tax: cluster upgrades every few months, a networking plugin to understand, ingress controllers, certificate rotation, RBAC, admission controllers, storage classes, and a new class of failure modes nobody on the team has seen before.
16. A useful test with three questions. Do you run more than about ten distinct services? Do you have someone whose job includes the platform? Would a single machine restarting cause a real outage today?
17. If the answer to all three is no, a managed container service such as ECS, Cloud Run or App Service, or even two virtual machines behind a load balancer, will serve you better and cost less in time.
18. Where experts disagree: some argue the standard interface is worth the tax even for small teams, because hiring and tooling are common. Others argue that complexity you do not need is complexity that will page you at 3am. Both are stated in good faith and depend on team size and turnover.

■ 43.6.6 WORDS - remember these

1. Orchestrator — the program that runs your containers for you — a cluster scheduler and lifecycle manager across many hosts.
2. Pod — the smallest thing Kubernetes runs — one or more containers sharing a network namespace and storage volumes, scheduled as a unit.
3. Deployment — the object that keeps N copies of your app alive — a controller managing ReplicaSets to provide declarative rolling updates.
4. Reconciliation — repeatedly closing the gap between wish and reality — a level-triggered control loop comparing desired and observed state.
5. etcd — the cluster's memory — a strongly consistent distributed key-value store using Raft, the only stateful control plane component.
6. kubelet — the agent on each machine — the node-level daemon that starts pods assigned to it and reports their status.
7. Service — a stable name in front of moving copies — a virtual IP and DNS name load-balancing to the current ready endpoints.

43.7 The Bezos API mandate

■ 43.7.1 PLAIN - in simple words

1. Around 2002, Jeff Bezos sent an internal message to everyone building software at Amazon.
2. It said, in effect: every team must publish its work as a service that other teams call over the network, and no team may reach into another team's data any other way.
3. That sounds like a boring technical rule. It was one of the most consequential management decisions in the history of the industry.
4. Before it, teams shared databases and linked directly to each other's code. That is fast to write and impossible to change later.
5. After it, the only way to use another team's work was to call its published interface, over the network, like an outsider.
6. And the last rule was the sharp one: build every interface as if a stranger outside the company would use it.
7. Nobody was selling anything to outsiders yet. But once every internal service was already built for outsiders, selling it later was a small step rather than a rewrite.

8. That is the link between the memo and AWS. AWS is what you get when a company spends four years turning itself into a set of public-quality services and then notices it can sell them.
9. Amazon never published the memo. What everyone quotes is one former employee's recollection, written down in 2011.

■ 43.7.2 PLAIN – a picture in your head

1. Think of a large hospital where every department shares one giant paper filing cabinet.
2. Radiology reaches into the pharmacy's drawer to check a prescription. Pharmacy reaches into billing's drawer. Everyone rummages everywhere.
3. It works, and it is fast, right up until one department reorganizes its drawer. Then four other departments break, and nobody knows who or why.
4. Now the director issues a rule: no department may open another department's drawer. If you need something, you fill in that department's request form and they answer it.
5. At first this is slower and everybody hates it. Forms take longer than rummaging.
6. But now pharmacy can completely redesign its drawer over a weekend, and nobody else notices, because the form did not change.
7. And one day the hospital realizes that since every department already answers formal requests, it can answer requests from other hospitals too, for a fee.

Where this comparison breaks: a paper form is slow but reliable. A network call can be slow, can fail halfway, can succeed while appearing to fail, and can arrive twice. The hospital analogy makes the mandate look like pure gain. The real version bought flexibility with a permanent bill in latency and in failure handling, and that bill never stops being paid.

■ 43.7.3 PLAIN – a worked example

1. Here is the mandate as it is publicly reported. The source is a long post by Steve Yegge, an engineer who worked at Amazon for about six years and then at Google. He wrote it in October 2011 as an internal Google memo and posted it publicly by accident.
2. His introduction is "His Big Mandate went something along these lines". The rules, quoted:
3. One. "All teams will henceforth expose their data and functionality through service interfaces."
4. Two. "Teams must communicate with each other through these interfaces."
5. Three. "There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network."
6. Four. "It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols – doesn't matter."
7. Five. "All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions."
8. Then the closing line: "Anyone who doesn't do this will be fired."
9. The honest version: in Yegge's original post this is a list of seven items, not five plus a closing line. Item six is the firing line, and item seven is a one-line pleasantry, which he immediately identifies as his own joke. The five substantive rules plus the firing line is the accurate summary.
10. Yegge also wrote that item six "was quite real, so people went to work". He was recalling, not quoting a document. Amazon has never released the memo, so the wording is a recollection and should always be described that way.
11. Note what rule four is not doing. It does not pick a technology. That is deliberate and correct: the mandate is about boundaries, not protocols.

■ 43.7.4 PLAIN – what is really happening inside

1. Follow what the rule forces, step by step. This is the part that matters.
2. Once teams may only call each other over the network, they must first be able to find each other. So somebody has to build a service registry, and the registry is itself a service.
3. Once hundreds of services call each other, any team can accidentally flood another. Yegge's words: "every single one of your peer teams suddenly becomes a potential DOS attacker". So quotas and throttling become mandatory everywhere, not optional.
4. Once a request crosses twenty services, an alarm has to reach the right team. Yegge again: a ticket "might bounce through 20 service calls before the real owner is identified", and at fifteen minutes per bounce that is hours. So you must build ownership metadata and automated pager routing.
5. Once you cannot see inside another team's code, "monitoring and QA are the same thing". A service that answers "I am fine" may have nothing working except the part that says "I am fine". So health checks must exercise real behaviour, and at that point monitoring has become automated testing.
6. Once debugging spans teams, it "is basically impossible unless there is a universal standard way to run every service in a debuggable sandbox". So you must build sandboxes and standardized local environments.
7. Look at that list again: discovery, quotas, paging, deep health checking, sandboxes. Every one of those is a component of a public cloud platform.
8. Amazon did not set out to build platform tooling. The mandate made it unavoidable, and four years later they had it.

■ 43.7.5 TECHNICAL – the engineer's version

1. Organizational consequence one: the **two-pizza team**. Bezos's stated rule was that a team should be small enough to be fed by two pizzas, roughly six to ten people. The number is not the point; the bound on communication paths is.
2. Organizational consequence two: **you build it, you run it**. Werner Vogels, Amazon's Chief Technology Officer, stated it in an ACM Queue interview with Jim Gray in 2006. The team that writes a service also operates it and carries its pager.
3. That closes a loop. If the author of the 3am page is also the person woken by it, quality arguments stop needing management.
4. Organizational consequence three: **Conway's law**. Melvin Conway wrote in *How Do Committees Invent?*, published in *Datamation* in April 1968, that organizations produce designs which copy their own communication structures.
5. The mandate is Conway's law used deliberately. Change the communication structure, and the architecture follows. This is sometimes called the inverse Conway manoeuvre.
6. Now the costs, stated plainly, because the reader is owed them.
7. Latency. An in-process function call is on the order of nanoseconds. A call to another service in the same datacentre is on the order of 0.5 to 2 milliseconds. That is a factor of roughly a million.

Call type	Typical latency
Function call in process	1 to 100 ns
Same-host loopback call	50 to 200 us
Same-AZ service call	0.5 to 2 ms
Cross-region call	50 to 250 ms

8. Failure semantics. A local call either returns or throws. A network call can also time out with the

work completed, be retried and applied twice, or succeed with the response lost. Every service boundary therefore needs idempotency keys, retries with backoff, timeouts and circuit breakers.

9. Data integrity. You can no longer wrap two teams' updates in one database transaction. You are pushed into eventual consistency and compensating actions.
10. Operational cost. Each service needs its own build, deployment, monitoring, on-call rota, dependency updates and security patching.
11. The most important honest point in this chapter: most companies that copied the microservice structure did not copy the mandate's discipline.
12. The mandate had four properties that copies usually lack. It was absolute, with no exceptions. It was enforced from the top. It required interfaces to be externalizable, which forces documentation, versioning and stability. And it was accompanied by real investment in the platform tooling that the consequences demanded.
13. Split a system into thirty services without discovery, quotas, tracing, ownership routing and sand-boxes, and you have not built a service-oriented architecture. You have built a distributed monolith, which has all the latency and failure modes of distribution and none of the independence.
14. Where experts disagree: some argue the mandate is a timeless engineering principle, others that it was a response to a specific scale and coupling crisis at Amazon in 2001 and 2002, and that applying it to a twelve-person company is cargo cult. The second view is the majority among practitioners today.

■ 43.7.6 WORDS – remember these

1. API mandate — the 2002 rule that teams talk only through published interfaces — an organization-wide prohibition on any interprocess communication other than service calls.
2. Externalizable — built as if a stranger will use it — designed with the documentation, versioning and stability required for third-party exposure.
3. Two-pizza team — a team small enough to feed with two pizzas — an organizational unit sized to bound internal communication paths.
4. You build it, you run it — the authors carry the pager — development teams own production operation of their own services.
5. Conway's law — you ship your org chart — systems mirror the communication structure of the organizations that design them, Conway 1968.
6. Distributed monolith — services that must all deploy together — a system with the costs of distribution and the coupling of a monolith.

43.8 How AWS actually started

■ 43.8.1 PLAIN – in simple words

1. By 2000 Amazon was a large online shop with a serious internal problem.
2. Every new project spent months on the same groundwork: getting machines, setting up databases, storage, networking and monitoring.
3. Teams were reinventing the same plumbing over and over, and waiting on each other to do it.
4. The API mandate had already forced them to expose everything as a service. The next thought was obvious: make the infrastructure itself a service.
5. In 2003, engineers wrote a proposal describing computing capacity offered as a service, so that any team could ask for a server through an interface.
6. In parallel, a senior manager wrote a proposal arguing that this could be a business, sold to the outside world.

7. Amazon approved both. Development of the compute service was set up in Cape Town, South Africa, of all places, because the engineer leading it wanted to move home.
8. The public services arrived in 2006: storage in March, and rentable computers in August.
9. It was not a side project selling leftovers. It was a deliberate new business with its own staffing plan.

■ 43.8.2 PLAIN – a picture in your head

1. Imagine a large restaurant chain that keeps building its own kitchens.
2. Every new branch designs plumbing, gas, extraction and cold storage from scratch, badly, and slowly.
3. Head office finally standardizes: one kitchen design, ordered as a kit, ready in a week.
4. Branches stop thinking about plumbing and start thinking about food. Opening a branch goes from a year to a month.
5. Then somebody points out that other restaurant chains have the same problem, and that the kit could be sold.
6. The kit business becomes larger and far more profitable than the original restaurants.

Where this comparison breaks: a kitchen kit is a product you build once and ship. AWS had to be operated forever, at growing scale, for customers who could leave. Also, the myth this analogy tempts you into is that head office sold its spare kitchens. It did not. It designed and built a product from scratch, then sold it, which is a completely different thing.

■ 43.8.3 PLAIN – a worked example

1. The dates, in order, from public records.
2. July 2002: Amazon launches Amazon.com Web Services, a free interface letting affiliates pull product data. This is where the name comes from. It is not cloud computing.
3. 2003: Chris Pinkham and Benjamin Black write a short internal paper proposing standardized, automated infrastructure delivered as a service, and noting that virtual servers could be sold.
4. 2003: Andy Jassy, who had joined Amazon in 1997 and become Bezos's technical assistant in 2001, writes a separate proposal for the business, including a plan for the first 57 hires. It is approved by the senior team.
5. 2004: Pinkham moves to Cape Town and opens a development centre there to build the compute service.
6. 3 November 2004: Simple Queue Service launches in beta. Chronologically this is the first thing recognizably cloud.
7. 14 March 2006: Simple Storage Service, S3, launches generally available, at 15 US cents per gigabyte-month.
8. 25 August 2006: Elastic Compute Cloud, EC2, launches in limited beta. One instance size only, 1.7 GHz, 1.75 GB of memory, 160 GB of local disk, at 10 US cents per hour. It reached general availability in October 2008.
9. From 10 cents an hour in 2006 to 128.7 billion US dollars of revenue in 2025 is the whole arc.

■ 43.8.4 PLAIN – what is really happening inside

1. Why Amazon and not IBM, Microsoft, Sun or a telephone company.
2. First, Amazon already had to run huge, spiky, low-margin infrastructure, and was already good at it out of necessity.
3. Second, the API mandate had already forced internal services to be built for external use. The hard organizational work was done before the product was conceived.

4. Third, retail is a low-margin business, so Amazon's culture was to treat infrastructure cost as an enemy. A company used to 3 per cent margins prices very differently from one used to 60 per cent margins.
5. Fourth, nobody else wanted the customers. In 2006 the buyers were startups with no money. Established vendors sold to enterprises with purchase orders and salespeople, and a self-service credit-card product looked like a toy.
6. Fifth, Amazon was willing to be unprofitable for years and did not have to protect an existing licence business. Microsoft and others did.
7. Now correct the myth. The popular story is that AWS was built to rent out servers left idle after the Christmas shopping peak.
8. It is false. Werner Vogels, Amazon's Chief Technology Officer, publicly rejected it in 2011, and Andy Jassy has repeated the correction many times.
9. The mechanics also do not work. Retail's peak is Q4, exactly when external customers would need capacity most, so you would have to evict them at the worst moment. And AWS ran on different hardware, in different facilities, with different software, from the start.
10. The kernel of truth is smaller and duller: Amazon's own experience of running infrastructure badly and expensively is what taught it what the product should be.

■ 43.8.5 TECHNICAL - the engineer's version

1. The service launch timeline, with dates as published:

Year	Service	Date
2004	SQS beta	3 Nov 2004
2006	S3	14 Mar 2006
2006	EC2 beta	25 Aug 2006
2007	SimpleDB beta	14 Dec 2007
2008	EBS	20 Aug 2008
2008	CloudFront	18 Nov 2008
2009	ELB, CloudWatch	18 May 2009
2009	VPC	25 Aug 2009
2009	RDS	26 Oct 2009
2010	Route 53	5 Dec 2010
2012	DynamoDB	19 Jan 2012
2012	Redshift	28 Nov 2012
2014	Lambda, ECS	13 Nov 2014
2017	Fargate, Nitro C5	Nov 2017
2018	EKS general availability	June 2018

2. Read the order. Storage, then compute, then block storage, then content delivery, then load balancing and monitoring, then networking, then managed databases. Each launch is a response to what the previous one made painful.
3. VPC arriving three years after EC2 is the most instructive gap. The original EC2 put every instance on a flat shared network with a public address, which enterprises would not accept.
4. Financially, AWS was reported separately for the first time in Amazon's first quarter 2015 results, revealing a business with roughly 5 billion US dollars of annualized revenue that was already profitable. That disclosure repriced Amazon's stock and forced competitors to respond.

5. Current figures, from company filings. Full year 2025 AWS net sales were 128,725 million US dollars, with segment operating income of 45,606 million, an operating margin near 35 per cent, against total Amazon net sales of 716,924 million.
6. In the quarter ending June 2026, AWS reported 42.2 billion US dollars, growth of about 37 per cent, and operating income of 16.6 billion, which Amazon's chief executive described as the fastest growth in eighteen quarters.
7. Reference customer for scale: Netflix began moving to AWS in 2008 after a database corruption incident halted DVD shipping for three days, and completed the shutdown of its last own-datacentre systems in January 2016. It took seven years, and Netflix is a well-resourced engineering organization.
8. Where sources disagree. Some accounts credit the 2003 Pinkham and Black paper as the origin of AWS; others credit Jassy's business proposal; Black himself has written that theirs was one of several parallel efforts. The honest summary is that both existed in 2003 and both mattered.

■ 43.8.6 WORDS – remember these

1. AWS — Amazon's cloud business — Amazon Web Services, launched publicly 2006, 128.7 billion US dollars of net sales in 2025.
2. EC2 — rentable computers by the hour — Elastic Compute Cloud, virtual machines billed per second with a 60 second minimum, beta August 2006.
3. S3 — a place to put files, reached over the web — Simple Storage Service, object storage launched 14 March 2006, the first generally available service.
4. SQS — a queue you do not have to run — Simple Queue Service, beta November 2004, chronologically the first AWS service.
5. Two-pizza origin — small teams owning services — the organizational precondition that let Amazon staff many independent services at once.
6. Spare capacity myth — the false story that AWS resold leftover retail servers — publicly denied by Amazon's CTO in 2011 and contradicted by the timeline.

43.9 The AWS services you actually need to know

■ 43.9.1 PLAIN – in simple words

1. AWS advertises more than 200 services. You will use about eighteen.
2. Here is each one, in a sentence, as the problem it solves.
3. EC2: I need a computer. Rent a virtual machine by the second.
4. S3: I need somewhere to put files that never fills up. Store objects by name, reached over the web.
5. EBS: my rented computer needs a hard disk that survives a reboot. Attach a network-backed disk to one machine.
6. VPC: I need my machines on a private network of my own design, not on the open internet.
7. Route 53: I need names to turn into addresses, and I need to steer traffic. This is DNS, run for you.
8. RDS: I need a normal database, but I do not want to install, patch, back up or fail it over myself.
9. DynamoDB: I need a store that never slows down as it grows, and I can live with looking things up only by key.
10. Lambda: I need to run a small piece of code when something happens, and I do not want a server sitting there waiting.
11. IAM: I need to say exactly who may do what. This is the permission system, and it applies to every other service.
12. CloudFront: my users are far away and my files are large. Keep copies near them.

13. ELB and ALB: I have several machines and one address. Spread the traffic and skip the dead ones.
14. SQS: I need to hand work to something else without waiting for it, and without losing it if the other side is down.
15. SNS: I need to tell many interested parties that something happened.
16. ECS: I have containers and I want them run for me, simply.
17. EKS: I have containers and I want Kubernetes, without operating the control plane myself.
18. CloudWatch: I need to see what is happening. Logs, numbers and alarms.
19. CloudFormation: I want my whole setup written down as a file, so it can be rebuilt exactly.
20. Secrets Manager: I need to keep passwords somewhere that is not the source code.

■ 43.9.2 PLAIN – a picture in your head

1. Think of building a shop in a market town.
2. EC2 is renting the shop unit. EBS is the storeroom attached to your unit. S3 is the town warehouse where anyone with a key can drop off boxes.
3. VPC is the private courtyard you rent, with your own gate and your own path layout. Route 53 is the street sign that tells people where your shop is.
4. ELB is the doorman who splits an arriving crowd between your three counters and stops sending people to the one that has closed.
5. RDS is a bookkeeper the market provides, who keeps your ledger, backs it up nightly and covers for himself when ill.
6. SQS is the in-tray by the back door. Deliveries go in whether you are ready or not, and you take them at your pace.
7. IAM is the master key system: who may open which door, when, from where.
8. CloudWatch is the set of meters and the alarm bell. CloudFormation is the architect's drawing from which the whole shop could be rebuilt.

Where this comparison breaks: in a market town you rent one unit and that is the bill. Here, almost every one of these has three or four separate meters running at once, and the doorman charges by the crowd as well as by the hour.

■ 43.9.3 PLAIN – a worked example

1. A three-tier web application, in AWS parts, in the order you would create them.
2. VPC with six subnets across two availability zones: two public, two private for applications, two private for the database.
3. An Application Load Balancer in the public subnets, holding the TLS certificate.
4. An Auto Scaling group of EC2 instances in the private application subnets, each with a gp3 EBS root volume.
5. RDS for PostgreSQL in the database subnets, with a standby in the second availability zone.
6. S3 for user uploads, with CloudFront in front of it for delivery.
7. Route 53 holding the hosted zone and an alias record pointing at the load balancer.
8. IAM roles attached to the instances so that no access key is ever written to disk.
9. Secrets Manager holding the database password, fetched at startup.
10. CloudWatch collecting logs and metrics, with alarms on error rate and database CPU.
11. CloudFormation or Terraform describing all of the above in one repository.

■ 43.9.4 PLAIN – what is really happening inside

1. Every one of these is reached the same way: an HTTPS request to a region-specific endpoint, signed with a key derived from your credentials.

2. That signature scheme is called Signature Version 4. It hashes the request, the date, the region and the service name, so a signature stolen for one region and service is useless elsewhere.
3. The AWS console, the aws command line tool and every language SDK all do exactly this. There is no private channel.
4. Services are separated into control plane operations, which create and modify resources, and data plane operations, which use them.
5. That split matters during outages. A control plane failure stops you creating new things while existing things keep serving.
6. Almost every service integrates with two others by default: IAM decides whether the call is allowed, and CloudTrail records that it happened.

■ 43.9.5 TECHNICAL – the engineer’s version

1. Equivalents across the three large providers. Names are current as of August 2026; providers rename services often, so check before quoting.

AWS	Azure	Google Cloud
EC2	Virtual Machines	Compute Engine
S3	Blob Storage	Cloud Storage
EBS	Managed Disks	Persistent Disk
VPC	Virtual Network	VPC
Route 53	Azure DNS	Cloud DNS
RDS	Azure SQL, Flexible	Cloud SQL
DynamoDB	Cosmos DB	Firestore, Bigtable
Lambda	Azure Functions	Cloud Run functions
IAM	Entra ID plus RBAC	Cloud IAM
CloudFront	Azure Front Door	Cloud CDN
ALB	Application Gateway	Cloud Load Balancing
SQS	Storage or Service Bus	Pub/Sub
SNS	Event Grid	Pub/Sub
ECS	Container Apps	Cloud Run
EKS	AKS	GKE
CloudWatch	Azure Monitor	Cloud Monitoring
CloudFormation	ARM, Bicep	Deployment Manager
Secrets Manager	Key Vault	Secret Manager

2. The mapping is approximate on purpose. DynamoDB and Cosmos DB have different consistency models. SQS is a queue with one consumer per message; Pub/Sub is a topic with fan-out, so it maps to both SQS and SNS.
3. Durability and availability commitments, as published in the service level agreements: S3 Standard is designed for 99.99999999 per cent durability, eleven nines, with a 99.9 per cent availability service level agreement.
4. Eleven nines means that for ten million objects you would statistically expect to lose one object every ten thousand years. It is a design target derived from redundancy, not a measured result.
5. EC2 offers a 99.99 per cent regional service level agreement when instances are spread across two or more availability zones, and 99.5 per cent for a single instance with certain storage configurations.

6. Storage duplication: S3 Standard stores objects redundantly across at least three availability zones. S3 One Zone-IA deliberately does not, which is why it is cheaper and why you must not use it for anything irreplaceable.
7. Service quotas are real and will surprise you: default limits such as 5 VPCs per region, 5 Elastic IP addresses per region, and per-instance-family vCPU limits are common causes of failed deployments. Check `aws service-quotas list-service-quotas`.
8. Useful commands: `aws sts get-caller-identity` answers “who am I”, which is the first thing to run when something is denied; `aws configure list` shows which credentials and region are in effect.

■ 43.9.6 WORDS - remember these

1. EC2 — a rented computer — Elastic Compute Cloud instances, billed per second with a 60 second minimum, in families tuned for compute, memory or storage.
2. S3 — files stored by name on the web — object storage with eleven nines design durability and a flat key namespace per bucket.
3. EBS — a disk for one machine — network-attached block storage, gp3 by default, snapshottable, tied to one availability zone.
4. VPC — your own private network in the cloud — a logically isolated virtual network with your own address ranges, subnets and route tables.
5. IAM — the rules about who may do what — Identity and Access Management, evaluating identity and resource policies on every API call.
6. Lambda — run code without a server — a function-as-a-service runtime billed per millisecond of execution and per request.
7. Signature Version 4 — how a request proves it is really you — the AWS request signing scheme scoped to date, region and service.

43.10 Azure, Google Cloud and the rest

■ 43.10.1 PLAIN - in simple words

1. Microsoft announced its cloud, then called Windows Azure, in October 2008, and opened it for paying customers on 1 February 2010.
2. Its first version made a bet that turned out to be wrong. It offered a platform: you gave it your application, it ran it, and you never saw a machine.
3. That was elegant, but customers did not want elegant. They wanted to move the servers they already had, unchanged.
4. So Microsoft turned around and added plain virtual machines, and then Linux, and then everything else. It renamed itself Microsoft Azure in 2014 to signal that it was no longer a Windows product.
5. That late start cost Microsoft the developer-first market, which AWS held.
6. But Microsoft had something AWS did not: it was already inside almost every large company on earth, selling Windows, Office and, crucially, the identity system that decides who employees are.
7. So Azure won the enterprise from the inside. If your staff already log in with a Microsoft account, the cloud that understands that account has an enormous advantage.
8. Google Cloud started earlier in one narrow sense and later in every other. It is respected for containers, networking and data analysis, and is a clear third by revenue.
9. Below the big three there is a real second tier, and for many small projects it is cheaper and simpler.

■ 43.10.2 PLAIN – a picture in your head

1. Imagine three shops selling building materials.
2. The first, AWS, opened years earlier, stocks absolutely everything, and its staff assume you are a professional builder. The catalogue is enormous and the labels are terse.
3. The second, Azure, is owned by the company that already supplied the doors, locks and keys for every office in town. It says: your existing keys work here, and we will send someone to your site.
4. The third, Google Cloud, is run by engineers who built the largest structures in the world for themselves, and now sell the tools. The tools are excellent. The shop assistants are fewer.
5. Around the corner are small shops selling four or five items at half the price, with a printed price list you can understand in a minute.

Where this comparison breaks: building materials are interchangeable between shops. Cloud services are not. Once your identity, your networking and your managed databases are with one provider, moving is a project measured in quarters, not an afternoon of carrying bricks.

■ 43.10.3 PLAIN – a worked example

1. Azure's naming conventions are worth learning, because they are consistent and they differ from AWS.
2. Everything lives in a **resource group**, which is a folder for related resources with a shared lifecycle. AWS has no exact equivalent; the closest habits are tags and stacks.
3. Above resource groups sit subscriptions, which are the billing boundary, and above those, management groups, which apply policy.
4. Identity is not per subscription. It comes from a **tenant** in Entra ID, which can span many subscriptions.
5. So the Azure hierarchy reads: tenant, management group, subscription, resource group, resource.
6. In AWS the hierarchy reads: organization, organizational unit, account, resource. There is no resource group layer, which is why AWS practitioners use separate accounts for separation far more aggressively.
7. Creating a machine, side by side:

```
# Azure
az group create -n festival-rg -l centralindia
az vm create -g festival-rg -n web1 \
  --image Ubuntu2404 --size Standard_B2s \
  --admin-username azureuser --generate-ssh-keys

# AWS
aws ec2 run-instances --image-id ami-0abcd1234 \
  --instance-type t3.medium --key-name mykey \
  --subnet-id subnet-0123456789abcdef0
```

■ 43.10.4 PLAIN – what is really happening inside

1. The reason Entra ID is described as Microsoft's real moat is worth understanding mechanically.
2. Most large organizations authenticate employees against Active Directory, a system Microsoft shipped in Windows 2000. It holds users, groups, machines and policies.
3. Entra ID, which was called Azure Active Directory until Microsoft renamed it on 11 July 2023, is the cloud version. It issues tokens for web applications using OAuth 2.0 and OpenID Connect.
4. Once employees sign in to Microsoft 365 with Entra ID, the same identity can authorize Azure resources, third-party software and internal applications.

5. That means the decision “which cloud” is partly already made by a decision taken years earlier about email and file sharing.
6. Switching cost is therefore not mainly about servers. It is about identity, group membership, conditional access rules and hundreds of application integrations.
7. Hybrid follows from the same fact. Azure Arc, Azure Stack and ExpressRoute exist so that machines in a company’s own building appear in the same management and identity plane as machines in Azure.

■ 43.10.5 TECHNICAL – the engineer’s version

1. Azure timeline. Announced at the Professional Developers Conference on 27 October 2008 by Ray Ozzie, under the internal codename Red Dog, led by Dave Cutler, the architect of Windows NT.
2. Commercially available 1 February 2010, initially as a platform service with web roles and worker roles, Blob storage, SQL Azure and Service Bus.
3. Infrastructure services, meaning ordinary virtual machines, entered preview in June 2012 and reached general availability in April 2013. Renamed from Windows Azure to Microsoft Azure on 25 March 2014.
4. Position today: Azure passed 100 billion US dollars of annual revenue in Microsoft’s fiscal year 2026, growing 41 per cent from 75 billion in fiscal 2025. Microsoft states over 70 Azure regions.
5. Google Cloud timeline. App Engine, a platform service, previewed 7 April 2008. Compute Engine was announced at Google I/O on 28 June 2012 in limited preview, opened broadly in May 2013 and reached general availability in December 2013. Google Kubernetes Engine launched in 2015, the first managed Kubernetes service.
6. Google Cloud’s strengths are concentrated: Kubernetes, since it wrote it; networking, since it owns an unusually large private backbone; and data analytics, where BigQuery separates storage from compute and is genuinely ahead. Revenue was 17.664 billion US dollars in the December 2025 quarter, up 48 per cent.

Provider	Regions	Recent revenue
AWS	39, 123 AZs	128.7 bn, FY2025
Azure	Over 70	Over 100 bn, FY2026
Google Cloud	43, 130 zones	17.7 bn, Q4 2025

7. The rest of the field, briefly and honestly. Oracle Cloud Infrastructure competes on database licensing terms and, since 2024, on large AI compute contracts. IBM Cloud is strongest in regulated industries and mainframe adjacency. Alibaba Cloud leads in China and much of South East Asia.
8. DigitalOcean, Hetzner and OVHcloud sell simple virtual servers at a fraction of hyperscaler list prices, with flat pricing and generous or unmetered traffic. For a small production workload the difference can be five to ten times.
9. Cloudflare is a different shape: a network-first provider whose Workers run code at edge locations and whose R2 object storage charges no egress fee, which is a direct attack on the industry’s most disliked charge.
10. The trade is real. The second tier gives you fewer managed services, fewer compliance certifications, smaller regional footprints and less depth of support. For a database-plus-web-server application, that often does not matter. For a regulated enterprise, it does.

■ 43.10.6 WORDS – remember these

1. Azure — Microsoft's cloud — announced October 2008, commercially available 1 February 2010, over 100 billion US dollars in fiscal 2026.
2. Entra ID — the Microsoft login your company already uses — the cloud identity service formerly named Azure Active Directory, renamed 11 July 2023.
3. Resource group — an Azure folder for related things — a management and lifecycle boundary containing resources within one subscription.
4. Hybrid cloud — some machines yours, some rented, managed together — a deployment model joining on-premises infrastructure with public cloud under one control and identity plane.
5. Hyperscaler — one of the very large providers — an operator running global fleets at a scale where custom hardware and custom networks are economic.
6. Egress-free storage — object storage with no charge to read data out — Cloudflare R2's pricing model, contrasted with per-gigabyte egress charges.

43.11 The service models: IaaS, PaaS, SaaS and FaaS

■ 43.11.1 PLAIN – in simple words

1. The models differ in one thing only: where the line is drawn between what you look after and what the provider looks after.
2. **IaaS**, infrastructure as a service. You rent machines, disks and networks. You install and patch the operating system, the runtime and your code. You have the most control and the most work.
3. **PaaS**, platform as a service. You give the provider your code. It provides the operating system, the runtime, scaling and deployment. You lose some control and most of the work.
4. **SaaS**, software as a service. You do not write anything. You use finished software over the web and configure it. Gmail, Salesforce, Zoom.
5. **FaaS**, function as a service, also called serverless. You give the provider one function. It runs it only when something triggers it, and charges only for the milliseconds it runs.
6. These are not four products. They are four points on one line, and the line is "how much of the stack do I own".
7. Nothing forces you to pick one. Real systems mix all four, usually without anybody deciding to.

■ 43.11.2 PLAIN – a picture in your head

1. The standard analogy is pizza as a service, and it is genuinely useful.
2. Made at home: you buy flour, cheese, an oven, gas and a table. That is owning your own servers.
3. Take and bake: you buy a ready pizza and use your own oven, gas and table. That is infrastructure as a service.
4. Delivered: someone else makes and cooks it; you supply the table, the drinks and the dining room. That is platform as a service.
5. Dining out: you supply nothing but yourself and money. That is software as a service.

Where this comparison breaks, and it breaks in three specific places. First, it suggests you own less as you move along, when in fact you still own all the important things: your data, your access rules and your legal obligations. Second, pizza has one dimension of choice. Cloud stacks let you take the oven from one supplier and the dough from another, which the analogy cannot express. Third, and worst, the analogy implies dining out costs the most. In cloud pricing the fully managed option is often the cheapest at small scale and the most expensive at large scale, and the crossover is the whole argument.

■ 43.11.3 PLAIN – a worked example

1. The same task, a small web application, at each level.
2. IaaS: launch an EC2 instance, install Ubuntu updates, install Python, install a web server, write a service unit file, configure log rotation, set up certificate renewal, and repeat all of it when you scale out.
3. PaaS: push your code to Azure App Service or Google Cloud Run. It builds a container, runs it, terminates TLS, scales copies with traffic, and rolls back if health checks fail.
4. FaaS: write one function that handles one request, register it against an HTTP trigger, and let the provider create and destroy execution environments as demand moves.
5. SaaS: realize you were building a form-and-spreadsheet tool and buy one.

Layer	IaaS	PaaS
Application code	You	You
Runtime, libraries	You	Provider
Operating system	You	Provider
Virtualization	Provider	Provider
Servers, network	Provider	Provider

■ 43.11.4 PLAIN – what is really happening inside

1. The important consequence of the split is not convenience. It is who is accountable when something goes wrong.
2. This is formalized as the **shared responsibility model**, and every major provider publishes a version of it.
3. The provider is responsible for security **of** the cloud: the buildings, the hardware, the hypervisor, the physical network, and the managed service software itself.
4. The customer is responsible for security **in** the cloud: operating system patching where applicable, network rules, identity and permissions, data encryption choices, and application code.
5. The line moves with the model. On IaaS you patch the guest operating system. On PaaS the provider does. On SaaS almost everything technical is theirs.
6. One thing never moves. Access control and data classification are always yours, in every model, without exception.
7. This is why almost every publicized cloud breach is a customer-side misconfiguration rather than a provider failure: a storage bucket made public, a permission granted too widely, a key committed to a repository.

■ 43.11.5 TECHNICAL – the engineer's version

1. The definitions are standardized in NIST SP 800-145, September 2011, which defines IaaS, PaaS and SaaS. FaaS post-dates it and is generally treated as a specialization of PaaS.
2. Responsibility split, stated as the providers state it:

Concern	IaaS	PaaS	SaaS
Data and access	You	You	You
Application code	You	You	Provider
Runtime, middleware	You	Provider	Provider
Guest OS patching	You	Provider	Provider
Host and hardware	Provider	Provider	Provider

Concern	IaaS	PaaS	SaaS

- Encryption responsibilities are subtler than the table suggests. Providers supply encryption at rest by default in most storage services, but key custody is a choice: provider-managed keys, customer-managed keys in a key management service, or customer-supplied keys held outside entirely.
- Compliance is inherited, not transferred. A provider's ISO/IEC 27001 or SOC 2 report covers their controls. Your auditor still tests yours. The document that connects the two is the provider's customer responsibility matrix.
- Service level agreements are financial instruments, not promises of uptime. A 99.99 per cent monthly agreement permits about 4 minutes 23 seconds of downtime per month, and the remedy for breach is typically a service credit of 10 to 100 per cent of that service's fee. It does not compensate your business loss.
- Availability arithmetic that engineers get wrong: dependencies multiply. A system depending on four independent components each at 99.9 per cent has a theoretical availability of 0.999 to the fourth power, which is 99.6 per cent, or about three hours per month.

■ 43.11.6 WORDS - remember these

- IaaS — renting raw machines — infrastructure as a service, where the tenant manages the guest operating system and everything above it.
- PaaS — hand over code, get a running service — platform as a service, provider-managed runtime, scaling and deployment.
- SaaS — finished software you just use — software as a service, delivered multi-tenant over the network, configured not coded.
- FaaS — one function, run on demand — function as a service, event-triggered, billed per invocation and per millisecond.
- Shared responsibility — who is to blame for what — the provider secures the cloud, the customer secures what they put in it.
- Service level agreement — the promised uptime and the refund if missed — a contractual availability target with service credits as the sole remedy.

43.12 Serverless, properly

■ 43.12.1 PLAIN - in simple words

- Serverless does not mean there are no servers. There are servers. You do not see them, name them, patch them or pay for them when idle.
- A better name would have been "someone else's servers, allocated per request". That name lost.
- You upload a function: a piece of code with one entry point.
- You attach a **trigger**: an HTTP request, a file appearing in storage, a message in a queue, a database change, a timer.
- When the trigger fires, the provider finds or creates an execution environment, runs your function, and returns the result.
- You are billed for the milliseconds it ran and the memory it was given. Between events the cost is zero.
- If a thousand events arrive at once, a thousand copies run at once. You did not configure that.
- The catch is that each run starts with no memory of any previous run. Nothing you leave in a variable can be relied on next time.

9. The other catch is that the very first run after a pause is slower, because the environment has to be created. That is the **cold start**.

■ 43.12.2 PLAIN – a picture in your head

1. Think of a hotel meeting room you can book by the minute.
2. Ordinary servers are like renting an office all year: it is yours, your things are on the desk, and you pay in August when nobody is there.
3. Serverless is like walking to reception, being handed an empty room in seconds, using it for ninety seconds, and leaving.
4. The room is always empty when you arrive. Anything you left last time is gone.
5. Most of the time a cleaned room is already waiting and you walk straight in. That is a warm start.
6. Occasionally none is ready, and staff must prepare one while you wait. That is a cold start.
7. If two hundred of your colleagues arrive at once, reception opens two hundred rooms, and you get one bill for two hundred short bookings.

Where this comparison breaks: a hotel has a fixed number of rooms and will eventually say no. Serverless platforms feel unlimited but have concurrency quotas, and hitting one produces throttling errors rather than a polite refusal. Also, in a hotel you pay per room. Here you pay per room multiplied by its size, so a function given four times the memory costs four times as much per second, even when it does not use the memory.

■ 43.12.3 PLAIN – a worked example

1. A real AWS Lambda function in Python, triggered when an image is uploaded to S3, which records the upload in DynamoDB.

```
import os, json, boto3, urllib.parse

ddb = boto3.resource("dynamodb")
table = ddb.Table(os.environ["TABLE_NAME"])

def handler(event, context):
    for record in event["Records"]:
        bucket = record["s3"]["bucket"]["name"]
        key = urllib.parse.unquote_plus(
            record["s3"]["object"]["key"])
        size = record["s3"]["object"]["size"]
        table.put_item(Item={
            "pk": f"{bucket}/{key}",
            "size": size,
            "eventTime": record["eventTime"],
        })
    return {"statusCode": 200,
            "body": json.dumps({"ok": len(event["Records"])))}
```

2. The two lines creating ddb and table are outside the handler on purpose. Code outside the handler runs once per execution environment, not once per event, so the connection is reused across warm invocations.
3. event describes what happened. context carries the request id and the remaining time in milliseconds.
4. unquote_plus is not decoration. S3 event keys are URL-encoded, and spaces arrive as plus signs. Forgetting this is one of the most common real bugs.

5. The function returns; the environment is kept warm for a while in case another event arrives; eventually it is destroyed.
6. Now the cost. Suppose 2 million invocations a month, 200 milliseconds each, at 512 MB.
7. Compute: $2,000,000 \times 0.2 \text{ s} \times 0.5 \text{ GB} = 200,000 \text{ GB-seconds}$. At 0.0000166667 US dollars per GB-second that is 3.33 US dollars.
8. Requests: 2,000,000 at 0.20 US dollars per million is 0.40 US dollars.
9. Total about 3.73 US dollars per month, before the free allowance of one million requests and 400,000 GB-seconds, which would cover most of it.
10. A single small EC2 instance running all month costs about 30 US dollars. For this workload serverless is roughly ten times cheaper.

■ 43.12.4 PLAIN – what is really happening inside

1. A cold start has four phases. The platform first downloads your code package.
2. It then creates an isolated execution environment. On AWS this is a Firecracker micro virtual machine, which boots in roughly 125 milliseconds.
3. It then starts the language runtime: interpreter, virtual machine or process.
4. Finally it runs your initialization code, the part outside the handler, and only then calls your handler.
5. After the handler returns, the environment is frozen rather than destroyed. A later event thaws it, and only the handler runs. That is a warm start, typically a few milliseconds of overhead.
6. Environments are frozen for some minutes and then reclaimed. The exact time is not documented and has changed over the years.
7. This is why state is unreliable. A warm environment keeps its variables, so caching sometimes appears to work, which is more dangerous than if it never worked.
8. Concurrency is not threads. Each concurrent event gets its own environment, so your function does not need to be thread-safe, but it also cannot share memory with its siblings.

■ 43.12.5 TECHNICAL – the engineer's version

1. AWS Lambda launched 13 November 2014, the first mainstream function service. Google Cloud Functions followed in 2016, Azure Functions in 2016.
2. Current published limits for Lambda: memory from 128 MB to 10,240 MB in 1 MB steps; maximum execution time 15 minutes; deployment package 50 MB zipped direct upload, 250 MB unzipped, or up to 10 GB as a container image; request and response payload 6 MB synchronous, 256 KB asynchronous; /tmp from 512 MB to 10 GB.
3. CPU is not configured separately. It is allocated in proportion to memory, with roughly one full virtual CPU at about 1,769 MB. Raising memory to make a function faster is a legitimate and frequently profitable tactic.
4. Pricing, x86, at the time of writing: 0.20 US dollars per million requests and 0.0000166667 US dollars per GB-second, with an always-free tier of one million requests and 400,000 GB-seconds per month. Provisioned concurrency costs 0.0000041667 US dollars per GB-second on top.
5. Measured cold start durations at the median, from public benchmarking:

Runtime	Typical cold start
Rust	150 to 300 ms
Python 3.12	200 to 300 ms
Node.js 20	200 to 400 ms
C# .NET 8	500 to 1,500 ms

Runtime	Typical cold start
Java 17, no SnapStart	800 to 2,000 ms

6. Cold start frequency depends entirely on traffic shape. At sustained high traffic it is well under 1 per cent of invocations. At a handful of requests per hour, 30 to 50 per cent of invocations are cold.
7. Two mitigations exist. Provisioned concurrency keeps environments warm at a price. SnapStart, introduced for Java in 2022 and later extended, takes a snapshot of the initialized environment and restores from it, cutting multi-second Java starts to a few hundred milliseconds.
8. The VPC penalty is historical. Before 2019, attaching a Lambda to a VPC added 10 to 15 seconds to a cold start because an elastic network interface was created per environment. AWS re-engineered this with shared interfaces, and the penalty is now typically under 50 milliseconds. Beware of old advice.
9. When serverless is right: spiky or unpredictable traffic; event processing glue between services; scheduled jobs; low steady volume where a server would sit idle; teams with no operations capacity.
10. When serverless is a trap: sustained high throughput, where per-request pricing exceeds a reserved instance by a wide margin; workloads needing more than 15 minutes; anything needing large in-memory state or a warm cache; latency-critical paths where a cold start is unacceptable; heavy database use, because a thousand concurrent functions open a thousand connections and exhaust the database, which is what RDS Proxy exists to fix.
11. Note also the debugging cost. Local reproduction is imperfect, stack traces cross service boundaries, and understanding a failure requires distributed tracing rather than a debugger.
12. Vendor lock-in is strongest here. The function body is portable. The triggers, permissions, event shapes and surrounding services are not.

■ 43.12.6 WORDS – remember these

1. Serverless — you never see or pay for an idle machine — an execution model with per-invocation billing, automatic scaling and no capacity management.
2. Cold start — the slow first run — the latency of creating an execution environment, starting the runtime and running initialization code.
3. Warm start — reusing a paused environment — invocation of a thawed environment, adding only milliseconds of overhead.
4. Trigger — the thing that causes your code to run — an event source binding such as HTTP, object creation, queue message or schedule.
5. Concurrency limit — how many copies may run at once — an account or function quota beyond which invocations are throttled.
6. Provisioned concurrency — paying to keep some copies warm — pre-initialized environments billed continuously to eliminate cold starts.
7. Firecracker — the tiny virtual machine underneath — AWS’s open-source micro-VM monitor, booting in about 125 milliseconds, used by Lambda and Fargate.

43.13 Storage in the cloud

■ 43.13.1 PLAIN – in simple words

1. There are three kinds of cloud storage, and choosing the wrong one is a common and expensive mistake.
2. **Block storage** is a virtual hard disk. You attach it to one machine, and the machine puts a filesystem on it. It is fast and it belongs to one owner.
3. **File storage** is a shared drive. Many machines mount it at once and see the same folders. Slower, but shared.
4. **Object storage** is different in kind. There are no folders and no disk. You store a whole item under a name, and you get the whole item back by name, over the web.
5. Object storage cannot be partly edited. To change one byte you upload the whole object again.
6. In exchange it is cheap, effectively unlimited, and enormously durable.
7. Choose block for databases and operating system disks. Choose file when several machines genuinely need the same files. Choose object for uploads, backups, logs, images, video and anything a browser will download.
8. There is one more thing everyone learns painfully. Putting data in is usually free. Taking data out to the internet is not.

■ 43.13.2 PLAIN – a picture in your head

1. Block storage is a blank notebook on your desk. You can rub out one word on page 40 and write another. Only you have it.
2. File storage is a shared filing cabinet in the office. Several people open drawers at once, and there are rules about who may write.
3. Object storage is a left-luggage counter. You hand over a sealed bag and get a ticket. Later you show the ticket and get the whole bag back.
4. You cannot ask the counter to change one shirt inside the bag. You take the bag, change the shirt, and hand back a new bag.
5. The counter is astonishingly cheap and never runs out of shelves.
6. But it charges you a small fee every time you carry a bag out of the building.

Where this comparison breaks: a left-luggage counter has one shelf per bag. Object storage keeps several copies of your bag in several buildings, which is why it survives a fire and a notebook does not. Also, the counter here is reachable by anyone in the world with the right ticket, which is a security property the analogy hides and which has caused a great many public data leaks.

■ 43.13.3 PLAIN – a worked example

1. Real S3 list prices for the North Virginia region, August 2026, per gigabyte per month.

Storage class	Price per GB	Use for
Standard, first 50 TB	0.0230	Active data
Standard-IA	0.0125	Monthly access
One Zone-IA	0.0100	Recreatable data
Glacier Instant	0.0040	Quarterly access
Glacier Flexible	0.0036	Backups
Glacier Deep Archive	0.00099	Legal retention

2. Deep Archive is about 23 times cheaper than Standard. That is why lifecycle rules exist.

3. Requests cost separately: 0.005 US dollars per 1,000 PUT requests and 0.0004 per 1,000 GET requests in Standard. In Glacier Deep Archive a PUT costs 0.05 per 1,000, ten times more, because the class is designed for few, large objects.
4. Store 10 terabytes of logs. In Standard: $10,240 \text{ GB} \times 0.023 = 235.52$ US dollars per month. In Deep Archive: $10,240 \times 0.00099 = 10.14$ US dollars.
5. Now the trap. You want to download those 10 terabytes to your own datacentre.
6. Data transfer out to the internet is billed at 0.09 US dollars per gigabyte for the first 10 TB each month, after 100 GB free.
7. $10,240 \text{ GB} \times 0.09 = 921.60$ US dollars, minus the free 100 GB, so about 912 US dollars.
8. Read that again. Storing the data for a month cost 10 dollars. Reading it once cost 912. The exit costs 90 times the rent.
9. That asymmetry is the single most important commercial fact about cloud storage, and it is not an accident.

■ 43.13.4 PLAIN – what is really happening inside

1. When you PUT an object, the service splits it, computes checksums, and writes copies to storage nodes in at least three separate availability zones before it answers “done”.
2. That is why the write is slower than a local disk write and why the durability claim is so high.
3. Object names look like paths, but there are no directories. `2026/08/log.txt` is one flat key that happens to contain slashes, and the console draws fake folders from them.
4. A lifecycle policy is a background process the service runs on your bucket daily. It looks at object age and moves or deletes objects by rule.
5. Transitions are not free: each object moved costs a lifecycle transition request. Moving ten million tiny objects to Glacier can cost more than leaving them where they were.
6. Egress is metered at the edge of the provider’s network. Traffic to the internet is charged; traffic to another service inside the same region is usually free; traffic between availability zones or regions has its own smaller rate.

■ 43.13.5 TECHNICAL – the engineer’s version

1. Consistency. S3 offered eventual consistency for overwrites and deletes until 1 December 2020, when AWS made it strongly read-after-write consistent for all operations, at no extra cost and with no performance penalty. A great deal of published advice about “eventual consistency workarounds” predates that date and is now wrong.
2. Strong consistency here means a successful PUT is immediately visible to a subsequent GET or LIST from any client. It does not provide transactions across objects, and there is still no locking beyond conditional writes.
3. Performance figures: at least 3,500 PUT, COPY, POST or DELETE and 5,500 GET or HEAD requests per second per partitioned prefix, scaling by adding prefixes. Maximum object size 5 TB; maximum single PUT 5 GB; multipart upload required above that, with up to 10,000 parts.
4. Block storage on AWS is EBS: gp3 at 0.08 US dollars per GB-month with 3,000 IOPS and 125 MB/s included, extra IOPS at 0.005 each per month; io2 at 0.125 per GB-month plus 0.065 per provisioned IOPS. Snapshots cost 0.05 per GB-month standard, 0.0125 archived, and are incremental.
5. gp2, the older class, costs 0.10 per GB-month and ties performance to size at 3 IOPS per GB. gp3 is cheaper and decouples the two. Migrating gp2 volumes to gp3 is one of the easiest genuine savings in AWS.
6. File storage is EFS on AWS, Azure Files, or Filestore on Google Cloud. EFS speaks NFSv4.1 and costs several times more per gigabyte than EBS, so use it only when sharing is a real requirement.
7. Lifecycle policy, expressed in the JSON that the API actually takes:


```
{
  "Rules": [
    {
      "ID": "logs-tiering",
      "Filter": {"Prefix": "logs/"},
      "Status": "Enabled",
      "Transitions": [
        {"Days": 30, "StorageClass": "STANDARD_IA"},
        {"Days": 90, "StorageClass": "GLACIER"},
        {"Days": 365, "StorageClass": "DEEP_ARCHIVE"}
      ],
      "Expiration": {"Days": 2555}
    }
  ]
}
```

8. Minimum storage durations are billed even if you delete early: 30 days for Standard-IA and One Zone-IA, 90 days for Glacier Flexible, 180 days for Deep Archive. Standard-IA and One Zone-IA also bill a minimum object size of 128 KB, so tiering many small files can increase the bill.
9. Retrieval from Deep Archive takes up to 12 hours in standard mode and costs per gigabyte on top. It is an archive, not cold storage you can browse.
10. Egress pricing tiers for data out to the internet from US and EU regions: first 100 GB per month free across all services, then 0.09 US dollars per GB to 10 TB, 0.085 to 50 TB, 0.07 to 150 TB, and 0.05 above that. Cross-availability-zone traffic is 0.01 per GB in each direction, and NAT gateway processing adds 0.045 per GB on top of everything else.
11. Two changes have eroded the egress lock. From 5 March 2024 AWS offers free data transfer out for customers who are leaving AWS entirely, with 90 days to complete the move and a support request required; other major providers made similar announcements. And the European Union's Data Act, Regulation 2023/2854, requires switching charges including egress fees to be fully withdrawn from 12 January 2027.
12. Cloudflare R2 and several second-tier providers charge no egress at all, which is a deliberate competitive attack on this pricing model.

■ 43.13.6 WORDS - remember these

1. Object storage — whole items stored and fetched by name — a flat key-value store over HTTP, immutable per object, replicated across zones.
2. Block storage — a virtual hard disk for one machine — network-attached block device presented as raw sectors, formatted by the guest.
3. File storage — a shared network drive — a managed NFS or SMB filesystem mountable by many clients at once.
4. Storage class — how cheap and how slow — a tier trading price per gigabyte against retrieval time, minimum duration and request cost.
5. Lifecycle policy — automatic tiering and deletion by age — bucket rules that transition or expire objects on a daily background pass.
6. Egress — the charge for data leaving — per-gigabyte data transfer out to the internet, typically 0.09 US dollars per GB at the first tier.

43.14 Cloud networking

■ 43.14.1 PLAIN - in simple words

1. A VPC, virtual private cloud, is your own private network inside the provider's network. You choose the address range and the layout.
2. Inside it you create **subnets**, which are slices of that address range, each pinned to one availability zone.

3. A subnet is called **public** if its route table sends unknown traffic to an internet gateway, and **private** if it does not. That is the only difference.
4. An **internet gateway** lets traffic in and out. A **NAT gateway** lets private machines reach out to the internet while nothing can reach in.
5. **Security groups** are firewalls attached to individual machines. They allow traffic; they cannot deny it, and they remember connections.
6. **Network ACLs** are firewalls attached to whole subnets. They can allow and deny, and they remember nothing.
7. **Load balancers** sit in front of your machines and spread traffic. Layer 4 forwards connections without reading them. Layer 7 reads the request and can route by hostname or path.
8. **Private endpoints** let you reach a provider service without your traffic going over the internet at all.
9. **Peering** joins two networks. A **transit gateway** joins many, in a hub pattern, instead of one link per pair.

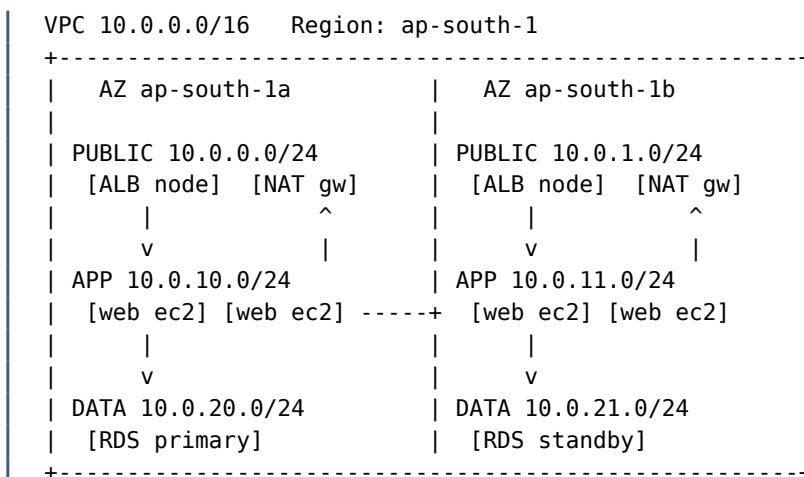
■ 43.14.2 PLAIN – a picture in your head

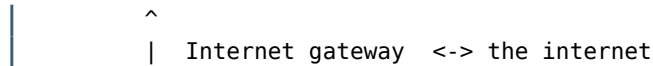
1. Think of a gated business park that you have rented and laid out.
2. The park boundary is the VPC. The address range is the numbering scheme for all the plots.
3. Subnets are the blocks of plots. Two blocks are in one part of the park and two in another, on separate power and water.
4. The main gate is the internet gateway.
5. A NAT gateway is a courier desk. Staff in the inner blocks can send parcels out through it and get replies, but nobody outside can address a parcel to them directly.
6. Security groups are the locks on each individual building's doors.
7. Network ACLs are the barriers at the entrance to each block, checking every vehicle in both directions with no memory of who came in.
8. A load balancer is the receptionist at the front, directing arrivals to whichever office is open.

Where this comparison breaks: in a real park a delivery van drives itself. Here, every packet's path is decided by a route table you wrote, and if no route matches, the packet is silently dropped. There is no lost-property office and no error message. Silence is the normal failure mode, which is why the first thing to check in any cloud networking problem is the route table.

■ 43.14.3 PLAIN – a worked example

1. A standard three-tier VPC, spread over two availability zones.





2. Address maths. $10.0.0.0/16$ gives 65,536 addresses. Each $/24$ gives 256, of which AWS reserves 5, leaving 251 usable.
3. Route table for the public subnets: $10.0.0.0/16$ local, $0.0.0.0/0$ to the internet gateway.
4. Route table for the application subnets: $10.0.0.0/16$ local, $0.0.0.0/0$ to the NAT gateway in the same availability zone.
5. Route table for the data subnets: $10.0.0.0/16$ local, and nothing else. The database cannot reach the internet at all, which is deliberate.
6. Security group chain: the load balancer group allows 443 from anywhere; the web group allows 8000 only from the load balancer's group; the database group allows 5432 only from the web group.
7. Referencing a group rather than an address range is the important habit. It keeps working when instances are replaced.
8. One NAT gateway per availability zone, not one shared, because a shared NAT gateway makes the other zone depend on the first, defeating the whole design.

■ 43.14.4 PLAIN – what is really happening inside

1. None of this is physical. There is no cable and no real router.
2. Every packet leaving an instance is intercepted by the host's virtualization hardware, which looks up your VPC's rules and rewrites the packet into an encapsulated form.
3. That encapsulated packet crosses the provider's physical network as ordinary traffic, is decapsulated at the destination host, and delivered.
4. Your route tables and security groups are entries in a distributed database that is pushed to every host that needs them.
5. On AWS this mapping work happens on the Nitro card, not on the CPU, which is why a security group costs almost nothing in performance.
6. Because rules are enforced at the source host, a denied packet never reaches the network. There is no device to log it, which is why denied traffic is invisible unless you enable flow logs.

■ 43.14.5 TECHNICAL – the engineer's version

1. Address planning uses RFC 1918 private ranges: $10.0.0.0/8$, $172.16.0.0/12$ and $192.168.0.0/16$. AWS VPC CIDR blocks may be $/16$ to $/28$. Overlapping ranges between VPCs make peering impossible later, so allocate deliberately from day one.
2. Security groups versus network ACLs, the exam question and the real one:

Property	Security group	Network ACL
Attaches to	Instance interface	Subnet
Rules	Allow only	Allow and deny
State	Stateful	Stateless
Evaluation	All rules, any match	Numbered, first match

3. **Stateful** means the return traffic of an allowed outbound connection is automatically permitted. **Stateless** means you must write both directions, including the ephemeral port range 1024 to 65535 for replies. This is the same distinction drawn in Chapter 34 for ordinary firewalls, and it is the most common source of network ACL mistakes.

4. Load balancers by layer. A network load balancer operates at layer 4, forwards TCP or UDP flows, preserves the client address, handles millions of connections per second and adds tens of microseconds. An application load balancer operates at layer 7, terminates TLS, understands HTTP, and can route by host header, path, method or header value, at the cost of about one millisecond and a more complex failure surface.
5. Private connectivity to services: gateway endpoints, which are route table entries for S3 and DynamoDB and are free; and interface endpoints, which are PrivateLink elastic network interfaces in your subnets, billed per hour and per gigabyte. Using a gateway endpoint for S3 removes both NAT gateway processing charges and the internet path.
6. VPC peering is one-to-one, non-transitive, and requires non-overlapping CIDR blocks. If A peers with B and B peers with C, A cannot reach C. That limitation is exactly why AWS Transit Gateway exists: a hub that all VPCs attach to, giving transitive routing, at a per-attachment hourly charge plus per-gigabyte processing.
7. Hybrid connectivity: site-to-site VPN over the internet using IPsec, typically 1.25 Gbps per tunnel and subject to internet variability; or Direct Connect, a dedicated physical circuit from 50 Mbps to 100 Gbps with predictable latency and lower egress rates. Azure calls the equivalent Express-Route; Google calls it Cloud Interconnect.
8. Observation: VPC Flow Logs record accepted and rejected flows to CloudWatch or S3. Reachability Analyzer traces a path through your configuration and tells you which rule blocked it, without sending a packet. `aws ec2 describe-route-tables` and `aws ec2 describe-security-groups` are the two commands most often needed at 3am.

■ 43.14.6 WORDS – remember these

1. VPC — your own private network in the provider — a logically isolated virtual network with tenant-defined CIDR, subnets and routing.
2. Subnet — a slice of that network in one zone — a CIDR range bound to a single availability zone with one associated route table.
3. Internet gateway — the way in and out — a horizontally scaled VPC component providing a route to and from the public internet.
4. NAT gateway — outbound only, for private machines — a managed network address translation service, billed hourly and per gigabyte.
5. Security group — a stateful allow-list on one machine — instance-level rules permitting traffic, with automatic return-path allowance.
6. Network ACL — a stateless filter on a whole subnet — numbered allow and deny rules evaluated in order, requiring explicit return rules.
7. Transit gateway — one hub instead of many pairwise links — a regional router providing transitive connectivity between many VPCs and on-premises networks.

43.15 Identity and access

■ 43.15.1 PLAIN – in simple words

1. Every action in a cloud account is an API call, and every API call is checked against rules before it runs.
2. The system holding those rules is identity and access management, IAM.
3. There are two kinds of actor. A **user** is a person with a long-lived password or key. A **role** is a set of permissions that anything can borrow temporarily.
4. A **policy** is a document listing which actions are allowed on which resources, and under which conditions.
5. The single most important habit is **least privilege**: grant exactly what is needed and nothing more, then review it.
6. The second most important habit is to stop using long-lived keys.
7. A long-lived key is a password that never expires, is often copied into a config file, and frequently ends up in a code repository.
8. A role gives out temporary credentials that expire in an hour. If they leak, the window is small.
9. Machines should never hold permanent keys. A server gets a role attached to it, and the credentials appear automatically and rotate themselves.

■ 43.15.2 PLAIN – a picture in your head

1. Think of a hospital again.
2. A user is a staff member with a permanent badge that opens certain doors.
3. A role is a coat hanging by the door of the operating theatre. Anyone authorized may put it on, and while wearing it they can do surgery.
4. The coat is returned. It cannot be taken home, and it stops working at the end of the shift.
5. A policy is the written list of which doors that coat opens.
6. Least privilege means the surgery coat does not also open the pharmacy safe, even though it would occasionally be convenient.
7. The commonest real-world failure is a coat that opens everything, made once during an emergency, and never taken out of circulation.

Where this comparison breaks: a coat is physical, so only one person wears it at a time. Temporary credentials can be copied and used from anywhere in the world until they expire. That is why conditions such as source address, multi-factor authentication and session duration matter as much as the permission list.

■ 43.15.3 PLAIN – a worked example

1. A policy allowing an application to read one bucket and nothing else.

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "ReadUploadsOnly",
    "Effect": "Allow",
    "Action": ["s3:GetObject", "s3:ListBucket"],
    "Resource": [
      "arn:aws:s3:::festival-uploads",
      "arn:aws:s3:::festival-uploads/*"],
    "Condition": {
      "Bool": {"aws:SecureTransport": "true"}
    }
  ]
}
```

```

    }
  }
}

```

2. Two resource lines are needed because `ListBucket` acts on the bucket and `GetObject` acts on objects inside it. Forgetting this is a daily support question.
3. The condition rejects any request that did not arrive over TLS.
4. Version is a fixed policy language date, not your version number. It must be 2012-10-17.
5. Attach this policy to a role, attach the role to the instance, and the application never sees a key.
6. Compare with the wildcard policy that people write when they are tired: `"Action": "s3:*", "Resource": "*"`. That grants delete on every bucket in the account, including the backups.

■ 43.15.4 PLAIN – what is really happening inside

1. Policy evaluation follows a fixed order, and knowing it saves hours.
2. Start from an implicit deny. Nothing is allowed by default.
3. Evaluate all applicable policies: identity policies on the caller, resource policies on the target, permission boundaries, service control policies from the organization, and session policies.
4. If any of them contains an explicit deny that matches, the answer is deny. Nothing can override an explicit deny.
5. Otherwise, if at least one allow matches and no boundary excludes it, the answer is allow.
6. Otherwise the answer is deny by default.
7. In one line: explicit deny beats allow, and allow beats the default deny.
8. Temporary credentials work through a token service. An entity calls “assume this role”, proving its own identity; the service checks the role’s trust policy; if satisfied it returns an access key, a secret key and a session token with an expiry.
9. On an EC2 instance a metadata service delivers those credentials to software on the machine and refreshes them before expiry. On Kubernetes, a service account token is exchanged for the same thing.

■ 43.15.5 TECHNICAL – the engineer’s version

1. Policy types and their interaction: identity-based policies attach to users, groups and roles; resource-based policies attach to buckets, queues and keys and can grant cross-account access without a role; permission boundaries cap the maximum a principal can be granted; service control policies cap an entire account within an organization; session policies narrow one session.
2. A principal’s effective permission is the intersection of all caps and the union of all grants, minus every explicit deny.
3. Instance metadata: IMDSv1 was a plain unauthenticated HTTP GET to 169.254.169.254. IMDSv2 requires a PUT to obtain a session token first and sets a low IP hop limit, which defeats the classic server-side request forgery path. AWS began defaulting new instances to IMDSv2 in 2024. Enforce it explicitly on older infrastructure.
4. That specific weakness produced the best-known cloud breach. In March 2019 an attacker exploited a server-side request forgery flaw in a misconfigured web application firewall at Capital One, reached the metadata service, obtained the credentials of an over-permissive role, and used them to list and copy S3 buckets containing roughly 106 million customer records. Regulators fined the bank 80 million US dollars in 2020 and a class action settled at 190 million.
5. Note that no AWS component failed. Every step used a documented feature. The failure was a firewall misconfiguration plus a role with far more S3 access than the application needed.
6. The two dominant causes of cloud compromise remain over-permissive policies and leaked long-lived keys. Keys leak through public repositories, container images, CI logs, browser bundles

and screenshots. Automated scanners find a newly committed AWS key in public GitHub within minutes.

7. Mitigations in order of value: use roles and short-lived credentials everywhere; federate human access through single sign-on so no human has an access key at all; enable multi-factor authentication on every account with console access; turn on GitHub push protection and secret scanning; set an organization-wide service control policy denying dangerous actions outright.
8. Right-sizing tools: IAM Access Analyzer generates a least-privilege policy from CloudTrail history; last-accessed data shows which services a principal has actually used; Access Advisor exposes the same for roles.
9. Naming, for cross-provider fluency. AWS uses roles and JSON policies. Azure uses Entra ID principals with role-based access control, where a role assignment binds a role definition to a scope. Google Cloud binds members to predefined or custom roles at a resource in the hierarchy, and its default Compute Engine service account is famously over-permissive and should be replaced.
10. Diagnostic commands: `aws sts get-caller-identity` to learn who you are; `aws iam simulate-principal-policy` to test a decision before deploying it; CloudTrail event history to find the exact denied call and its reason.

■ 43.15.6 WORDS – remember these

1. IAM — the rules about who may do what — identity and access management, evaluated on every API call against identity and resource policies.
2. Role — a borrowable set of permissions — an identity with no long-term credentials, assumed to obtain time-limited session credentials.
3. Policy — the document listing allowed actions — a JSON statement set of effect, action, resource and condition.
4. Least privilege — grant only what is needed — the principle of minimizing granted permissions to those actually exercised.
5. Explicit deny — a refusal nothing can override — the highest-precedence outcome in policy evaluation, above any allow.
6. Instance metadata service — where a machine gets its own credentials — the link-local endpoint at 169.254.169.254, hardened as IMDSv2.
7. Service control policy — an account-wide ceiling — an organization policy limiting the maximum permissions available in member accounts.

43.16 What cloud costs and how bills explode

■ 43.16.1 PLAIN – in simple words

1. A cloud bill is not one price. It is dozens of meters running at once.
2. The usual dimensions are: time a machine existed, storage held per month, number of requests, gigabytes moved, and per-hour charges for things that feel free, such as addresses and gateways.
3. Three ways to pay for compute. **On demand** is the list price, no commitment. **Reserved**, or savings plans, means promising one or three years of spending for a discount of roughly 30 to 72 per cent. **Spot** means using spare capacity at up to 90 per cent off, on the condition it can be taken away with two minutes' notice.
4. Bills explode for boring reasons, almost never exciting ones.
5. Something was left running. Somebody chose a machine ten times bigger than needed. Logs were kept forever. Data crossed a boundary that charges.
6. The free tier is genuinely useful and genuinely a trap: it hides which resources cost money until the day it expires.
7. The honest headline: at steady, predictable load, owning hardware is usually cheaper than renting it. Cloud wins on speed, elasticity and not needing a team, not on unit price.

■ 43.16.2 PLAIN – a picture in your head

1. Think of a mobile phone bill from before unlimited plans.
2. The handset rental is the machine. The data pack is the storage. Calls are requests. Roaming is egress.
3. Everyone understood the handset charge. Everyone was shocked by roaming.
4. Roaming had three properties that made it dangerous: it was per unit, it was invisible while it happened, and it was triggered by crossing a boundary nobody was thinking about.
5. Cloud egress and cross-zone traffic are exactly that. The boundary is invisible in your code and expensive in your bill.
6. And there is one extra hazard the phone bill did not have: your program can generate charges by itself, all night, with nobody watching.

Where this comparison breaks: a phone bill has a ceiling because a human can only talk so much. A misconfigured job can generate a five-figure bill in hours, and there is no hard spending limit you can set on the major clouds. Budgets alert you; they do not stop you.

■ 43.16.3 PLAIN – a worked example

1. A real monthly bill for a modest production application in North Virginia, using the list prices quoted earlier in this chapter. Two web servers across two zones, a managed database with a standby, a load balancer, storage and 500 GB of traffic to users.

Item	Calculation	USD
2 x t3.medium, 730 h	2x730x0.0416	60.74
2 x 30 GB gp3 root	60x0.08	4.80
RDS db.t4g.medium, MAZ	730x0.129	94.17
RDS storage 100 GB gp3	100x0.115	11.50
ALB, base + ~8 LCU	730x(0.0225+0.064)	63.15
NAT gateway, 1 zone	730x0.045	32.85
NAT data, 200 GB	200x0.045	9.00
S3 200 GB Standard	200x0.023	4.60

Item	Calculation	USD
Egress 500 GB, 100 free	400x0.09	36.00
Route 53 zone + queries	0.50 + ~0.40	0.90
Secrets Manager, 4	4x0.40	1.60
CloudWatch logs 20 GB	20x0.50	10.00
Total		329.31

- Now read the bill rather than the total. The two web servers, the thing everybody thinks of as “the cost”, are 18 per cent of it.
- The database costs more than the servers. The load balancer costs as much as the servers. The NAT gateway costs 42 dollars to forward packets.
- Egress of half a terabyte, which is a small website, costs more than the object storage holding everything.
- Savings available without changing the architecture: a one-year compute savings plan takes about 30 per cent off the EC2 and RDS lines, roughly 47 dollars. A gateway endpoint for S3 removes much of the NAT data charge. Log retention set to 14 days cuts CloudWatch.
- For comparison, a dedicated server of similar capability from a low-cost European host costs about 40 to 70 US dollars per month including generous traffic, but with no managed database, no standby, no load balancer and no support.

■ 43.16.4 PLAIN – what is really happening inside

- Every service writes usage records continuously, usually per hour, sometimes per second, into a billing pipeline.
- Those records land in a detailed usage report with one row per resource, per usage type, per hour. The monthly invoice is a summary of tens of millions of rows.
- This is why bills are hard to read: the invoice groups by service, but costs are caused by resources and by teams.
- Tags are the only mechanism that connects the two. An untagged resource is effectively anonymous spending.
- Charges accrue whether or not anything useful happens. A stopped EC2 instance still bills its EBS volume, its snapshots and its allocated public address.
- Since February 2024 AWS charges for every public IPv4 address, in use or not, at 0.005 US dollars per hour, about 3.60 per month each. Many organizations discovered hundreds of forgotten addresses.

■ 43.16.5 TECHNICAL – the engineer’s version

- Commitment mechanisms: Reserved Instances lock instance family and region; Savings Plans commit to a dollar per hour of spend and are more flexible. Discounts run roughly 30 to 40 per cent for one year with no upfront payment, and up to 72 per cent for three years paid fully in advance.
- Spot instances draw on spare capacity at typically 70 to 90 per cent off, with a two-minute interruption notice. Correct uses are stateless web tiers behind a load balancer, batch processing, continuous integration runners and fault-tolerant analytics. Incorrect uses are databases and anything holding state a customer cares about.
- Free tier changed materially on 15 July 2025. Accounts created from that date receive 100 US dollars of credit at sign-up and up to 100 more for completing activities, on a plan that ends after six months or when credits run out. The old twelve-month free tier applies only to accounts cre-

ated before that date. Always-free allowances, such as one million Lambda requests per month, continue under both schemes.

4. The classic explosion patterns, in the order they actually occur: a development environment left running over a holiday; log retention set to never expire; a NAT gateway processing data that should have used an endpoint; cross-availability-zone chatter between services that could have been zone-aware; oversized instances chosen from a template; unattached EBS volumes and old snapshots; and a recursive trigger, where a function writes to the bucket that triggers it.
5. **FinOps** is the discipline that grew up around this. The FinOps Foundation was formed in 2019 and moved under the Linux Foundation in 2020. Its model is three iterative phases: inform, meaning visibility and allocation; optimize, meaning rightsizing and commitments; and operate, meaning continuous governance.
6. Practical controls, in order of return on effort: mandatory cost allocation tags enforced by policy; AWS Budgets with alerts at 50, 80 and 100 per cent; automatic shutdown schedules for non-production; Compute Optimizer or equivalent for rightsizing; S3 lifecycle rules and log retention limits; gateway endpoints for S3 and DynamoDB; and one named person who reviews the top ten line items every month.
7. Now the honest comparison, which the industry avoided for a decade.
8. A single dedicated server bought outright, amortized over five years, is typically two to five times cheaper per unit of compute than the equivalent on-demand cloud instance, before staff costs.
9. The best-documented case is 37signals, the company behind Basecamp. It spent 3,201,564 US dollars on cloud in 2022, moved seven applications off AWS during 2023 onto about 700,000 US dollars of Dell hardware, and reported savings of nearly 2 million US dollars in the first full year, with total projected savings above 10 million over five years. It then moved roughly 10 petabytes off S3 as well.
10. Dropbox is the other well-known case. It moved the bulk of its storage off AWS onto its own infrastructure, a project called Magic Pocket, and its 2018 filing reported 74.6 million US dollars of infrastructure savings across 2016 and 2017.
11. Both are the same shape: large, steady, predictable workloads, with strong in-house engineering, and a bill dominated by storage and bandwidth.
12. When cloud is genuinely the better economic choice: unpredictable or seasonal load; early-stage products where the requirement is unknown; small teams with nobody to run hardware; workloads needing global reach quickly; and anything where time to market dominates unit cost.
13. When owning is better: steady load at scale; heavy egress; large storage volumes; regulatory requirements that push to dedicated hardware anyway; and organizations that already employ infrastructure staff.
14. The reasonable middle position, and the one most large companies have reached by 2026, is a split: steady baseline on owned or long-committed capacity, burst and experimentation in the cloud.

■ 43.16.6 WORDS – remember these

1. On-demand pricing — full price, no promises — per-second or per-hour billing with no commitment and no discount.
2. Reserved and savings plans — a discount for a promise — one or three year commitments giving roughly 30 to 72 per cent off.
3. Spot — very cheap, can be taken away — interruptible spare capacity at 70 to 90 per cent off with a two-minute notice.
4. Egress — the charge for data leaving — per-gigabyte data transfer out, the most common source of bill surprise.
5. Idle spend — paying for what nobody uses — unattached volumes, stopped instances still holding storage, unused addresses and forgotten environments.
6. FinOps — treating cloud spend as an engineering problem — the inform, optimize and operate practice defined by the FinOps Foundation from 2019.
7. Repatriation — moving workloads back off cloud — relocating steady, high volume workloads to owned hardware for cost reasons.

43.17 DevOps, infrastructure as code and the modern workflow

■ 43.17.1 PLAIN – in simple words

1. For decades, two groups fought. Developers were paid to ship changes. Operators were paid to keep things stable. Change is the enemy of stability, so they were structurally opposed.
2. **DevOps** is the idea of removing that opposition: one team, shared goals, shared responsibility for both shipping and running.
3. It is a way of organizing people. It is not a tool, not a job title and not a department, although the industry turned it into all three.
4. The technical half is **infrastructure as code**: instead of clicking in a web console, you write a file describing what should exist, and a tool makes reality match it.
5. That file goes into version control, gets reviewed, and can rebuild the entire environment from nothing.
6. **Immutable infrastructure** takes it further: never patch a running machine. Build a new one from a new image and throw the old one away.
7. **GitOps** takes it further still: the repository is the only source of truth, and an agent inside the cluster continuously pulls from it and corrects drift.
8. **Observability** is being able to answer questions about a running system that you did not think to ask in advance. It needs logs, metrics and traces together, because each answers a different question.
9. **Site reliability engineering** is the discipline of deciding, numerically, how reliable a service must be, and spending the remaining unreliability deliberately on shipping features.

■ 43.17.2 PLAIN – a picture in your head

1. Think of two ways to furnish a flat.
2. Way one: over three years you buy things, move them, and drill holes. It looks fine. Now furnish an identical flat next door exactly the same way. You cannot. Nobody wrote anything down.
3. Way two: you keep a plan on one sheet. Every change is drawn on the plan first, then done. Furnishing the identical flat takes an afternoon.
4. Infrastructure as code is the second way.
5. Immutable infrastructure is the rule that you never repaint a wall. You build a new wall to the new plan and remove the old one.
6. GitOps adds a caretaker who compares the flat to the plan every few minutes and undoes anything that does not match, including your own quick fix.

Where this comparison breaks: a flat has no state. A database has state, and you cannot throw it away and build a new one. The immutable rule applies cleanly to servers that hold no data, and needs careful exceptions for anything that does. That distinction, stateless versus stateful, is the whole reason people speak of cattle and pets.

■ 43.17.3 PLAIN – a worked example

1. The same load balancer and instance, written in Terraform.

```
resource "aws_security_group" "web" {
  name     = "festival-web"
  vpc_id   = aws_vpc.main.id

  ingress {
    from_port     = 8000
    to_port       = 8000
    protocol      = "tcp"
    security_groups = [aws_security_group.alb.id]
  }
  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_instance" "web" {
  count          = 2
  ami            = var.app_ami_id
  instance_type  = "t3.medium"
  subnet_id      = aws_subnet.app[count.index].id
  vpc_security_group_ids = [aws_security_group.web.id]
  iam_instance_profile = aws_iam_instance_profile.web.name

  tags = {
    Name = "festival-web-${count.index}"
    Env  = "prod"
    Cost = "festival"
  }
}
```

2. `aws_security_group.alb.id` is a reference, not a copied value. Terraform builds a dependency graph from those references and orders creation itself.

3. `count = 2` creates two instances. Changing it to four is a one-character edit and a reviewable change.
4. `var.app_ami_id` points at a machine image built in advance. Changing that variable replaces both instances, which is immutable infrastructure in practice.
5. The tags are not decoration. They are how the monthly bill gets attributed.
6. The workflow:

```
terraform init
terraform plan -out=tf.plan
terraform apply tf.plan
terraform state list
terraform destroy
```

7. `plan` is the important command. It prints exactly what will be created, changed or destroyed, before anything happens. Reviewing a plan is the equivalent of reviewing a diff.

■ 43.17.4 PLAIN – what is really happening inside

1. Terraform keeps a **state file**: a record of which real resources correspond to which blocks in your code.
2. On `plan` it does three reads: your configuration, the state file, and the actual cloud API. It then computes the difference.
3. That is why the state file matters so much. Lose it and Terraform no longer knows it owns anything, and will try to create everything again.
4. In any team the state lives remotely, in S3 or an equivalent, with locking so two engineers cannot apply at once.
5. GitOps inverts the direction of control. Instead of a pipeline pushing changes into the cluster, an agent inside the cluster pulls the repository and applies it continuously.
6. That means the cluster needs no inbound credentials from the pipeline, and any manual change made by hand is reverted within minutes.
7. Observability data flows separately. Applications emit structured logs, numeric metrics and trace spans; agents collect them; backends store and index them; dashboards and alerts read them.

■ 43.17.5 TECHNICAL – the engineer's version

1. Origins. The word DevOps comes from DevOpsDays, organized by Patrick Debois in Ghent, Belgium, in October 2009. The cultural model is often summarized as CALMS: culture, automation, lean, measurement, sharing.
2. Tooling lineage for infrastructure as code: CFEngine 1993, Puppet 2005, Chef 2009, SaltStack 2011, Ansible 2012. Those are configuration management, which converges the inside of an existing machine.
3. Provisioning tools create the machines and the surrounding cloud resources: AWS CloudFormation from February 2011, Terraform from HashiCorp in 2014, Azure Resource Manager templates and the friendlier Bicep, and Pulumi from 2018, which uses ordinary programming languages instead of a domain-specific one.
4. A licensing event worth knowing: HashiCorp changed Terraform's licence from Mozilla Public License to the Business Source License in August 2023. The community forked it as OpenTofu, adopted by the Linux Foundation in September 2023, with version 1.0 in January 2024. If you are choosing today, this is a live consideration.

Tool	Language	Scope
CloudFormation	YAML or JSON	AWS only
Terraform, OpenTofu	HCL	Multi-provider
Pulumi	Python, TS, Go	Multi-provider
Ansible	YAML	Config management

5. Immutable infrastructure practice: bake an image with Packer or an equivalent, version it, deploy by replacing instances, and forbid interactive login to production. The phrase “cattle, not pets” comes from a 2012 talk by Bill Baker of Microsoft, popularized by Randy Bias.
6. GitOps was named by Alexis Richardson of Weaveworks in 2017. The CNCF OpenGitOps project states four principles: the desired state is declarative; it is versioned and immutable; agents pull it automatically; and agents continuously reconcile actual to desired. Argo CD and Flux are the two main implementations.
7. Observability, the three signals, and why you need all three. Metrics are cheap numbers over time and tell you that something is wrong. Logs are detailed records of individual events and tell you what happened in one case. Traces follow a single request across service boundaries and tell you where the time went and which service failed.
8. Metrics alone cannot tell you which user. Logs alone cannot show a system-wide pattern or a cross-service path. Traces alone are usually sampled and cannot be the record of truth. The three are complementary, not alternatives.
9. OpenTelemetry, formed in 2019 from the merger of OpenTracing and OpenCensus under the CNCF, is now the vendor-neutral standard for emitting all three. The wire format is OTLP.
10. Site reliability engineering began at Google in 2003 under Ben Treynor Sloss and was published as the book Site Reliability Engineering in 2016. Its core vocabulary:
11. An **SLI**, service level indicator, is a measured number, such as the proportion of requests served successfully in under 300 milliseconds.
12. An **SLO**, service level objective, is your internal target for that indicator, such as 99.9 per cent over 28 days.
13. An **SLA** is the external contract, always weaker than the objective, with financial consequences.
14. An **error budget** is the allowed failure: 100 per cent minus the objective. At 99.9 per cent over 30 days it is about 43 minutes of failure.
15. The budget is the point. If it is unspent, you are being too cautious and should ship faster. If it is exhausted, feature work stops until reliability is restored. It converts an argument about risk into arithmetic.
16. Blameless postmortems, formalized at Etsy by John Allspaw and colleagues from around 2012, assume that people act reasonably given the information they had, and look for the systemic conditions that made the error possible. Naming an individual ends learning, because the next person hides the near miss.
17. On-call practice that is defensible: a rota no thinner than one week in six, documented runbooks for every alert, alerts that are actionable or deleted, compensation or time off for nights worked, and a rule that a page nobody acted on gets removed rather than ignored.

■ 43.17.6 WORDS – remember these

1. DevOps — one team owning both shipping and running — a cultural model removing the structural conflict between change and stability, from 2009.
2. Infrastructure as code — the environment written down as a file — declarative definitions applied by a tool, versioned and reviewed like source code.

3. State file — the tool's record of what it owns — Terraform's mapping between configuration blocks and real resource identifiers, stored remotely with locking.
4. Immutable infrastructure — replace, never patch — deploying new pre-built images instead of modifying running machines.
5. GitOps — the repository is the truth and an agent enforces it — declarative state pulled and continuously reconciled from version control.
6. SLI, SLO, SLA — the measurement, the internal target, the external contract — the three levels of reliability commitment in site reliability engineering.
7. Error budget — the failure you are allowed to spend — 100 per cent minus the objective, used to arbitrate between shipping and stabilizing.
8. Observability — being able to ask new questions of a live system — the combination of metrics, logs and traces, standardized by OpenTelemetry.

43.18 Microservices versus monolith, decided honestly

■ 43.18.1 PLAIN - in simple words

1. A **monolith** is one program containing all your features, built and deployed as one unit.
2. **Microservices** means splitting those features into many small programs that talk over the network and deploy independently.
3. The genuine benefit is independence. Teams ship without coordinating, and one service can be rewritten or scaled without touching the rest.
4. The genuine cost is that every internal function call you replaced with a network call can now be slow, fail, arrive twice or vanish.
5. A method call cannot half-succeed. A network call can.
6. So the question is never "which is better". It is "do I have enough teams for the independence to be worth the failure handling".
7. For a team of five, the answer is almost always no.
8. The recommendation, which is now mainstream rather than contrarian: start with a **modular monolith**. One deployable unit, with strict internal boundaries and no reaching across them.
9. Then, if a module genuinely needs to scale differently, be written in another language, or be owned by another team, extract that one module.
10. You get the mandate's discipline about boundaries without paying the network bill until you must.

■ 43.18.2 PLAIN - a picture in your head

1. Picture a restaurant kitchen with one large team in one room.
2. Anyone can shout to anyone. Coordination is instant and free. The room does not scale past a certain number of cooks.
3. Now picture five separate kitchens in five buildings, each producing one part of the meal, connected by couriers.
4. Each kitchen can be rebuilt, restaffed or expanded without closing the others.
5. But every request is now a courier trip. Couriers get stuck in traffic. Sometimes a courier arrives and the dish has already been sent again.
6. A modular monolith is one building with five clearly separated stations, fixed hatches between them, and a rule that nobody walks into another station's area.
7. If the dessert station later needs its own building, it already has a defined hatch to replace with a courier route.

Where this comparison breaks: kitchens are physically fixed, so the analogy suggests the split is

permanent and expensive. In software the split is reversible in principle, and several well-known organizations have merged services back. It also understates the failure modes. A courier who never arrives is obvious. A network call that succeeded but whose reply was lost looks exactly like one that failed, and that ambiguity is the hardest problem in the whole subject.

■ 43.18.3 PLAIN – a worked example

1. One user action, “place order”, in both designs.
2. Monolith: one process calls `reserve_stock()`, `charge_card()` and `create_shipment()` inside a single database transaction. If the card fails, the transaction rolls back and no stock was reserved. Total time about 40 milliseconds. The failure model is one exception.
3. Microservices: the order service calls the inventory service, then the payment service, then the shipping service, each over the network.
4. There is no shared transaction. If payment fails after stock was reserved, the reservation must be actively undone by a **compensating action**.
5. That pattern is called a **saga**: a sequence of local transactions where each step has a defined undo.
6. Now the hard case. The order service calls payment and the reply is lost. Was the card charged? Unknown.
7. If it retries blindly, it may charge twice. So payment must accept an **idempotency key** and treat a repeat of the same key as the same charge.
8. Timing: three network calls at roughly 2 milliseconds each, plus serialization at both ends, plus retries, gives something like 15 to 40 milliseconds instead of 40 microseconds of function calls.

Aspect	Monolith	Microservices
Call cost	Nanoseconds	Milliseconds
Failure model	Exception	Timeout, dupe, loss
Consistency	Transaction	Saga, eventual
Deploy unit	One	Many, independent

■ 43.18.4 PLAIN – what is really happening inside

1. Once a request crosses services, no single log file contains the story.
2. **Distributed tracing** fixes this. The first service creates a trace identifier and passes it to everything it calls, in an HTTP header.
3. Each service records a **span**: its own start time, duration, name, and the identifier of the span that caused it.
4. A backend collects the spans and reassembles them into one tree by trace identifier, showing exactly where the time went.
5. Without tracing, debugging a slow request across eight services means guessing and correlating timestamps by hand.
6. Eventual consistency means different services briefly hold different answers. The order exists but the shipment does not yet. Your user interface must be designed for that, not surprised by it.

■ 43.18.5 TECHNICAL – the engineer’s version

1. The **fallacies of distributed computing** are the checklist. L. Peter Deutsch wrote seven at Sun Microsystems around 1994; James Gosling added the eighth. Each is something engineers assume without noticing.
2. The network is reliable. Latency is zero. Bandwidth is infinite. The network is secure. Topology does not change. There is one administrator. Transport cost is zero. The network is homogeneous.

3. Every one is false, and every microservice architecture that ignores one will fail in exactly the way that fallacy predicts.
4. Distributed tracing derives from Google's Dapper paper, published in 2010. The modern interoperability standard is W3C Trace Context, a Recommendation since 2021, which defines the `traceparent` and `tracestate` HTTP headers.
5. Sagas come from a 1987 paper by Hector Garcia-Molina and Kenneth Salem at Princeton, originally about long-lived database transactions. Two coordination styles exist: choreography, where services react to each other's events, and orchestration, where one coordinator drives the sequence and knows the undo steps. Choreography is looser and harder to debug; orchestration is clearer and centralizes the logic.
6. The consistency background is the CAP theorem: conjectured by Eric Brewer in 2000 and proved by Seth Gilbert and Nancy Lynch in 2002. Under a network partition you must choose between consistency and availability. Most business systems choose availability and reconcile afterwards.
7. Required infrastructure before splitting, and this list is the real cost: service discovery, centralized configuration, distributed tracing, correlated structured logging, per-service alerting and ownership routing, contract testing between services, an API gateway, retries with exponential backoff and jitter, circuit breakers, idempotency keys, and a deployment pipeline per service.
8. If your organization cannot build or buy that list, splitting will make reliability worse, not better. This is the Bezos mandate lesson restated: Amazon built all of it, and that is why the structure worked.
9. Published reversals are instructive. In March 2023 an Amazon Prime Video team wrote publicly that moving their audio and video monitoring service from a distributed serverless design to a single scaled process reduced infrastructure cost by over 90 per cent. Shopify and Stack Overflow are long-standing examples of very large systems run as well-structured monoliths.
10. Martin Fowler's 2015 article *MonolithFirst* argues that almost every successful microservice system he had seen started as a monolith that was later split, and almost every system built as microservices from scratch ran into serious trouble. That remains the best short statement of the position.
11. Where experts disagree: proponents argue that organizational independence is worth the cost from a surprisingly small size, especially with many teams and high turnover. Critics argue the boundaries chosen before you understand the domain are almost always wrong, and wrong boundaries are far more expensive across a network than inside one process. The second view has gained ground since about 2022.
12. The defensible default for a small team: one deployable unit; modules with explicit interfaces; no module reading another module's tables; a single database with clear schema ownership; and extraction only when a specific, named pressure demands it.

■ 43.18.6 WORDS – remember these

1. Monolith — one program with everything in it — a single deployable unit containing all business capabilities.
2. Modular monolith — one deployable unit with strict internal walls — enforced module boundaries and no cross-module data access, without network calls.
3. Microservices — many small independently deployed programs — services owned by separate teams communicating only over the network.
4. Distributed monolith — the worst of both — services that must be deployed together, paying network costs without gaining independence.
5. Saga — a sequence of steps with defined undos — a long-running transaction composed of local transactions plus compensating actions, Garcia-Molina and Salem 1987.
6. Idempotency key — a token making a repeat safe — a client-supplied identifier letting a server recognize and collapse duplicate requests.
7. Distributed tracing — following one request across services — correlated spans under one trace identifier, propagated via W3C Trace Context headers.
8. Fallacies of distributed computing — the eight false assumptions — Deutsch's seven from about 1994 plus Gosling's eighth.

43.98 Common wrong ideas

1. Wrong: the cloud is just someone else's computer, so it is the same thing as renting a server. Right: the rented computer is the smallest part. What you are buying is self-service provisioning in seconds, elasticity, a metered bill, and roughly 200 managed services you did not have to build. A rented server with a two-week ticket queue fails the NIST definition on two of its five required characteristics.
2. Wrong: serverless means there are no servers. Right: there are many servers. You do not choose, name, patch or pay for them while idle. You still hit memory limits, execution timeouts, concurrency quotas and cold starts, all of which are properties of real machines.
3. Wrong: cloud is always cheaper. Right: cloud is cheaper for spiky, unpredictable or small workloads, and usually more expensive per unit for steady large ones. 37signals reported saving nearly 2 million US dollars a year by leaving, and Dropbox reported 74.6 million over two years. Both had large, steady, predictable workloads and their own engineers.
4. Wrong: multi-cloud avoids lock-in. Right: it usually multiplies it. You now depend on the intersection of features from two providers, need staff expert in both, pay egress between them, and lose the deep managed services that made the cloud worth using. Genuine portability comes from boundaries in your own code, not from a second account.
5. Wrong: a container is a lightweight virtual machine. Right: a container is a process on the host kernel, isolated by namespaces and limited by cgroups. There is no second operating system and no hardware boundary. That is why it starts in milliseconds and why a kernel flaw crosses it.
6. Wrong: deploying across availability zones protects me from an outage. Right: it protects you from one building. It does not protect you from a regional control plane failure, which is what actually happens most often. The AWS event of 20 October 2025 affected an entire region for more than fifteen hours while the individual zones kept their power.
7. Wrong: Kubernetes is the modern way to run applications, so we should use it. Right: Kubernetes solves problems that appear with many services, many teams and real bin-packing pressure. Below that, it is a permanent tax in upgrades, networking, certificates and new failure modes. Most teams are better served by a managed container service.

8. Wrong: the provider is responsible for security. Right: the provider secures the cloud; you secure what you put in it. Access control and data classification never transfer, in any service model. Nearly every publicized cloud breach has been a customer misconfiguration, not a provider failure.
9. Wrong: AWS was built to sell Amazon's spare Christmas capacity. Right: AWS was a deliberate new business, proposed in writing in 2003 with its own staffing plan, built on separate hardware. Amazon's Chief Technology Officer publicly denied the spare capacity story in 2011, and retail's peak coincides with customers' peak, so the arrangement could never have worked.
10. Wrong: microservices make systems more reliable. Right: they make failure partial rather than total, which is only an improvement if you have built tracing, retries, idempotency, circuit breakers and ownership routing. Split without those and you get a distributed monolith: all the network failure modes with none of the independence.

43.99 Chapter summary in 20 lines

1. The cloud is other people's computers in other people's buildings, rented by the second and reached over the internet; the mystique is marketing.
2. What genuinely changed is self-service in seconds, elasticity, metered billing and somebody else owning the hardware; what did not change is that physical machines still fail.
3. NIST SP 800-145, from September 2011, gives the five-part test: self-service, network access, pooling, elasticity and measurement.
4. A datacentre is racks measured in 44.45 millimetre units, top-of-rack switches in a spine-leaf fabric, dual power through UPS and generators, and hot and cold aisle cooling; efficiency is PUE, globally stuck near 1.54 while hyperscalers reach 1.09 to 1.15.
5. A region is a place, an availability zone is an independent failure domain inside it, and an edge location is a cache; deploy across zones always, and treat multi-region as a large project rather than a checkbox.
6. India has AWS Mumbai from 2016 and Hyderabad from 2022, Azure's three regions from 2015 plus India South Central from August 2026, and Google Mumbai from 2017 and Delhi NCR from 2021.
7. Virtualization made renting possible: a hypervisor slices one machine into many, made cheap by Intel VT-x in 2005 and AMD-V in 2006, and made nearly free by offload hardware such as AWS Nitro from 2017.
8. A container is a process with namespaces and cgroups on a shared kernel, not a small virtual machine; the idea runs back through LXC, Solaris Zones, FreeBSD jails and chroot in 1979, and Docker in 2013 supplied the packaging.
9. Kubernetes, first committed on 6 June 2014 and descended from Google's Borg, works by declarative reconciliation: state the desired end and let control loops close the gap forever.
10. Most teams do not need Kubernetes, and saying so is not contrarian; it is what the cost of upgrades, networking and new failure modes implies.
11. Around 2002 Jeff Bezos mandated that all teams expose data and functionality only through service interfaces, with no back doors, and that every interface be designed to be externalizable, on pain of dismissal.
12. That mandate forced Amazon to build service discovery, quotas and throttling, pager routing, monitoring merged with quality assurance and standard sandboxes, which is precisely the toolkit of a cloud platform.
13. It also bought latency, partial failure and operational cost; most companies copied the structure without the discipline and got a distributed monolith.

14. AWS itself came from a 2003 paper by Chris Pinkham and Benjamin Black and a parallel business proposal by Andy Jassy; SQS launched in beta in November 2004, S3 on 14 March 2006 and EC2 on 25 August 2006 at ten cents an hour.
15. It was never built from spare retail capacity, and AWS reported 128.7 billion US dollars of net sales for 2025 against Azure's 100 billion for fiscal 2026.
16. The service models differ only in where the line falls between what you manage and what the provider manages, and the security version of that line is the shared responsibility model, in which access control is always yours.
17. Serverless means per-invocation billing and no idle cost, at the price of cold starts of 200 milliseconds to two seconds, statelessness, a fifteen minute ceiling and heavy lock-in.
18. Storage splits into object, block and file; the decisive commercial fact is that storing ten terabytes costs about ten dollars a month in deep archive while reading it out once costs about nine hundred.
19. Cloud networking is a VPC of subnets and route tables, with stateful security groups on machines and stateless network ACLs on subnets, and identity is governed by policies where an explicit deny always wins.
20. Bills explode from idle resources, oversized instances, egress and cross-zone chatter rather than from compute; own the steady baseline, rent the burst, and start a small system as a modular monolith.

Security

44.0 What this chapter gives you

1. You will be able to say what security actually means, using the five properties professionals name, and build a threat model in plain words.
2. You will be able to explain what a hash is, why MD5 and SHA-1 are dead, and read real hash values computed in front of you.
3. You will be able to store passwords correctly, choose Argon2id parameters that a working engineer would accept, and explain salt versus pepper.
4. You will be able to describe symmetric and asymmetric encryption, name the modes that are safe in 2026, and say why nonce reuse destroys everything.
5. You will be able to explain post-quantum cryptography, the standards NIST approved on 13 August 2024, and why “harvest now, decrypt later” matters.
6. You will be able to describe every common web vulnerability class, the flawed pattern behind it, and the fix that removes it by construction.
7. You will be able to explain memory safety, why C allows the bugs it does, and what the 2023 to 2025 government guidance actually asks for.
8. You will be able to name the realistic ways machines get infected, recognize social engineering, and run a first hour of incident response calmly.
9. You will read this as a defender. The chapter explains how attacks work so that you can stop them, and contains no working attack code by design.
10. You will build on two earlier chapters rather than repeat them: Chapter 32 covered TLS and certificates, and Chapter 42 covered SSH keys and tokens.

44.1 What security actually means

■ 44.1.1 PLAIN – in simple words

1. Security is not one thing. It is a small set of promises about your data.
2. Promise one: only the people who should see it can see it. That is **confidentiality**.
3. Promise two: nobody can change it without you noticing. That is **integrity**.
4. Promise three: when you need it, it is there. That is **availability**.
5. Those three together have a nickname in the trade: the CIA triad. It has nothing to do with any spy agency. The letters are just the three words.
6. Two more promises matter and are often left out.
7. Promise four: the thing claiming to be Ravi really is Ravi. That is **authenticity**.
8. Promise five: once Ravi signed something, he cannot later say he did not. That is **non-repudiation**.
9. Notice that these promises can fight each other.

10. Locking a file with a key nobody has protects confidentiality perfectly and destroys availability completely.
11. So security is always a set of trade-offs, never a single maximum.
12. And this is the sentence to carry with you: security is a property of a whole system in a particular situation. It is not a product you buy.
13. A very strong lock on a door with a glass panel next to it is not security. It is a strong lock.

■ 44.1.2 PLAIN - a picture in your head

1. Think of a small jewellery shop on a busy street.
2. Confidentiality is the curtain in the back room, so passers-by cannot see the safe being opened.
3. Integrity is the sealed ledger, so nobody can quietly add a line saying they already paid.
4. Availability is the shop being open at ten in the morning, as promised.
5. Authenticity is checking the ID of the man who says he is from the insurance company.
6. Non-repudiation is the signed receipt the customer cannot later deny.
7. Now do the thing a security professional does. Ask three questions.
8. What am I protecting? The gold, the ledger, the customer list, the reputation.
9. From whom? A casual thief, a professional gang, a dishonest employee, a competitor, a government.
10. What happens if I fail? Money lost, customers gone, licence revoked, somebody physically harmed.
11. Those three questions, written down, are a **threat model**.

Where this comparison breaks: a shop thief must physically be on that street. A network attacker can be in another country, can try a million shops per hour, and can automate everything. Scale is the difference that changes everything. A lock that keeps out ninety-nine of a hundred thieves is a good lock in a street and a broken lock on the internet, because the internet will send you ten million thieves per day and only needs one to succeed.

■ 44.1.3 PLAIN - a worked example

1. Let us threat model something real: the reader's own laptop and its GitHub account, from the running example used in this book.
2. Assets, written plainly.

Asset	Why it matters	If lost
Source code	The work itself	Leak, theft
GitHub token	Can push code	Malicious commits
SSH private key	Identity to servers	Impersonation
Laptop disk	Everything at once	Total compromise

3. Adversaries, from cheapest to most expensive.
4. An automated scanner that tries known passwords on anything it finds. Cost to the attacker: near zero. Likelihood: certain, continuously.
5. A phishing email pretending to be GitHub. Cost: low. Likelihood: high.
6. A thief who steals the laptop from a cafe. Cost: moderate. Likelihood: low but real.
7. A targeted attacker who wants this specific person. Cost: high. Likelihood: low unless the person is unusually interesting.
8. Now score the risk. **Risk = likelihood x impact**. Both parts matter.

9. Stolen laptop: likelihood low, impact very high. Product: medium-high. So full disk encryption is worth it.
10. Automated password guessing: likelihood certain, impact high. Product: very high. So a hardware key or passkey on GitHub is worth it.
11. A nation-state adversary: likelihood very low for most readers. Do not spend your budget there first.
12. That last line is the whole discipline. You cannot defend everything, so you rank by likelihood times impact and start at the top.

■ 44.1.4 PLAIN – what is really happening inside

1. Every system has an **attack surface**: the complete list of places where untrusted input can reach your code.
2. Open network ports are attack surface. So are file uploads, query parameters, environment variables, image parsers and USB sockets.
3. Attackers do not “hack in” the way films show. They find one place where your assumption about input was wrong.
4. The pattern is nearly always the same. A program expected data and received instructions, or expected a small thing and received a big one.
5. There is a second, equally common pattern: the program did the right check on the wrong thing, or forgot the check on one path out of twenty.
6. Attacks chain. A low-value flaw plus another low-value flaw often equals a total compromise.
7. Example of a chain: a public page leaks internal server names, one of those servers has an old unpatched service, that service runs as administrator, and administrator can read the database credentials.
8. No single link in that chain was catastrophic. The chain was.
9. Defenders therefore think in layers, not in walls. If one control fails, the next one should still be standing. That is called **defence in depth**.
10. And defenders think about **blast radius**: when this component is fully compromised, what exactly can the attacker now reach?
11. Reducing blast radius is usually cheaper and more reliable than trying to make a component unbreakable.

The honest version: “attackers exploit vulnerabilities” is only half true. In real incident data, most intrusions begin with valid credentials or a person being tricked, not with a clever technical exploit. The technical exploit usually comes later, to move sideways once inside.

■ 44.1.5 TECHNICAL – the engineer’s version

1. The five properties are formalized in ISO/IEC 27000 and in NIST SP 800-53. Confidentiality, integrity and availability are the statutory triad in United States federal law under FISMA 2002.
2. Authenticity and non-repudiation are usually added as separate objectives; the combined five are sometimes called the Parkerian hexad when utility and possession are added as well, following Donn Parker’s 1998 formulation.
3. Threat modelling has named methodologies. STRIDE, created at Microsoft by Loren Kohnfelder and Praerit Garg in 1999, enumerates Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege.
4. PASTA (Process for Attack Simulation and Threat Analysis, 2012) and LINDDUN for privacy threats are the other two commonly taught.

5. Attacks are catalogued by MITRE ATT&CK, first published in 2013 and updated continuously. It maps real observed adversary behaviour into tactics and techniques with identifiers such as T1566 for phishing.
6. Vulnerabilities get CVE identifiers, run by MITRE since 1999. Severity is scored by CVSS, currently version 4.0, published by FIRST in November 2023.
7. Weakness classes get CWE identifiers. CWE-79 is cross-site scripting, CWE-89 is SQL injection, CWE-787 is out-of-bounds write.
8. Note the difference precisely: a CWE is a kind of mistake, a CVE is one specific instance of it in one specific product version.
9. Risk in formal work is expressed as annualized loss expectancy: $ALE = \text{single loss expectancy} \times \text{annual rate of occurrence}$. In practice the inputs are estimates, so the number is used for ranking, not for accounting.
10. FAIR (Factor Analysis of Information Risk) is the main quantitative alternative and expresses risk as probability distributions rather than single numbers.

Term	Formal meaning	Example
Threat	Actor with intent	Ransomware crew
Vulnerability	The weakness	Unpatched CVE
Exploit	Code using it	Working payload
Risk	Likelihood x impact	Ranked list

11. Tools that observe security posture on a machine you own: `nmap` for open ports, `ss -tlnp` on Linux or `lsof -i -P` on macOS for listening sockets, `osqueryd` for fleet-wide state, and `lynis` for host audit.
12. The word “secure” without a threat model attached is meaningless. Always ask: secure against whom, doing what, at what cost to them?

■ 44.1.6 WORDS - remember these

Confidentiality — only the right people can read it — protection of data from unauthorized disclosure.

Integrity — nobody changed it secretly — assurance that data has not been modified in an unauthorized or undetected way.

Availability — it works when you need it — the property of being accessible and usable on demand by an authorized entity.

Authenticity — you are who you claim — verified identity of a principal or origin of data.

Non-repudiation — you cannot deny you did it — cryptographic proof binding an action to an identity, resistant to later denial.

Threat model — who might attack and how — structured enumeration of assets, adversaries, entry points and mitigations.

Attack surface — every way in — the complete set of points where an untrusted actor can supply input or influence behaviour.

Blast radius — how far damage spreads — the set of assets reachable after a given component is fully compromised.

Risk — likelihood times impact — expected loss, used to rank work rather than to predict exact outcomes.

44.2 Hashing

■ 44.2.1 PLAIN – in simple words

1. A **hash function** takes any amount of data and produces a short, fixed-length fingerprint.
2. Feed it one letter or a two-gigabyte film. The fingerprint is the same length every time.
3. It is a one-way street. From the data you can compute the fingerprint easily. From the fingerprint you cannot get the data back.
4. Change one single bit of the input and about half the bits of the output change, in an unpredictable way.
5. That effect has a name: the **avalanche effect**.
6. The same input always gives the same output, on any machine, forever. It is not random.
7. Four properties matter, and they are worth learning in order.
8. One: it is fast to compute forwards.
9. Two: **preimage resistance**. Given a fingerprint, you cannot find any input that produces it.
10. Three: **second preimage resistance**. Given one file, you cannot find a different file with the same fingerprint.
11. Four: **collision resistance**. You cannot find any two different files with the same fingerprint, even if you get to choose both.
12. Collision resistance is the hardest of the four, and it is always the first one to fall when a hash function is broken.
13. A hash is good for checking that something did not change. A hash by itself is not a secret and not encryption. It hides nothing you can guess.

■ 44.2.2 PLAIN – a picture in your head

1. Imagine a machine that turns any book into a six-word summary.
2. The summary is always six words, whether the book is a pamphlet or a thousand pages.
3. From the six words nobody can reconstruct the book.
4. Change one comma in the book and the six words come out completely different, not slightly different.
5. So if a friend tells you the six words over the phone, you can check that the copy of the book you received is exactly the one they sent.
6. That is what people do when they publish a checksum next to a download.
7. A **collision** is two genuinely different books producing the same six words. If that ever happens, the summary can no longer prove anything.

Where this comparison breaks: a six-word summary of a book is related to the book's meaning. A cryptographic hash is deliberately not. Two files that differ by a single byte give summaries with no visible relationship at all, and that unpredictability is the entire point. Also, a real hash has a fixed tiny output while the input is unlimited, so collisions must mathematically exist. The claim is only that nobody can find one.

■ 44.2.3 PLAIN – a worked example

1. Here are real hashes, computed in the sandbox used to write this chapter, with OpenSSL 3.0.13.
2. The input is the seven characters `kedbyte`, with no newline.

```
md5      31d826fc69ac61899d792d7c21411dd5
sha1     ea431b1d4e594752169a04adea3245395d4ea7eb
sha256   7e88ddebb634ffbb28b3037dd122c08cf849ce53bd
         f3a8d0a08e13ac5b0eab3f
sha3-256 72c5e9c053ffabec027977d730987430dc98f9b553
         ac1aa248bc683385b532b1
blake2s  03b9df1f134bdfb34740d2353323dd58f48360c23c
         3af7b393fda6e52c6c14af
```

3. Note the lengths. MD5 is 32 hex characters, which is 128 bits. SHA-1 is 40 characters, 160 bits. SHA-256 is 64 characters, 256 bits.
4. Now change the last letter only, from `kedbyte` to `kedbytf`, and hash again with SHA-256.

```
kedbyte -> 7e88ddebb634ffbb28b3037dd122c08c...
kedbytf -> 718433d73cc98a937bdfce4a6c2e4d90...
```

5. Counted properly, 123 of the 256 output bits differ. The ideal is 128, which is exactly half. That is the avalanche effect measured.
6. SHA-256 of the empty input is a constant worth recognizing, because it turns up in logs and in Git internals: `e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b855`.
7. On the machine used here, `openssl speed sha256` measured 400 megabytes per second on 8 kilobyte blocks, on one core.
8. That speed is wonderful for verifying downloads and terrible for storing passwords, which is the subject of the next section.

■ 44.2.4 PLAIN – what is really happening inside

1. Most classic hash functions work by chewing the input in fixed-size blocks.
2. SHA-256 uses 512-bit blocks. It keeps eight 32-bit working values, called the state, that start from fixed constants.
3. For each block it runs 64 rounds of mixing: rotations, exclusive-or, additions and a few nonlinear choices between bits.
4. After each block, the mixed values are added back into the state. That step is what makes the process irreversible.
5. At the end, the eight state values are written out as the digest.
6. The padding rule matters. The message gets a single 1 bit, then zeros, then the original length in bits. Without the length, different messages could collide trivially.
7. This design is called Merkle-Damgard, after Ralph Merkle and Ivan Damgard, both 1979 to 1989 era work.
8. It has a known quirk called **length extension**: if you know the hash of a secret message and its length, you can compute the hash of that message with extra data appended, without knowing the secret.
9. That quirk is exactly why you must never authenticate a message by hashing a secret joined to the message. You use HMAC instead, which is built to resist it.
10. SHA-3 has a completely different internal shape called a sponge. It absorbs input into a large 1600-bit state and squeezes output back out, and it is immune to length extension by design.
11. BLAKE3 goes further and splits the input into 1 kilobyte chunks arranged in a tree, so many cores can hash different chunks at the same time.

12. When a hash is “broken”, it almost never means someone can reverse it. It means someone found a shortcut to produce collisions faster than brute force. That is still fatal for signatures and certificates.

■ 44.2.5 TECHNICAL – the engineer’s version

1. MD5 was published by Ron Rivest in 1992 as RFC 1321, output 128 bits.
2. Practical MD5 collisions were announced by Wang Xiaoyun, Feng Dengguo, Lai Xuejia and Yu Hongbo at the CRYPTO 2004 rump session in August 2004.
3. By 2008 a research team used chosen-prefix MD5 collisions to forge a working certificate authority certificate. In 2012 the Flame malware used the same class of attack to forge a Microsoft code-signing certificate.
4. SHA-1 was published by NIST in 1995, output 160 bits. Theoretical attacks appeared in 2005, also from Wang’s group.
5. The first public SHA-1 collision, named SHAttered, was announced on 23 February 2017 by CWI Amsterdam and Google. It cost roughly nine quintillion SHA-1 computations, about 6,500 CPU-years and 110 GPU-years.
6. In 2020 Gaetan Leurent and Thomas Peyrin demonstrated a chosen-prefix SHA-1 collision for a rented compute cost in the tens of thousands of dollars.
7. SHA-2 is a family standardized in FIPS 180-4: SHA-224, SHA-256, SHA-384, SHA-512, and the truncated SHA-512/224 and SHA-512/256. No practical break exists as of 2026.
8. SHA-3 is Keccak, by Guido Bertoni, Joan Daemen, Michael Peeters and Gilles Van Assche. It won the NIST competition in October 2012 and was published as FIPS 202 in August 2015. It is a complement to SHA-2, not a replacement.
9. BLAKE3 was released in January 2020 by Jack O’Connor, Jean-Philippe Aumasson, Samuel Neves and Zooko Wilcox-O’Hearn. It is not a NIST standard.
10. Security levels, stated honestly. A hash with n bits of output gives about n bits against preimage attacks and only $n/2$ bits against collisions, by the birthday bound. SHA-256 therefore offers 128-bit collision resistance.

Function	Output bits	Status 2026
MD5	128	Broken 2004
SHA-1	160	Broken 2017
SHA-256	256	Recommended
SHA-3-256	256	Recommended
BLAKE3	256 default	Fast, not FIPS

11. Where each is still acceptable: MD5 and SHA-1 only for non-security uses such as cache keys and non-adversarial deduplication. Never for signatures, certificates, integrity against an attacker, or passwords.
12. Git historically used SHA-1 for object names and has a SHA-256 object format available since Git 2.29 in 2020, still not the default in 2026. Git also ships a hardened SHA-1 that detects the SHAttered attack pattern.
13. Commands to observe: `sha256sum file` and `md5sum file` on Linux, `shasum -a 256 file` on macOS, `openssl dgst -sha256 file` anywhere, `certutil -hashfile file SHA256` on Windows.
14. For message authentication use HMAC (RFC 2104, 1997), for example HMAC-SHA-256, or use a modern AEAD cipher which includes authentication.

■ 44.2.6 WORDS – remember these

Hash function — a fingerprint machine — a deterministic map from arbitrary input to fixed-length output.

Digest — the fingerprint itself — the output value of a hash function.

Collision — two inputs, one fingerprint — two distinct messages producing the same digest.

Preimage resistance — cannot work backwards — infeasibility of finding any input matching a given digest.

Avalanche effect — one bit in, half the bits out — small input change causes statistically independent output.

Length extension — appending without the secret — a Merkle-Damgard weakness allowing extension of a hashed message.

HMAC — a hash with a key done properly — keyed message authentication code defined in RFC 2104.

Checksum — a change detector — a short value used to detect accidental corruption, not necessarily cryptographic.

44.3 Password storage done properly

■ 44.3.1 PLAIN – in simple words

1. If you build a login system, you must store something that lets you check a password without storing the password.
2. Storing the password itself, readable, is the worst possible choice. One database leak and every user is exposed everywhere they reused it.
3. So we store a hash instead. The user types a password, we hash it, we compare hashes. We never need the original.
4. But a plain fast hash is almost as bad, and here is why.
5. An attacker who steals the database can guess billions of passwords per second on one graphics card and hash each guess.
6. Worse, if everyone's password is hashed the same way, the attacker can precompute a giant table once and reuse it against every stolen database.
7. Those precomputed tables are called **rainbow tables**.
8. The fix for that is a **salt**: a random value, different for every user, mixed in before hashing and stored alongside the hash.
9. Two users with the same password now get completely different stored values, so a precomputed table is useless and every user must be attacked separately.
10. A salt is not a secret. It is stored in the database in plain view. Its job is uniqueness, not secrecy.
11. Then there is a second idea: a **pepper**, a single secret value added for everyone, stored somewhere other than the database.
12. If the attacker steals only the database, the pepper is missing and the guessing cannot even start.
13. And the last idea is the important one: make the hash deliberately slow and memory-hungry.
14. If checking one password takes a quarter of a second and a lot of memory, a legitimate login is unaffected and mass guessing becomes uneconomic.

■ 44.3.2 PLAIN – a picture in your head

1. Think of a hotel that must check whether you know the door code, without keeping a list of door codes.
2. Fast hash: the clerk has a machine that instantly turns a code into a coloured token, and keeps only the tokens. A thief who steals the tokens buys the same machine and tries every code in an afternoon.
3. Salt: each guest's code is combined with a printed sticker number unique to that guest before the machine runs. The thief must now redo the whole afternoon separately for each guest.
4. Pepper: the machine also needs a small key that the manager keeps at home. Stealing the tokens gets you nowhere without visiting the manager.
5. Slow hash: the machine takes a quarter of a second per attempt and needs a whole desk covered in paper to work. One guest checking in does not care. A thief trying a billion codes now needs a warehouse and years.

Where this comparison breaks: the hotel thief cannot buy a hundred thousand machines. A password attacker can rent that many in a cloud for an hour. The slow hash is not making attack impossible; it is changing the price. Price is the real currency of security, and it is a moving target because hardware gets faster. Parameters chosen in 2015 are too weak in 2026.

■ 44.3.3 PLAIN – a worked example

1. Real timings, measured in the sandbox for this chapter, single core.

Method	Time per check	Attacker friendly?
Raw SHA-256	0.000000074 s	Extremely
PBKDF2 600k	0.42 s	No
bcrypt N=2 ¹⁷	1.08 s	No

2. Read the first row again. A raw SHA-256 took 740 nanoseconds. That is roughly 1.3 million guesses per second on one Python core.
3. On a graphics card the numbers are far worse for the defender. Published hashcat benchmarks on an NVIDIA RTX 5090 report about 220 billion MD5 hashes per second and about 28 billion SHA-256 hashes per second.
4. The same card manages only about 300,000 bcrypt hashes per second at a low work factor. That is roughly a hundred thousand times slower per guess.
5. That single ratio is the entire argument for slow password hashing.
6. Now watch the salt work. The same password, PBKDF2-HMAC-SHA256 with 600,000 iterations, with two different salts.

```

salt 00000000... -> 0460eeec7ddf8b5f91f2037b3e2ab248
salt a3f10000... -> 52e0053a9dda9d910cb58befaaf9e67

```

7. Same password, unrelated results. No shared precomputation is possible.
8. A stored record in practice looks like this. It is one string containing the algorithm, its parameters, the salt and the hash, so you can change parameters later without breaking old accounts.

```

$argon2id$v=19$m=19456,t=2,p=1$c29tZXNhbnHQ$aG...
|      |      |      |      |
algo    ver  parameters    salt    hash

```

9. The parameters live inside the record on purpose. When you raise them next year, old users still verify with the old settings, and you re-hash their password at their next successful login.

■ 44.3.4 PLAIN - what is really happening inside

1. On registration: generate a fresh random salt of at least 16 bytes from a cryptographic random source, run the slow hash, store algorithm, parameters, salt and result.
2. On login: read the stored record, take the salt and parameters out of it, hash the submitted password the same way, compare.
3. Compare in constant time. A comparison that stops at the first different byte leaks information through timing. Every good library has a constant-time compare function.
4. Never tell the user which part was wrong. "Wrong username or password" and nothing more, or you have built an account enumeration tool.
5. The slow hash resists attack in three different ways, and they are not the same thing.
6. Time hardness: many sequential iterations, so you cannot go faster than the chain allows.
7. Memory hardness: the algorithm needs a large block of memory that must stay filled during the computation. Graphics cards have thousands of small cores but limited memory per core, so memory cost hurts attackers far more than defenders.
8. Parallelism resistance: the memory accesses depend on earlier results, so the work cannot simply be split.
9. bcrypt has time hardness and a small fixed 4 kilobyte memory need. scrypt and Argon2 add real memory hardness.
10. The pepper is applied differently in good designs: not concatenated into the password, but used as an HMAC key over the password before hashing, or used to encrypt the resulting hash. Both keep it separable and rotatable.
11. Where does the pepper live? A hardware security module, a cloud key management service, or at minimum an environment variable on the application server that is not in the database backup.

The honest version: the pepper only helps in the specific case where the database leaks but the application server does not. That case is common, from SQL injection and stolen backups, so peppering is worth doing. But it is a second line, not a substitute for a proper slow hash.

■ 44.3.5 TECHNICAL - the engineer's version

1. bcrypt: Niels Provos and David Mazieres, USENIX 1999, based on the Blowfish key schedule. Cost parameter is a power of two. OWASP recommends a work factor of at least 10 as of the current Password Storage Cheat Sheet.
2. bcrypt truncates input at 72 bytes. Enforce a 72-byte maximum, or pre-hash with SHA-256 and base64 the result before feeding bcrypt, being careful about null bytes.
3. scrypt: Colin Percival, 2009, standardized as RFC 7914 in 2016. Parameters N (cost), r (block size), p (parallelism).
4. Argon2: winner of the Password Hashing Competition in July 2015, by Alex Biryukov, Daniel Dinu and Dmitry Khovratovich. Standardized as RFC 9106 in September 2021. Three variants: Argon2d, Argon2i, and Argon2id.
5. Use Argon2id. It is the hybrid: data-independent memory access in the first pass to resist side-channel attacks, data-dependent afterwards to resist time-memory trade-offs.
6. Current OWASP recommended parameters, any one of which is acceptable and equivalent in strength.

Algorithm	Parameters	Notes
Argon2id	m=47104, t=1, p=1	46 MiB memory
Argon2id	m=19456, t=2, p=1	19 MiB memory
scrypt	N=2 ¹⁷ , r=8, p=1	128 MiB memory
bcrypt	cost 10 minimum	72-byte limit

7. PBKDF2 is the fallback where FIPS validation is required. OWASP currently lists 600,000 iterations for PBKDF2-HMAC-SHA256 and 210,000 for PBKDF2-HMAC-SHA512. PBKDF2 is defined in RFC 8018.
8. Salt: 16 bytes minimum from a CSPRNG. Every modern library generates and embeds it for you. Do not hand-roll this.
9. Attack taxonomy, defensively stated. A **dictionary attack** tries a wordlist plus rules such as appending digits. A **brute force** tries all combinations, feasible only for short passwords. **Credential stuffing** replays username and password pairs stolen from another site.
10. Credential stuffing is the dominant real attack because password reuse is near universal. The attacker does not need to break your hashes at all if the user used the same password somewhere weaker.
11. That is the actual disaster of reuse: your security becomes equal to the security of the worst site the user ever signed up for.
12. Defences against stuffing: rate limiting per account and per network, device fingerprinting, checking submitted passwords against known-breached lists using a k-anonymity range API, and above all offering MFA.
13. NIST SP 800-63B revision 4, published 31 July 2025, is the current authority. Its key rules for verifiers: require at least 8 characters and encourage at least 15; allow at least 64; allow all printable characters including spaces and Unicode.
14. The same document says verifiers **shall not** impose composition rules such as requiring mixed character types, and **shall not** require periodic password rotation without evidence of compromise.
15. The reasoning is empirical, not aesthetic. Forced rotation causes predictable transformations such as incrementing a trailing digit, and composition rules push users toward the same small set of substitutions.
16. Practical advice to give a user: use a password manager, generate long random passwords per site, and use a long passphrase only for the few secrets you must type from memory, such as the manager's own master password and the disk encryption password.
17. Length beats complexity. A five-word random passphrase from a large word list carries roughly 64 bits of entropy; an eight-character password with the usual substitutions carries far less in practice.

■ 44.3.6 WORDS – remember these

Salt — a unique random extra per user — public per-record value preventing precomputation and hash reuse.

Pepper — one secret extra for everyone — site-wide secret kept outside the password database, applied via HMAC or encryption.

Rainbow table — a precomputed cracking table — time-memory trade-off structure for reversing unsalted hashes.

Work factor — the slowness dial — a tunable cost parameter of a password hashing function.

Memory hardness — needs a lot of RAM to compute — property that raises the cost of parallel hardware attacks.

Credential stuffing — reusing stolen logins elsewhere — automated replay of breached username and password pairs.

Argon2id — the current recommended choice — hybrid memory-hard password hash, RFC 9106, winner of the 2015 Password Hashing Competition.

Account enumeration — learning which users exist — information leak from differing error messages or response times.

44.4 Symmetric encryption

■ 44.4.1 PLAIN - in simple words

1. **Symmetric encryption** means both sides share one secret key. The same key locks and unlocks.
2. It is fast, which is why every real system uses it for the actual data.
3. There are two shapes of symmetric cipher.
4. A **block cipher** transforms a fixed-size chunk at a time, typically 16 bytes, and you must decide how to handle a message longer than one chunk.
5. A **stream cipher** produces an endless keystream of pseudo-random bytes which you combine with the message one byte at a time.
6. The dominant block cipher is **AES**, with key sizes of 128, 192 or 256 bits. Its block is always 128 bits regardless of key size.
7. The way you chain blocks together is called a **mode of operation**, and choosing it wrongly is the most common cryptographic mistake in the world.
8. The worst mode is ECB, which encrypts every block independently. Identical input blocks give identical output blocks, so the shape of your data leaks straight through.
9. Modern practice is not to choose a cipher and a mode separately. You choose an **authenticated encryption** scheme that also detects tampering.
10. Encryption without authentication is not enough. An attacker who cannot read your message can still flip bits in it and change what it says when decrypted.
11. Almost every cipher mode also needs a per-message unique value called an **IV** or a **nonce**. Reusing one with the same key is catastrophic.

■ 44.4.2 PLAIN - a picture in your head

1. Think of a stencil-based code where each 16-letter chunk of your message is replaced by a scrambled 16-letter chunk, using a fixed rule set by the key.
2. ECB mode is applying that stencil to each chunk independently.
3. Now imagine a photograph converted this way. Every patch of plain white sky becomes the same scrambled patch. The scrambled version still shows the outline of the picture.
4. That is why the encrypted penguin image is famous in every cryptography course. The bird is still clearly visible after ECB encryption.
5. CBC mode fixes it by mixing each chunk with the previous encrypted chunk before scrambling, so identical inputs stop looking identical.
6. CTR mode does something different: it does not encrypt your data at all. It encrypts a counter to make a keystream, then combines that with your data.
7. GCM is CTR plus a running checksum computed with the key, so any change to the ciphertext is detected on decryption.

Where this comparison breaks: real block ciphers are not letter substitution. AES mixes bits across the whole block through several layers, so one changed input bit changes the whole output block. The stencil picture also suggests the key changes the alphabet, when in reality the key changes a schedule of operations applied over many rounds.

■ 44.4.3 PLAIN – a worked example

1. Here is ECB failing, computed in the sandbox for this chapter with a real AES-128 key.
2. The plaintext is YES YES YES YES YES YES YES YES, which is exactly two identical 16-byte blocks.

```
AES-ECB block 1: 15516924691d0a3e5f81ff297ad6c86e
AES-ECB block 2: 15516924691d0a3e5f81ff297ad6c86e
identical: True
```

```
AES-CBC block 1: 15516924691d0a3e5f81ff297ad6c86e
AES-CBC block 2: 234d8347677995c4547c148bbfa58f4f
identical: False
```

3. Look at the ECB output. The repetition in the plaintext is visible in the ciphertext with no key and no effort at all.
4. CBC produced different blocks for identical input because each block was mixed with the previous ciphertext block first.
5. Now the nonce reuse problem, stated without giving an attack recipe.
6. In CTR and GCM, the ciphertext is the plaintext combined with a keystream derived from key and nonce.
7. Encrypt two different messages with the same key and the same nonce, and both use the identical keystream.
8. Combining the two ciphertexts cancels the keystream out entirely, leaving a direct relationship between the two plaintexts.
9. For GCM specifically, nonce reuse does something worse: it leaks the authentication subkey, which lets an attacker forge messages that your system will accept as genuine.
10. The rule that follows is absolute. Never reuse a nonce with the same key. Generate it randomly with enough bits, or use a strict counter you can prove never repeats.

■ 44.4.4 PLAIN – what is really happening inside

1. AES works on a 4 by 4 grid of bytes called the state.
2. Each round does four steps: substitute every byte through a fixed lookup table, shift the rows sideways by different amounts, mix each column mathematically, and add the round key with exclusive-or.
3. AES-128 does 10 rounds, AES-192 does 12, AES-256 does 14.
4. The **key schedule** expands the single key into one distinct round key per round, so no two rounds are the same.
5. The substitution table is the only nonlinear step. Without it the whole cipher would collapse into simple algebra and be trivially breakable.
6. In CBC mode, encryption is a chain: block n is combined with ciphertext n-1 before encryption, and the first block uses the IV. Chains cannot be parallelized when encrypting, though decryption can be.
7. In CTR mode, block n of the keystream is the encryption of nonce plus n. No chain, so everything parallelizes and you can decrypt from the middle.
8. GCM adds GHASH, a multiplication in a finite field over the ciphertext and any associated data, producing a 16-byte authentication tag.

9. On decryption, GCM computes the tag again and compares. If it does not match, it returns an error and no plaintext. That refusal is the whole value of authenticated encryption.
10. ChaCha20 is a different design: 20 rounds of additions, rotations and exclusive-ors on a 4 by 4 grid of 32-bit words, producing 64 bytes of keystream per invocation.
11. It uses no lookup tables, so it has no cache-timing side channel and it is fast in plain software on processors without AES hardware instructions.
12. Poly1305 is its companion authenticator, a one-time message authentication code using arithmetic modulo a large prime.

■ 44.4.5 TECHNICAL – the engineer’s version

1. AES is Rijndael, by Joan Daemen and Vincent Rijmen. It won the NIST competition in October 2000 and was published as FIPS 197 in November 2001.
2. Modern processors have hardware support: Intel AES-NI since Westmere in 2010, ARMv8 cryptographic extensions since 2011. Throughput on a single modern core is commonly several gigabytes per second.
3. Mode summary, with the practical verdict for 2026.

Mode	Authenticated?	Verdict
ECB	No	Never use
CBC	No	Legacy only
CTR	No	Only inside AEAD
GCM	Yes	Standard choice
ChaCha20-Poly1305	Yes	Standard choice

4. AES-GCM is specified in NIST SP 800-38D (2007). Its nonce is 96 bits by convention, and the standard limits a single key to about 2^{32} random nonces before collision risk becomes unacceptable.
5. AES-GCM-SIV, RFC 8452 (2019), is nonce-misuse resistant: repeating a nonce leaks only whether two plaintexts were equal, not the keystream. Use it when you cannot guarantee nonce uniqueness.
6. ChaCha20-Poly1305 is RFC 8439 (2018), by Daniel J. Bernstein. It is a TLS 1.3 mandatory-to-implement suite alongside AES-GCM and is preferred on mobile processors without AES acceleration.
7. XChaCha20-Poly1305 extends the nonce to 192 bits, making random nonce generation safe without counters.
8. CBC without authentication enabled the padding oracle class of attacks, including the 2002 Vaudenay result and the 2010 attacks on ASP.NET. That history is why encrypt-then-MAC or AEAD is mandatory now, not optional.
9. Key sizes in practice: AES-128 gives 128-bit classical security and is fine. AES-256 is chosen for long-term data and for regulatory reasons, and is the sensible default against future quantum search attacks.
10. Grover’s algorithm, published by Lov Grover in 1996, would give a quantum computer a square-root speed-up on key search, notionally halving AES-256 to 128 bits of effective strength. Doubling the key size answers it.
11. Never write your own mode, your own padding, or your own comparison of authentication tags. Use libsodium, the Go `crypto/cipher` AEAD interface, Java’s `javax.crypto` with GCM, or Python cryptography’s `AESGCM` class.

12. Commands to observe: `openssl enc -aes-256-cbc` for files, `openssl speed -evp aes-256-gcm` for throughput, `cryptsetup benchmark` on Linux for disk encryption performance including AES-XTS.
13. Disk encryption uses XTS mode rather than GCM, because a disk sector has no room to store an authentication tag. That is a real trade-off: disk encryption protects confidentiality, not integrity.

■ 44.4.6 WORDS - remember these

Symmetric encryption — one shared key both ways — encryption where the same key encrypts and decrypts.

Block cipher — scrambles fixed chunks — a keyed permutation over a fixed block size, 128 bits for AES.

Mode of operation — how chunks are chained — the construction turning a block cipher into a scheme for arbitrary-length messages.

ECB — the mode that leaks patterns — Electronic Codebook, encrypts each block independently, never acceptable.

IV / nonce — the per-message unique value — initialization vector or number used once, must never repeat under one key.

AEAD — encryption that also detects tampering — Authenticated Encryption with Associated Data, for example AES-GCM.

Authentication tag — the tamper seal — short value verified on decryption, causing refusal if the ciphertext changed.

ChaCha20-Poly1305 — the software-fast alternative — stream cipher plus one-time authenticator, RFC 8439.

44.5 Asymmetric encryption, signatures and the quantum question

■ 44.5.1 PLAIN - in simple words

1. Symmetric encryption has one hard problem: how do two strangers agree on a shared key over a network anyone can read?
2. **Asymmetric encryption** solves it with a pair of matched keys.
3. The **public key** can be published to the whole world. The **private key** never leaves its owner.
4. Anything encrypted with the public key can only be decrypted with the private key.
5. That direction gives you confidentiality: anyone can write to you, only you can read it.
6. Run it the other way and you get something different and equally useful.
7. If you transform a message with your private key, anyone with your public key can check it. That is a **digital signature**.
8. A signature proves three things at once: the message came from the holder of that private key, it has not been altered, and the signer cannot plausibly deny it later.
9. Encryption alone proves none of those. Anyone with your public key can encrypt to you, so a decrypted message tells you nothing about who sent it.
10. There is also a way to agree a shared secret without either side sending it: **Diffie-Hellman key exchange**.
11. Asymmetric operations are slow, hundreds to thousands of times slower than symmetric ones.

12. So every real system is **hybrid**: use asymmetric maths once to agree a symmetric key, then use the fast symmetric cipher for all the data.
13. That is exactly what TLS does, which Chapter 32 covered in detail.

■ 44.5.2 PLAIN – a picture in your head

1. Picture an open padlock that you hand out freely, of which you alone keep the key.
2. Anyone can put a letter in a box, snap your padlock shut, and post it. Only you can open it.
3. Signing is the mirror image: you press a seal that only your ring can make, and anyone with a picture of your seal can confirm it is genuine.
4. Diffie-Hellman is a different trick. Two people each mix a private colour into a shared public colour, swap the mixtures in the open, then each mix their own private colour into the mixture they received.
5. Both end with the same final colour. An observer saw the two mixtures but cannot unmix them to reach it.

Where this comparison breaks: paint mixing is only a metaphor for a one-way mathematical operation. Real Diffie-Hellman uses exponentiation in a group where reversing it is the discrete logarithm problem. Also, a physical padlock protects against a person with a hacksaw, while a cryptographic one protects only against computation, and computation gets cheaper every year.

■ 44.5.3 PLAIN – a worked example

1. RSA rests on multiplication being easy and factoring being hard.
2. Multiplying two 1024-bit primes takes microseconds. Recovering those primes from the 2048-bit product is beyond any known machine.
3. The largest RSA modulus ever publicly factored is RSA-250, a 829-bit number, broken in February 2020 using about 2,700 core-years.
4. That is why 2048-bit RSA keys are still standard and 1024-bit ones were deprecated by browsers around 2013 to 2014.
5. Elliptic curves get the same strength from much smaller numbers.

Security level	RSA / DH	Elliptic curve
112 bits	2048	224
128 bits	3072	256
192 bits	7680	384
256 bits	15360	512

6. Read the last row. To match a 512-bit curve you would need a 15,360-bit RSA key. That is why modern systems moved to curves.
7. Curve25519, published by Daniel J. Bernstein in 2005, gives roughly 128-bit security with 32-byte keys and is the default in SSH, Signal and TLS 1.3.
8. Its signature counterpart is Ed25519, standardized in RFC 8032 in 2017, with 64-byte signatures.
9. A sense of the speed gap: on a typical modern core, an X25519 key agreement takes tens of microseconds while an AES-GCM operation on a kilobyte takes well under one microsecond. Hence the hybrid design.

■ 44.5.4 PLAIN - what is really happening inside

1. RSA key generation: pick two large random primes p and q , multiply to get the modulus n , choose a public exponent e , usually 65537, and compute the matching private exponent d .
2. Encryption raises the message to the power e modulo n . Decryption raises the result to the power d modulo n . The maths brings you back to the start.
3. Raw RSA is unsafe. Real use requires padding: OAEP for encryption, PSS for signatures. The old PKCS#1 v1.5 padding produced the Bleichenbacher attack family from 1998 onwards and its many modern echoes.
4. Signing does not actually sign the message. It hashes the message first, then signs the hash. That is why a broken hash breaks signatures, which is what SHA-1 collisions meant for certificates.
5. Elliptic curve cryptography replaces “multiply numbers” with “add points on a curve”. The private key is a number, the public key is a point reached by adding a fixed base point to itself that many times.
6. Going forwards is fast; recovering the number from the point is the elliptic curve discrete logarithm problem, and no efficient classical method is known.
7. Diffie-Hellman in curve form is simple: each side multiplies the other’s public point by their own private number, and both land on the same point.
8. **Forward secrecy** comes from throwing the Diffie-Hellman private values away after the session. An attacker who later steals the long-term key still cannot decrypt yesterday’s recorded traffic.
9. Now the quantum part. Peter Shor published an algorithm in 1994 that, on a sufficiently large fault-tolerant quantum computer, factors integers and computes discrete logarithms in polynomial time.
10. That breaks RSA, finite-field Diffie-Hellman and every elliptic curve scheme at once. It does not break symmetric ciphers or hashes in the same way; those need only larger sizes.
11. No such machine exists in 2026. Published estimates for breaking 2048-bit RSA remain in the range of millions of physical qubits with error correction, against current devices measured in the low thousands.
12. But traffic recorded today can be stored and decrypted later. That is **harvest now, decrypt later**, and it makes migration urgent for any data that must stay secret for ten or twenty years.

■ 44.5.5 TECHNICAL - the engineer’s version

1. Diffie-Hellman was published by Whitfield Diffie and Martin Hellman in 1976 in “New Directions in Cryptography”. Ralph Merkle’s related work preceded it. RSA followed in 1977 from Ron Rivest, Adi Shamir and Leonard Adleman.
2. Elliptic curve cryptography was proposed independently by Neal Koblitz and Victor Miller in 1985.
3. British researchers at GCHQ, James Ellis, Clifford Cocks and Malcolm Williamson, developed equivalent ideas between 1969 and 1974; the work was classified until 1997.
4. NIST approved three post-quantum standards on 13 August 2024.

Standard	Name	Purpose
FIPS 203	ML-KEM	Key encapsulation
FIPS 204	ML-DSA	Digital signatures
FIPS 205	SLH-DSA	Hash-based signatures

5. ML-KEM was submitted as CRYSTALS-Kyber, ML-DSA as CRYSTALS-Dilithium, SLH-DSA as SPHINCS+. The initial announcement was 6 August 2024 and the Secretary of Commerce ap-

proved them on 13 August 2024.

6. A fourth signature scheme, FN-DSA, based on FALCON, was still in draft as FIPS 206 as of 2026. On 11 March 2025 NIST selected HQC as a fifth algorithm and a backup key encapsulation mechanism, with its own standard expected around 2027.
7. Sizes are the practical cost. ML-KEM-768 has a 1,184-byte encapsulation key and a 1,088-byte ciphertext, against 32 bytes each for X25519.
8. ML-DSA-65 signatures are about 3,309 bytes and SLH-DSA signatures range from roughly 7,856 bytes to about 49,856 bytes, against 64 bytes for Ed25519. That is why signature migration lags key exchange migration.
9. SLH-DSA exists as insurance. It relies only on hash function security, so if lattice mathematics turns out to have an unexpected weakness, hash-based signatures still stand.
10. Deployment state as of 2026, separating fact from claim. Established fact: the hybrid key agreement X25519MLKEM768 is on by default in current Chrome, Firefox and Edge. Cloudflare reported in late October 2025 that over half of human-initiated traffic to its network used post-quantum key agreement.
11. Also established: signatures and certificates have barely moved, because they need certificate authorities, root programmes and every client to change together.
12. Active research: how to shrink post-quantum certificate chains, and how to do post-quantum authentication for constrained devices.
13. Marketing claim: any product described as “quantum-proof” or “unbreakable” without naming FIPS 203, 204 or 205. Ask which standard, which parameter set, and whether it is hybridized with a classical algorithm.
14. Hybrid is the current professional consensus: run the classical and the post-quantum algorithm together, so a break in either alone is not fatal. Experts disagree on how long to keep hybrids. Some say they add complexity and should be retired soon; others argue for a decade.
15. Commands to observe: `openssl s_client -connect host:443` and read the negotiated group, `openssl list -kem-algorithms` on OpenSSL 3.5 and later, and `ssh -Q kex` to list key exchange algorithms your SSH build supports.

■ 44.5.6 WORDS – remember these

Public key — the half you publish — the shareable component of an asymmetric key pair.

Private key — the half you never share — the secret component whose exposure destroys the pair’s security.

Digital signature — proof of who and unaltered — a value verifiable with a public key, providing authenticity and non-repudiation.

Key exchange — agreeing a secret in the open — a protocol producing a shared secret without transmitting it, such as ECDH.

Forward secrecy — yesterday stays safe — property where compromise of long-term keys does not expose past sessions.

Hybrid encryption — asymmetric once, symmetric after — the standard construction combining key agreement with an AEAD cipher.

Shor’s algorithm — the quantum threat to RSA — polynomial-time quantum factoring and discrete logarithm, published 1994.

ML-KEM — the new key exchange standard — FIPS 203, lattice-based key encapsulation, approved 13 August 2024.

Harvest now, decrypt later — record today, break tomorrow — the threat model justifying migration before quantum computers exist.

44.6 Authentication and identity

■ 44.6.1 PLAIN – in simple words

1. **Authentication** answers one question: are you who you claim to be?
2. There are three classic kinds of evidence, called factors.
3. Something you **know**: a password, a PIN, an answer to a question.
4. Something you **have**: a phone, a hardware key, a smart card.
5. Something you **are**: a fingerprint, a face, a voice.
6. Multi-factor authentication means using evidence from two different kinds, not two things of the same kind.
7. A password plus a security question is not two factors. Both are things you know, and both leak in the same database breach.
8. Not all second factors are equally strong, and the gap between them is much wider than most people think.
9. A six-digit code from an app is decent. A code sent by text message is weak. A hardware key is in a different class entirely.
10. The reason hardware keys are different is that they refuse to be phished. The others can all be relayed by an attacker in real time.
11. A **passkey** is that same phishing-resistant technology, with the secret stored on your phone or laptop and unlocked by your fingerprint or face.

■ 44.6.2 PLAIN – a picture in your head

1. Think of collecting a parcel from a counter.
2. The password is knowing the reference number. Anyone who overhears it can collect your parcel.
3. The app code is a slip of paper that changes every thirty seconds. Better, but if someone phones you and talks you into reading it out within those thirty seconds, they can still collect the parcel.
4. The hardware key is different in kind. It is a stamp that only works on that one counter, in that one building.
5. Take the stamp to a fake counter in the next street and it produces nothing, because it checks the address of the counter before stamping.
6. That address check is why phishing fails against it. The user can be fully fooled and the attack still does not work.

Where this comparison breaks: a stamp can be stolen physically. A hardware key can too, which is why it usually also requires a PIN or a fingerprint. And the key does not check a street address but the exact web origin, which the browser supplies and the user cannot override by being persuaded.

■ 44.6.3 PLAIN – a worked example

1. Here is how the six-digit code in your authenticator app is actually made.
2. When you scan the QR code, the site gives your app a shared secret, usually 20 bytes, encoded in base32.
3. Both sides take the current Unix time in seconds and divide by 30, throwing away the remainder. That gives a counter that changes every 30 seconds.

4. Both compute HMAC-SHA-1 of that counter using the shared secret.
5. That produces 20 bytes. A truncation step picks 4 bytes from a position determined by the last nibble of the hash, giving a 31-bit number.
6. Take that number modulo 1,000,000 and pad to six digits. That is your code.

```
secret + floor(unixtime / 30)
-> HMAC-SHA-1
-> dynamic truncation (31 bits)
-> mod 1000000
-> "042318"
```

7. Nothing is transmitted to generate it. That is why the app works on a plane with no signal.
8. Servers usually accept the previous and next window too, to tolerate clocks being a little out.
9. Now see the weakness clearly. The code is just six digits typed by a human. A convincing fake login page collects it and replays it within 30 seconds.
10. That is not a flaw in the maths. It is a flaw in the shape of the interaction, and no amount of better hashing fixes it.

■ 44.6.4 PLAIN – what is really happening inside

1. SMS one-time codes are weaker still, for reasons outside your control.
2. **SIM swapping**: an attacker persuades a mobile operator to move your number to their SIM, using social engineering on a support agent. Your codes now arrive on their phone.
3. The mobile signalling network itself has known interception weaknesses, and messages travel through third parties in plain form.
4. Push-approval prompts fixed the typing problem but created a new one: **MFA fatigue**, where an attacker with your password sends prompt after prompt until you approve one to make it stop, often at three in the morning.
5. This is not theoretical. The September 2022 Uber breach began with a contractor's stolen password plus repeated push prompts, finished off by a message pretending to be internal IT support.
6. Number matching, where the login screen shows two digits you must type into the prompt, is the standard mitigation and is now on by default in the major identity providers.
7. Now the hardware key. Registration: the key generates a fresh key pair for that one website, keeps the private key inside the hardware, and sends the public key to the site.
8. Login: the site sends a random challenge. The browser adds the exact origin it is talking to. The key signs both together.
9. If the origin is a look-alike domain, the signature is over the wrong origin and the real site rejects it. There is no code for a human to relay.
10. The private key never leaves the device. There is no shared secret in the site's database, so a breach of the site leaks only public keys.
11. A **passkey** is the same protocol with the key material held by the operating system or password manager and typically synchronized across your devices through your platform account.
12. That synchronization is the trade-off. It makes recovery practical, at the cost of the cloud account becoming part of the security boundary.

■ 44.6.5 TECHNICAL – the engineer’s version

1. HOTP is RFC 4226 (2005), counter-based. TOTP is RFC 6238 (2011), time-based, with a default 30-second step and 6 digits. The default HMAC is SHA-1, which is acceptable here because HMAC-SHA-1 is not broken by collisions.
2. FIDO2 combines two specifications: W3C Web Authentication, WebAuthn Level 2 became a W3C Recommendation in April 2021 with Level 3 in progress, and CTAP2 from the FIDO Alliance for the link between browser and authenticator.
3. WebAuthn signs a client data structure containing the challenge, the origin and the type. Origin binding is the phishing resistance, and it is a property of the protocol, not of user training.
4. Attestation optionally proves what kind of authenticator was used. Enterprises use it to require certified hardware; consumer sites usually should not, for privacy reasons.
5. Passkeys are discoverable WebAuthn credentials. The FIDO Alliance State of Passkeys report of 7 May 2026 put around 5 billion passkeys in use, with 75 percent of surveyed consumers having enabled at least one and 68 percent of surveyed organizations deploying them.
6. Comparison of second factors, ranked honestly.

Factor	Phishing resistant	Main weakness
SMS code	No	SIM swap, relay
TOTP app	No	Real-time relay
Push approve	No	Fatigue attacks
Number match	Partly	Still relayable
FIDO2 key	Yes	Loss, recovery

7. NIST SP 800-63B revision 4, published 31 July 2025, classifies restricted authenticators and requires that risks of the public switched telephone network be assessed; SMS remains permitted but discouraged.
8. Biometrics are convenient and have real limits. They are not secrets, they cannot be revoked, and matching is probabilistic with a false accept rate and a false reject rate that trade against each other.
9. Apple published a figure of roughly 1 in 50,000 false accept for Touch ID and 1 in 1,000,000 for Face ID, per attempt, with limited retries.
10. The right mental model: on a phone, the biometric unlocks a key stored in secure hardware locally. It is a convenient local gate, not an identity sent over the network.
11. Session management matters as much as login. Rotate session identifiers on privilege change, set HttpOnly, Secure and SameSite on cookies, and give tokens the shortest lifetime the product can tolerate.
12. Account recovery is where most authentication systems actually fail. If a help desk can reset MFA after a phone call, the strength of your authenticator is irrelevant. The 2023 MGM Resorts intrusion is the textbook case.

■ 44.6.6 WORDS – remember these

Authentication — proving who you are — verification of a claimed identity against registered authenticators.

Factor — a category of evidence — knowledge, possession or inherence.

TOTP — the app’s six-digit code — time-based one-time password, RFC 6238, derived from a shared secret and the clock.

SIM swap — hijacking your phone number — social engineering a carrier into porting a number to an attacker's SIM.

MFA fatigue — approving to stop the buzzing — repeated push prompts intended to wear the user down.

WebAuthn — the browser standard for hardware login — W3C API binding a signature to a challenge and an origin.

Passkey — a synchronized hardware-grade credential — discoverable WebAuthn credential held by an OS or password manager.

Origin binding — the key checks the address — protocol-level inclusion of the verified origin in the signed data.

44.7 Authorization and access control

■ 44.7.1 PLAIN - in simple words

1. Authentication asks who you are. **Authorization** asks what you may do.
2. They are different questions, they fail differently, and mixing them up causes a large share of real breaches.
3. Being logged in is not permission. Every request must be checked, not just the first one.
4. There are four common ways to decide.
5. **Discretionary access control**: the owner of a thing decides who may use it. This is how files on your laptop work.
6. **Mandatory access control**: a central policy decides, and even the owner cannot override it. This is how classified military systems work.
7. **Role-based access control**: permissions attach to roles, and people get roles. Editor, viewer, administrator.
8. **Attribute-based access control**: a rule engine decides from facts about the user, the resource, the action and the context.
9. Two principles run through all of them.
10. **Least privilege**: every person and every program gets the minimum access needed to do its job, and no more.
11. **Separation of duties**: the person who requests a payment is not the person who approves it.
12. And one classic trap, the **confused deputy**: a program with more privilege than you does something on your behalf that you were not allowed to ask for.

■ 44.7.2 PLAIN - a picture in your head

1. A hospital. Authentication is the badge reader at the door confirming you are Dr Sharma.
2. Authorization is the separate question of which wards, which drug cabinets and which patient records Dr Sharma may open.
3. Discretionary control is a doctor lending their office key to a colleague.
4. Mandatory control is the pharmacy safe, where policy overrides everyone, including the head of the department.
5. Role-based control is the badge saying "consultant", and consultants have a fixed list of doors.
6. Attribute-based control is a rule saying: a doctor may open a record if they are the treating clinician, on shift, in this hospital, today.
7. Least privilege is not giving the night cleaner a master key to the drug store.
8. Separation of duties is two signatures for a controlled substance.

9. The confused deputy is the porter who is trusted to enter any room, being asked by a stranger to fetch a file from a locked office. The porter has the right; the stranger did not.

Where this comparison breaks: a hospital has physical friction, and a human porter might feel that a request is odd. Software has no instinct. A service with broad rights will carry out a strange request a million times a second without hesitating once.

■ 44.7.3 PLAIN – a worked example

1. Here is a real least-privilege event from the running example in this book.
2. The reader's GitHub personal access token could push code, but one push was refused because the change touched a CI workflow file and the token did not carry the workflow scope.
3. That refusal is authorization working exactly as designed.
4. The token was authenticated perfectly. GitHub knew whose token it was. It simply was not authorized for that class of change.
5. Notice the security value. If that token had leaked, an attacker could have pushed ordinary code, which review would catch, but could not have silently rewritten the pipeline that builds and deploys.
6. Now a broken example, stated as a pattern rather than a recipe. A web application shows an invoice at a path containing an invoice number, and the server checks that you are logged in but never checks the invoice belongs to you.
7. Changing the number in the address bar then shows somebody else's invoice. That is an insecure direct object reference, and it is the single most common serious web flaw found in real assessments.
8. The fix is not to hide or scramble the number. The fix is that every lookup asks the database for the invoice with that number and that owner.
9. Compare the two designs in one table.

Design	Check performed	Result
Broken	Is user logged in	Any invoice visible
Correct	Is invoice user's	Only own invoices

■ 44.7.4 PLAIN – what is really happening inside

1. Access decisions have a standard shape with two named parts.
2. The **policy decision point** works out the answer. The **policy enforcement point** actually blocks or allows the request.
3. If the enforcement point sits only in the user interface, you have no security at all. Hiding a button does not remove the endpoint behind it.
4. Enforcement belongs on the server, as close to the data as possible.
5. The strongest pattern is to make the check impossible to forget: scope every query by owner at the data layer, so an unscoped query cannot even be written.
6. Row-level security in PostgreSQL, or a mandatory tenant filter in the object-relational mapper, both achieve this.
7. Confused deputy in software, concretely. Your service holds cloud credentials that can read any file in a bucket. A user supplies a file name. Your service reads it and returns it.
8. The user has borrowed your privilege. The service was authorized; the user was not. Server-side request forgery is the same disease with network access instead of file access.

9. The cure is to never let untrusted input select the target of a privileged operation directly. Map user input through a list of allowed values that you control, and check ownership before acting.
10. Capability-based systems attack the problem from the other side. Instead of saying “who are you, and are you allowed”, you hold an unforgeable token that already names exactly what you may do.
11. A pre-signed cloud storage URL is a capability. It carries its own authority, is scoped to one object and one action, and expires.

■ 44.7.5 TECHNICAL – the engineer’s version

1. The Bell-LaPadula model, published in 1973 by David Bell and Leonard LaPadula, formalized mandatory confidentiality: no read up, no write down.
2. The Biba model, 1977, is its integrity mirror: no write up, no read down.
3. Role-based access control was formalized by David Ferraiolo and Richard Kuhn in 1992 and standardized as ANSI INCITS 359-2004.
4. Attribute-based access control is described in NIST SP 800-162 (2014). XACML is the long-standing policy language; Open Policy Agent with its Rego language is the common modern implementation.
5. The confused deputy problem was named by Norm Hardy in a 1988 paper about a compiler that could be tricked into overwriting its own billing file.
6. In cloud practice: AWS IAM policies, Azure RBAC and Google Cloud IAM are all attribute-flavoured RBAC. The recurring failure is an over-broad wildcard in the action or resource field.
7. OAuth 2.0, RFC 6749 (2012), is a delegation framework, not an authentication protocol. OpenID Connect layers authentication on top of it. Confusing the two produces real vulnerabilities.
8. Scopes in OAuth are coarse permissions granted to an application. They are not a substitute for per-object ownership checks inside your service.
9. Broken Access Control is ranked A01 in the OWASP Top 10 for both the 2021 and 2025 editions, and in the 2025 edition server-side request forgery was folded into it.
10. Practical controls to implement: deny by default, centralize the authorization function rather than scattering conditionals, log every denial, and write tests that assert a user cannot reach another user’s objects.
11. Commands and tools to observe: `aws iam simulate-principal-policy` for AWS, `kubectl auth can-i` for Kubernetes, `getfacl` and `ls -l` for POSIX file permissions, `getenforce` and `ausearch -m avc` for SELinux decisions.
12. On Linux, capabilities split the historic all-or-nothing root privilege into pieces such as `CAP_NET_BIND_SERVICE`. Granting only that is far better than running a web server as root to bind port 80.

■ 44.7.6 WORDS – remember these

Authorization — what you are allowed to do — the decision about permitted actions on resources, distinct from identity.

Least privilege — only what the job needs — granting the minimum rights required, for the minimum time.

Separation of duties — two people, two steps — splitting a sensitive operation so no single actor can complete it.

RBAC — permissions through job titles — role-based access control, ANSI INCITS 359-2004.

ABAC — permissions from rules and facts — attribute-based access control, NIST SP 800-162.

Confused deputy — borrowing someone else's power — a privileged component misused by a less privileged caller.

IDOR — changing the number in the URL — insecure direct object reference, a missing per-object ownership check.

Policy enforcement point — where the block happens — the component that actually denies or permits a request.

44.8 The web application vulnerability classes, defensively

■ 44.8.1 PLAIN - in simple words

1. Almost every web vulnerability is one of two mistakes.
2. Mistake one: data supplied by a user was treated as instructions.
3. Mistake two: a check that should have happened did not happen on this path.
4. **Injection** is mistake one against a language: SQL, shell, LDAP, XPath.
5. **Cross-site scripting** is mistake one against a browser: your page ends up carrying somebody else's script.
6. **Cross-site request forgery** is a browser being tricked into sending a request it should not, using cookies it already holds.
7. **Broken access control** is mistake two: the user is logged in but nobody checked the thing belongs to them.
8. **Server-side request forgery** is your server being used as a proxy to reach places the user cannot reach.
9. **Path traversal** is a file name from a user escaping upwards out of the directory you intended.
10. **Insecure deserialization** is rebuilding an object graph from untrusted bytes, which in many languages can run code as a side effect.
11. **XML external entities** is an XML parser being told to fetch files or URLs while parsing.
12. **Mass assignment** is a form binder copying every submitted field onto your object, including the one called `is_admin`.

■ 44.8.2 PLAIN - a picture in your head

1. Imagine a clerk who reads out a sentence you wrote on a form, into a machine that obeys sentences.
2. You write your name. The clerk reads "Ravi" and the machine files it.
3. Someone else writes a name that continues into a new sentence, and the machine obeys that too, because the clerk never knew where the name ended.
4. The bad fix is to inspect names for suspicious words. Attackers write sentences you did not think of.
5. The good fix changes the shape of the conversation. The clerk hands the machine the sentence template and the name through a separate slot.
6. The machine now knows, structurally, that the name is a name. It cannot become an instruction no matter what is written in it.
7. That is exactly the difference between building a query by gluing strings together and using a parameterized query.

Where this comparison breaks: the clerk analogy suggests the danger is only in one direction. In reality the same data may pass through several machines with different rules, a database, then a browser, then a shell, and each needs its own correct handling. Data safe for one is not automatically safe for another.

■ 44.8.3 PLAIN – a worked example

1. The flawed pattern for SQL injection, shown so you can recognize it in a code review. This is the shape to reject, not code to run.

```
# WRONG: query text is built from user input
q = "SELECT * FROM users WHERE email = '" + email + "'"
db.execute(q)
```

2. Because the value is glued into the query text, a value containing a quote character can end the string early and the rest is parsed as query structure. Nothing after that is under your control.
3. The correct version, which removes the class of bug entirely.

```
# RIGHT: structure and data travel separately
q = "SELECT * FROM users WHERE email = ?"
db.execute(q, (email,))
```

4. Why this fixes it **by construction**: the database receives the query structure first and compiles it, then receives the value as a typed parameter. The value can never be reinterpreted as syntax.
5. Escaping and filtering are weaker answers. They depend on getting every edge case right, in every encoding, forever. Parameterization does not.
6. The same principle applies to shell commands. Do not build a command string; pass an argument list to the operating system directly, so the shell never parses your data.
7. The real-world consequence of getting this wrong: the MOVEit Transfer incident of May 2023 began with a SQL injection flaw, CVE-2023-34362, which the Cl0p group used at scale. Public tracking put the total above 2,600 organizations and tens of millions of individuals.

■ 44.8.4 PLAIN – what is really happening inside

1. Cross-site scripting comes in three shapes, and the difference matters for the fix.
2. **Stored**: the attacker's text is saved in your database, and every later visitor receives it. Highest impact.
3. **Reflected**: the text comes in on the request and is echoed straight back in the response. Needs a victim to follow a crafted link.
4. **DOM-based**: the server never sees it. Client-side JavaScript reads part of the page address and writes it into the page.
5. The fix is contextual output encoding. Encode when writing out, not when reading in, because the correct encoding depends on where the value lands: HTML body, attribute, JavaScript string, URL or CSS each differ.
6. Modern frameworks encode by default. The vulnerabilities cluster around the escape hatches, the functions with names like `dangerouslySetInnerHTML` or `v-html`.
7. A **Content Security Policy** header is the second layer. It tells the browser which script sources are allowed, so injected inline script does not execute even if it reaches the page.
8. A strict policy uses per-response nonces or hashes rather than a domain allowlist, because allowlists are routinely bypassed through permissive files on the allowed domains.
9. Cross-site request forgery works because browsers attach cookies for a site automatically, whatever page triggered the request.
10. The modern structural defence is the `SameSite` cookie attribute. `Lax` is the default in current Chrome and Firefox and blocks cookies on cross-site POST requests. `Strict` is stronger and sometimes inconvenient.
11. Keep anti-CSRF tokens as well, because `SameSite` behaviour varies across browsers and versions and does not protect against same-site attackers.

■ 44.8.5 TECHNICAL – the engineer’s version

1. The current edition is the **OWASP Top 10:2025**, presented at OWASP Global AppSec on 6 November 2025, built from data across more than 2.8 million applications plus roughly 175,000 CVE-to-CWE mappings.

Rank	2025 category	Change
A01	Broken Access Control	Still first
A02	Security Misconfiguration	Up from 5
A03	Software Supply Chain	New framing
A04	Cryptographic Failures	Down from 2
A05	Injection	Down from 3

2. The remaining five in 2025 are A06 Insecure Design, A07 Authentication Failures, A08 Software or Data Integrity Failures, A09 Security Logging and Alerting Failures, and A10 Mishandling of Exceptional Conditions.
3. Two categories are new or newly framed: Software Supply Chain Failures, and Mishandling of Exceptional Conditions. Server-side request forgery, which had its own slot in 2021, was consolidated into Broken Access Control.
4. Defences by class, stated as the fix a reviewer should insist on.

Class	Fix that works by construction
SQL injection	Parameterized queries
Command injection	Argument arrays, no shell
XSS	Contextual encoding plus CSP
CSRF	SameSite plus tokens
Path traversal	Canonicalize, verify prefix
XXE	Disable DTD and entities
SSRF	Allowlist, block metadata IP

5. Path traversal, CWE-22: resolve the path to its canonical absolute form first, then confirm it starts with the intended base directory. Checking for the literal two-dot sequence before resolving fails against encodings.
6. XXE, CWE-611: disable document type definitions entirely in the parser. In Java that means setting `disallow-doctype-decl` to `true`; in Python use `defusedxml`. Many parsers changed to safe defaults after 2018, but not all.
7. Insecure deserialization, CWE-502: never deserialize untrusted data into arbitrary types. Use a data-only format such as JSON with an explicit schema. Java native serialization, Python pickle, PHP `unserialize` and .NET `BinaryFormatter` are all unsafe on untrusted input by design.
8. Mass assignment, CWE-915: bind to an explicit allowlist of fields, or use a separate data transfer object per endpoint rather than the database model.
9. SSRF, CWE-918: block link-local metadata addresses such as `169.254.169.254`, re-validate after every redirect, and resolve the hostname once and connect to that exact address to avoid time-of-check to time-of-use tricks.
10. Log4Shell, CVE-2021-44228, disclosed 9 December 2021 with a CVSS score of 10.0, was an injection flaw in a logging library: attacker-controlled text that reached a log call could trigger a remote lookup. It is the clearest reminder that logging is also an input-handling path.

11. Tools that observe: zap and Burp Suite for interactive testing, semgrep and CodeQL for static analysis, sqlmap only against systems you own or are authorized in writing to test.

■ 44.8.6 WORDS – remember these

Injection — data treated as instructions — untrusted input reaching an interpreter as part of its command structure.

Parameterized query — structure and data kept apart — precompiled statement with typed placeholders, immune to SQL injection.

XSS — someone else's script on your page — cross-site scripting, stored, reflected or DOM-based.

Content Security Policy — a browser rulebook for scripts — response header restricting which sources may execute.

CSRF — your browser sending a request it should not — forged cross-site request riding on ambient cookies.

SameSite — the cookie's travel restriction — cookie attribute limiting sending on cross-site requests, Lax by default.

SSRF — your server fetching what the user cannot — server-side request forgery against internal or metadata endpoints.

Mass assignment — the form that set `is_admin` — automatic binding of request fields to object properties without an allowlist.

44.9 Memory safety

■ 44.9.1 PLAIN – in simple words

1. Some languages let a program read and write memory outside the space it asked for. C and C++ are the important ones.
2. They do this on purpose. Bounds checking costs time, and these languages were designed when every cycle mattered.
3. A **buffer overflow** happens when a program copies more bytes into a space than that space can hold, and the extra bytes land on whatever is next.
4. What is next, on the stack, is often bookkeeping: saved registers and the address the function will return to.
5. Overwrite that return address and the processor jumps somewhere the programmer never intended when the function finishes.
6. **Use-after-free** is using a pointer to memory that was already released. The allocator may have handed that space to someone else.
7. **Double-free** is releasing the same block twice, which corrupts the allocator's own bookkeeping.
8. **Integer overflow** is a size calculation wrapping around past the largest representable value, producing a tiny allocation for a huge copy.
9. All four are the same underlying story: the program lost track of how big something is or how long it lives.

■ 44.9.2 PLAIN – a picture in your head

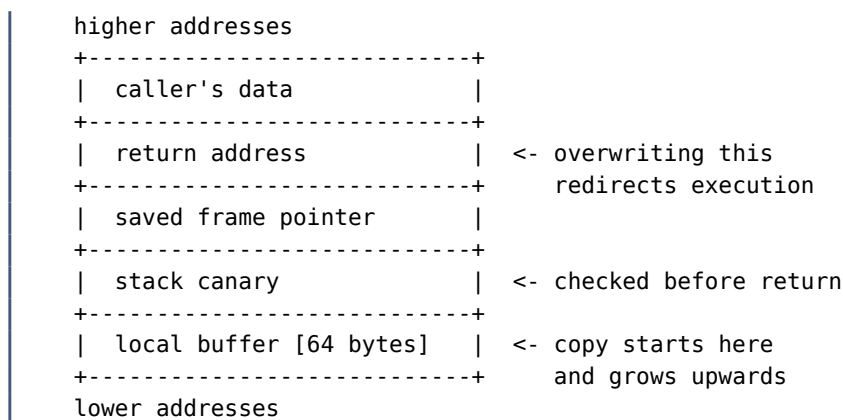
1. Think of a row of numbered pigeonholes on a wall, each one letter wide.
2. You are told to write an eight-letter word into a four-letter hole.

3. Nothing stops you. The last four letters spill into the next holes.
4. Now suppose the hole immediately after yours holds the room number the messenger will visit next.
5. You have just changed where the messenger goes, without touching the messenger.
6. A **stack canary** is a small agreed word placed just before that room number. Before leaving, the messenger checks the word is unchanged. If your spill overwrote it, the program stops.

Where this comparison breaks: real overflows are not always sequential and not always on the stack. Heap corruption works by damaging the allocator's own records, which is subtler and does not pass through a canary at all. And the canary only detects a contiguous overwrite, not a targeted single-word write.

■ 44.9.3 PLAIN – a worked example

1. A stack frame, drawn as it grows downwards in memory on a typical system.



2. The copy starts at the bottom of the buffer and moves up towards the canary, the frame pointer and the return address.
3. So a copy of 80 bytes into a 64-byte buffer reaches exactly the fields that control what happens when the function returns.
4. Integer overflow feeding an allocation, shown as a pattern to reject.

```
/* WRONG: count * size can wrap around */
buf = malloc(count * size);
memcpy(buf, src, count * size);
```

5. If count times size exceeds the maximum of the type, the product wraps to a small number. The allocation succeeds at that small size, and the copy then writes far past it.
6. The correct pattern checks for overflow first, or uses a checked helper such as `calloc`, or uses a language where sizes cannot silently wrap.

■ 44.9.4 PLAIN – what is really happening inside

1. Four defences are standard on modern systems, and each raises cost rather than removing the bug class.
2. **Stack canaries**, from StackGuard in 1998: a random value written before the return address and verified before returning.
3. **Non-executable memory**, called NX or DEP: pages are either writable or executable, not both, so injected data cannot be run as code.
4. Attackers answered with return-oriented programming, which chains together small fragments of code that already exist in the program. Nothing new is injected, so NX does not help.

5. **Address space layout randomization:** load the program and its libraries at unpredictable addresses so the attacker cannot know where anything is.
6. Attackers answer with information leaks that reveal one address, from which the rest follows.
7. **Control flow integrity:** check at run time that every indirect jump goes to a legitimate target. ARM Pointer Authentication and Intel CET are hardware support for this idea.
8. Honest summary: each of these turns a reliable exploit into an unreliable or expensive one. None of them makes the underlying bug disappear.
9. The real fix is a language that cannot express the bug. Rust checks lifetimes and ownership at compile time. Go, Java, C#, Python and JavaScript check bounds at run time and manage memory automatically.
10. That is not free. Garbage collection has pauses, run-time checks cost a few percent, and Rust's rules are hard to learn. But the cost is bounded and the bug class is gone.

■ 44.9.5 TECHNICAL - the engineer's version

1. Scale, from vendor data. Microsoft reported at BlueHat in 2019 that about 70 percent of the CVEs it assigned each year since 2006 were memory safety issues. Google reported a similar proportion for Chrome.
2. Google's Android security team reported that memory safety vulnerabilities fell from about 76 percent of Android vulnerabilities in 2019 to about 24 percent in 2024, as new code was written in Rust and Java rather than C++.
3. Government guidance, with dates. CISA, the NSA, the FBI and international partners published "The Case for Memory Safe Roadmaps" in December 2023.
4. The United States Office of the National Cyber Director published "Back to the Building Blocks: A Path Toward Secure and Measurable Software" in February 2024, explicitly recommending memory-safe languages.
5. The NSA and CISA published "Memory Safe Languages: Reducing Vulnerabilities in Modern Software Development" in June 2025.
6. None of these are laws. They are procurement pressure and expectation setting, and they are already changing what buyers ask for.

Weakness	CWE	Typical impact
Out-of-bounds write	CWE-787	Code execution
Out-of-bounds read	CWE-125	Memory disclosure
Use-after-free	CWE-416	Code execution
Integer overflow	CWE-190	Undersized buffer

7. Heartbleed, CVE-2014-0160, disclosed 7 April 2014, was an out-of-bounds read in OpenSSL's TLS heartbeat handling. It leaked up to 64 kilobytes of process memory per request, including private keys, and it left no trace in ordinary logs.
8. Tools that find these before shipping: AddressSanitizer and MemorySanitizer, UndefinedBehaviorSanitizer, Valgrind, the compiler flags `-fstack-protector-strong` and `-D_FORTIFY_SOURCE=3`, and fuzzing with libFuzzer, AFL++ or OSS-Fuzz.
9. Check a binary's own defences with `checksec --file=./binary` on Linux, or `otool -hv` on macOS. Confirm ASLR is enabled system-wide with `sysctl kernel.randomize_va_space`, where 2 means full randomization.

10. Where experts disagree: whether to rewrite existing C code, or only write new code in safe languages. The rewrite camp cites persistent bug rates; the incremental camp cites the risk of reintroducing logic bugs in mature, heavily tested code. Practice mostly favours the incremental route.

■ 44.9.6 WORDS – remember these

Memory safety — the program cannot stray outside its own memory — guaranteed absence of out-of-bounds and lifetime errors.

Buffer overflow — writing past the end of a space — out-of-bounds write, often CWE-787.

Use-after-free — using something already returned — dereferencing a pointer to released memory, CWE-416.

Stack canary — a tripwire before the return address — random guard value verified before function return.

ASLR — moving the furniture every boot — address space layout randomization.

NX / DEP — memory is writable or runnable, not both — hardware-enforced non-executable pages.

Control flow integrity — checking every jump is legitimate — run-time or hardware validation of indirect branch targets.

Sanitizer — a bug detector built into a test build — instrumentation such as AddressSanitizer that reports violations at run time.

44.10 Network attacks conceptually

■ 44.10.1 PLAIN – in simple words

1. Three network attacks matter for a working engineer.
2. **Man-in-the-middle**: someone sits between you and the server, reading and possibly changing what passes.
3. **Spoofing**: someone lies about an address, so your traffic goes to them instead of the real destination.
4. **Denial of service**: someone sends so much traffic or so many expensive requests that the real users cannot get through.
5. TLS answers the first one. It encrypts, so a listener sees nothing useful, and it authenticates the server by certificate, so an impostor cannot pass.
6. Chapter 32 covered how that certificate check works. The one-line version: the server proves it holds the private key for a name signed by an authority your device already trusts.
7. Spoofing works at two levels. On your local network, **ARP spoofing** lets a machine claim to be the router.
8. On the wider internet, **DNS spoofing** makes a name resolve to the wrong address.
9. Denial of service comes in three flavours: raw volume, connection exhaustion, and expensive requests that cost the server far more than the attacker.
10. **Amplification** is the nastiest volume trick: send a small question with a faked return address to a public server, and it sends a large answer to the victim.

■ 44.10.2 PLAIN – a picture in your head

1. Picture a post room in an office building.
2. ARP spoofing is somebody standing up and repeatedly announcing “I am the mail chute”, until everyone believes them and hands over their post.
3. There is no verification step in the announcement, by design. The protocol simply believes the last thing it heard.
4. DNS spoofing is somebody rewriting the internal phone directory, so calls for accounts reach a different desk.
5. Amplification is posting a one-line letter that says “send me your full catalogue”, with the victim’s return address on it, to ten thousand companies at once.
6. Each catalogue is heavy, none of them is your problem, and all of them arrive at the victim.

Where this comparison breaks: office post is slow and physical, so the flood is visible. On a network the same trick delivers terabits per second within seconds, and there is no time for a human to react. That is why the defences must be automatic.

■ 44.10.3 PLAIN – a worked example

1. Amplification factors, measured by researchers and widely published.

Service	Amplification	Notes
DNS open resolver	about 28 to 54x	Very common
NTP monlist	about 556x	Mostly patched
memcached	up to 51,000x	UDP port 11211

2. Read the last row. A single byte sent can produce tens of thousands of bytes delivered to the victim.
3. That is not theoretical. On 28 February 2018 GitHub absorbed a 1.35 terabit per second memcached amplification attack, the record at the time. It was mitigated within about ten minutes by routing traffic through a scrubbing provider.
4. Scale has moved a long way since. Cloudflare reported mitigating attacks of 22.2 terabits per second in September 2025, 29.7 in October 2025, and 31.4 during December 2025. All were attributed to the Aisuru botnet family, running largely on compromised consumer devices.
5. The December record lasted about 35 seconds. There is no human response time available in that window.
6. Attacks are not only about volume. In October 2023 the HTTP/2 Rapid Reset technique, CVE-2023-44487, produced request-rate records by opening and immediately cancelling streams, costing the server far more than the client.

■ 44.10.4 PLAIN – what is really happening inside

1. ARP has no authentication. A device asks “who has 192.168.0.1”, and any machine on that network segment can answer. Most systems accept unsolicited answers and cache them.
2. That is why untrusted local networks matter, and why a VPN or plain TLS everywhere is the practical answer for a laptop in a cafe.
3. DNS cache poisoning aims to get a false answer accepted and cached by a resolver, so every later user of that resolver is sent to the wrong place.
4. Classic DNS used a 16-bit transaction identifier and no signatures, which Dan Kaminsky showed in July 2008 was far too little entropy.

5. The immediate fix was source port randomization, which multiplied the guessing space. The structural fix is DNSSEC, which signs records.
6. The reader's own machine uses 1.1.1.1 as its resolver rather than the router. That reduces exposure to a compromised or mis-configured home router, and it moves trust to Cloudflare instead. It is a trade, not a removal of trust.
7. Encrypted DNS, DNS over HTTPS in RFC 8484 and DNS over TLS in RFC 7858, hides the query from the local network and from the internet provider. It does not prove the answer is correct; only DNSSEC does that.
8. For denial of service, the defences are layered and none works alone.
9. **Rate limiting** by address, by account and by expensive operation.
10. **Filtering** at the network edge, dropping obviously forged or malformed packets before they reach an application.
11. **Anycast**, where the same address is announced from many locations, so an attack is split across dozens of data centres instead of hitting one.
12. **Scrubbing**, where traffic is diverted through a provider that removes attack packets and forwards the rest.
13. And the structural fix nobody can do alone: source address validation at the internet provider, described in BCP 38, published as RFC 2827 in 2000. Amplification only works because forged source addresses still get forwarded.

■ 44.10.5 TECHNICAL – the engineer's version

1. What TLS does and does not do against interception. It authenticates the server's name and protects the channel. It does not protect against a client that has been made to trust an extra certificate authority, which is how corporate interception proxies work, as Chapter 32 described.
2. Certificate Transparency, RFC 6962 and its successor RFC 9162, provides public append-only logs so that mis-issued certificates for your domain can be detected after the fact. Monitoring those logs is a real control.
3. HSTS, RFC 6797, removes the plaintext first request. Preloading puts your domain in a list shipped inside browsers.
4. Layer 2 defences: dynamic ARP inspection and DHCP snooping on managed switches, and 802.1X port authentication for wired and wireless access.
5. DNS defences: DNSSEC validation on the resolver, DNS cookies in RFC 7873, and response rate limiting on authoritative servers.
6. Denial of service categories map to network layers. Volumetric floods saturate bandwidth, protocol attacks such as SYN floods exhaust connection state, and application-layer attacks exhaust processing.
7. SYN cookies, invented by Daniel J. Bernstein in 1996, remove the need to keep state for half-open connections. Enable with `sysctl net.ipv4.tcp_syncookies=1` on Linux.
8. Observation commands: `tcpdump -i any -n` and Wireshark for packet capture, `arp -a` to see the current address table, `dig +dnssec example.com` to check signature records, `mttr` for continuous path measurement, and `ss -s` for socket state counts under load.
9. Historical scale markers, for calibration.

Year	Event	Peak
2016	Mirai, Dyn and OVH	about 1.1 Tbps
2018	GitHub memcached	1.35 Tbps
2025	Aisuru, December	31.4 Tbps

10. Mirai was notable for what it infected: consumer routers and cameras with factory default passwords. Its source code was published in September 2016, which is why variants are still active a decade later.

■ 44.10.6 WORDS - remember these

Man-in-the-middle — someone sitting between you and the server — an active interception attack on a communication channel.

ARP spoofing — claiming to be the router — forged address resolution replies poisoning a local network's address cache.

DNS cache poisoning — putting a lie in the phone book — injecting a forged record that a resolver stores and reuses.

Amplification — a small question, a huge answer — reflection attack exploiting protocols with large response-to-request ratios.

Anycast — one address, many locations — routing technique spreading traffic across distributed sites.

Scrubbing — cleaning the traffic elsewhere — diverting traffic through a provider that filters attack packets.

BCP 38 — stop forged source addresses at the door — RFC 2827 ingress filtering, the structural cure for amplification.

SYN cookies — answering without remembering — stateless response encoding that defeats half-open connection floods.

44.11 Malware and how infection actually happens

■ 44.11.1 PLAIN - in simple words

1. **Malware** is any program written to work against the interests of the person whose machine it runs on.
2. A **virus** attaches itself to another file and spreads when that file is run by a person.
3. A **worm** spreads by itself across a network, needing no human at all.
4. A **trojan** pretends to be something useful so you install it willingly.
5. **Ransomware** encrypts your files and demands payment for the key.
6. **Spyware** watches you: keystrokes, screenshots, messages, location.
7. A **rootkit** hides other malware by lying to the operating system about what files and processes exist.
8. A **bootkit** goes lower still and loads before the operating system does, so the system it lies to never sees the truth.
9. A **botnet** is many infected machines taking orders from one place.
10. A **cryptominer** quietly spends your electricity and processor on mining.
11. **Fileless malware** never writes a program to disk; it lives in memory and in legitimate system tools, so file scanning finds nothing.

■ 44.11.2 PLAIN - a picture in your head

1. Think of a building and the ways an unwanted person gets in.
2. Someone posts a package that a staff member opens: that is a malicious attachment.

3. Someone phones pretending to be head office and gets a door code: phishing.
4. A supplier's cleaning contractor is bribed, and their key still works: supply chain compromise.
5. A window at the back has been unlocked for two years because nobody checked: an unpatched internet-facing service.
6. Someone leaves a labelled envelope in the car park hoping it gets carried inside: removable media.

Where this comparison breaks: a building has one entrance list. A modern company has thousands of accounts, hundreds of cloud services and dozens of suppliers, and no single person knows all of them. The unlocked window is usually one nobody remembered existed.

■ 44.11.3 PLAIN - a worked example

1. Realistic infection routes, ranked by how often they appear in incident reports, with the control that actually helps.

Route	Frequency	Control that helps
Phishing / stolen login	Very high	Phishing-resistant MFA
Unpatched public service	High	Patch within days
Malicious attachment	High	Macro blocking, sandbox
Supply chain	Growing	Pinning, provenance
Drive-by download	Medium	Browser updates
Removable media	Low	Device control policy

2. Ransomware in particular follows a pattern that has barely changed since 2019: get in, look around quietly for days or weeks, steal a copy of the data, find and destroy the backups, then encrypt everything at once.
3. Stealing the data first is deliberate. It creates a second threat, to publish, so that having good backups is no longer a complete answer.
4. Chainalysis reported that tracked ransomware payments fell about 35 percent in 2024, to roughly 813 million dollars, as more victims refused to pay.
5. The FBI Internet Crime Complaint Center reported 20.9 billion dollars in total reported losses for 2025, from more than one million complaints, in its annual report published in April 2026.

■ 44.11.4 PLAIN - what is really happening inside

1. Detection works in three different ways, and each fails differently.
2. **Signatures:** match known bad bytes or known bad file hashes. Precise, no false alarms, and useless against anything new or repacked.
3. **Heuristics:** score suspicious structure, such as packed code that unpacks itself, or a document containing a macro that reaches the network.
4. **Behaviour:** watch what the program does at run time. A process that enumerates files, opens each one, writes back encrypted content and deletes shadow copies is ransomware regardless of what its bytes look like.
5. **EDR**, endpoint detection and response, is the behavioural approach with central recording. It keeps a timeline of process launches, network connections and file changes so an analyst can reconstruct what happened.
6. That recording is often more valuable than the blocking. Without it you cannot answer the only question that matters after an incident: what did they touch?
7. Why antivirus alone is not enough: it is mostly signature and heuristic, it sees one machine, and it cannot judge whether a legitimate administrator tool is being used legitimately.

8. Attackers exploit exactly that gap with **living off the land**: using PowerShell, certutil, wmic, rundll32 and remote management tools that are already installed and already trusted.
9. For ransomware, the honest advice is unglamorous. Backups are the only reliable answer, and only if they are tested, versioned, and stored where the production credentials cannot delete them.
10. The rule of thumb is three copies, on two kinds of media, with one held offline or immutable. A backup that a compromised administrator account can erase is not a backup.

■ 44.11.5 TECHNICAL – the engineer’s version

1. History markers. The Morris worm of 2 November 1988 infected several thousand machines and led to the founding of the first CERT.
2. ILOVEYOU spread by email in May 2000. Code Red and Nimda hit in 2001. SQL Slammer in January 2003 doubled its population every 8.5 seconds.
3. Stuxnet, discovered June 2010, used four zero-day flaws and stolen code-signing certificates to damage centrifuges. WannaCry, 12 May 2017, and NotPetya, 27 June 2017, both used the Eternal-Blue SMB exploit; NotPetya caused damages widely estimated above 10 billion dollars.
4. Persistence mechanisms to look for on a compromised host: scheduled tasks, systemd units, launch agents, registry Run keys, WMI event subscriptions, browser extensions, and shell profile files.
5. Firmware and boot-level threats are real but rare. LoJax in 2018 was the first UEFI rootkit found in the wild; BlackLotus in 2023 was the first observed to bypass UEFI Secure Boot on patched systems.
6. Ransomware defensive controls that measurably work: phishing-resistant MFA on all remote access, patching internet-facing services within days, network segmentation so one workstation cannot reach every server, least-privilege service accounts, and immutable backups.
7. Tools that observe: ps, lsof, netstat, Sysinternals Process Explorer and Autoruns on Windows, osquery for fleet inspection, YARA for writing your own detection rules, and VirusTotal for reputation checks on hashes.
8. Never analyze suspicious samples on a working machine. Use an isolated virtual machine with no network path back to anything you care about.

■ 44.11.6 WORDS – remember these

Worm — spreads without a human — self-propagating malware exploiting network services directly.

Trojan — useful on the outside, hostile inside — malware disguised as legitimate software.

Ransomware — your files held hostage — encrypting malware, now usually combined with data theft and threatened publication.

Rootkit — malware that hides other malware — code that subverts the operating system’s view of files and processes.

Fileless malware — nothing on disk to scan — in-memory execution using existing signed system tools.

EDR — a flight recorder for endpoints — endpoint detection and response, behaviour-based with central telemetry.

Living off the land — using your own tools against you — abuse of legitimate administrative binaries to avoid detection.

44.12 Social engineering

■ 44.12.1 PLAIN – in simple words

1. Most breaches do not start with clever code. They start with a person being helpful.
2. **Phishing** is a message designed to make you click, log in or pay.
3. **Spear phishing** is the same, tailored to you personally, using details from your company website and social media.
4. **Pretexting** is inventing a believable story: a new supplier, an audit, a delivery problem, a colleague locked out before a big meeting.
5. **Business email compromise** is a targeted fraud in which someone with apparent authority instructs a payment or a change of bank details.
6. **Vishing** is the same by voice call. **Smishing** is the same by text message.
7. **MFA fatigue** is pushing approval prompts until someone approves one.
8. Every one of these works by putting a person under pressure and giving them a way to make the pressure stop.
9. The three levers are always the same: authority, urgency, and a plausible reason not to follow the normal process.

■ 44.12.2 PLAIN – a picture in your head

1. Imagine a new employee on their third day.
2. An email arrives from the finance director, who is travelling, about a confidential acquisition, asking for a transfer today, before close.
3. Every element is designed to disable the normal check: seniority, secrecy, time pressure, and a reason the usual approver is unreachable.
4. The employee is not stupid. They are being professional in a situation engineered so that professionalism produces the wrong action.
5. That is why blaming users is both unkind and useless. The attack was designed against the process, using the person as the tool.

Where this comparison breaks: the picture suggests one dramatic moment. Most real social engineering is patient and boring, a few emails over weeks establishing that the sender is a normal part of the working day.

■ 44.12.3 PLAIN – a worked example

1. Real costs, from published figures.
2. The FBI Internet Crime Complaint Center recorded 2.77 billion dollars in business email compromise losses in 2024 and 3.04 billion dollars in 2025.
3. Phishing remained the most-reported crime type in the 2025 report, with 191,561 complaints and 215.8 million dollars in direct losses.
4. The September 2022 Uber intrusion combined a purchased contractor password, repeated push prompts, and a message claiming to be internal IT support.
5. The September 2023 attacks on MGM Resorts and Caesars Entertainment began with phone calls to help desks, persuading staff to reset credentials and MFA for real employees. MGM disclosed an effect of roughly 100 million dollars on quarterly results.
6. Recognition signs worth teaching, in plain language.

Sign	What it looks like
Urgency	Must be done today
Authority	From a senior person
Secrecy	Do not tell the team
Process bypass	Skip the usual approval
Channel switch	Move to WhatsApp or SMS
New bank details	Same supplier, new account

■ 44.12.4 PLAIN – what is really happening inside

1. Attackers do reconnaissance first: company website, LinkedIn, job adverts, conference talks, code repositories, and out-of-office replies.
2. Job adverts leak the technology stack. Out-of-office replies leak who is away and who covers for them. Both are used directly.
3. Domain look-alikes are cheap. A domain differing by one character, or using a different top-level domain, costs a few dollars and reads correctly to a tired human.
4. Since 2023, generative AI has removed the last easy signal: bad grammar. Convincing text in any language, and cloned voices from short public audio samples, are both established capability now, not speculation.
5. So detection must move away from “does it look wrong” to “is this the normal process”.
6. The organizational answers, in order of effectiveness.
7. Phishing-resistant authentication, meaning FIDO2 or passkeys, which removes credential phishing as a category rather than reducing it.
8. Out-of-band verification for money and for access: any change of bank details or any MFA reset is confirmed by calling a number already on file, never a number in the message.
9. A named, blameless reporting route, with a target of reporting in minutes. The measure that matters is time to report, not click rate.
10. Removing single points of authority, so no one person can move money or reset an executive’s access alone.
11. Help desk identity proofing: for high-privilege accounts, require a verification step the caller cannot talk their way around, such as a manager callback or an in-person or video check against a record.

■ 44.12.5 TECHNICAL – the engineer’s version

1. Email authentication is the technical floor. SPF, RFC 7208, lists which servers may send for a domain. DKIM, RFC 6376, signs messages. DMARC, RFC 7489, ties the two to the visible From address and sets a policy.
2. Publish DMARC with `p=reject` once alignment is confirmed. A monitoring-only policy of `p=none` left in place for years is a common half-measure.
3. These stop exact-domain spoofing. They do not stop look-alike domains or compromised legitimate accounts, which is where most real fraud now lives.
4. Add external sender banners, attachment sandboxing, link rewriting with time-of-click checks, and lookalike-domain monitoring through certificate transparency logs.
5. MITRE ATT&CK identifiers to use in reporting: T1566 phishing, T1566.001 spearphishing attachment, T1598 phishing for information, T1621 multi-factor authentication request generation.
6. Training works only in a specific form: short, frequent, role-relevant, and focused on process rather than on spotting fakes. Punitive simulated phishing programmes reliably reduce reporting, which makes things worse.

7. Measure two things: median time from first delivery to first report, and the proportion of high-risk transactions confirmed out of band.

■ 44.12.6 WORDS - remember these

Phishing — a message engineered to make you act — social engineering by email or messaging, MITRE T1566.

Spear phishing — phishing written just for you — targeted variant using researched personal or organizational detail.

Pretexting — a believable cover story — fabricated scenario used to justify an unusual request.

Business email compromise — fraud by fake authority — payment or data fraud using impersonated or compromised business accounts.

Vishing and smishing — the phone and text versions — voice and SMS social engineering.

Out-of-band verification — confirm on a different channel — validating a request using contact details held independently of the request.

DMARC — the policy that ties email checks together — RFC 7489 alignment and enforcement policy over SPF and DKIM.

44.13 Supply chain security

■ 44.13.1 PLAIN - in simple words

1. Your program is mostly other people's code. A typical application pulls in hundreds of packages, and those pull in more.
2. You trust every one of them, and everyone who can publish an update to any of them.
3. **Dependency confusion**: your build tool asks for an internal package name and a public registry answers first, with a package somebody else uploaded.
4. **Typosquatting**: a package with a name one character away from the one you meant, waiting for a typing mistake.
5. **Compromised maintainer**: a real package, a real author, a stolen account.
6. **Malicious update**: version 3.4.1 is fine and 3.4.2 is not, and your build accepted it automatically because you asked for "the latest".
7. The uncomfortable part is that this attacks the trusted path. The code is signed, the download is over TLS, and the certificate is valid. Everything is correct except the contents.

■ 44.13.2 PLAIN - a picture in your head

1. Think of a kitchen buying ingredients from many suppliers.
2. Typosquatting is a sack labelled almost like your usual flour brand.
3. Dependency confusion is a stranger delivering to your door claiming to be your in-house supplier, and your staff accepting because the name matched.
4. Compromised maintainer is your genuine supplier, whose delivery van was taken over last night.
5. A lock file is writing down the exact batch number you accepted, so tomorrow's delivery must match or be questioned.
6. An SBOM is the ingredient list on the finished product, so when a contamination notice is issued you can tell in minutes whether you used it.

Where this comparison breaks: a kitchen can smell bad flour. Malicious code is designed to behave normally until a condition is met, and often only in the published archive rather than in the source repository everyone reads.

■ 44.13.3 PLAIN – a worked example

1. Four real incidents, with the mechanism and the lesson.

Incident	Year	Mechanism
event-stream	2018	Handed-over maintainership
SolarWinds	2020	Compromised build system
Codecov	2021	Altered uploader script
xz-utils	2024	Multi-year social engineering

2. event-stream, November 2018: a popular npm package's maintainer handed the project to a volunteer who then added a dependency containing code targeting a specific cryptocurrency wallet application. Downloads were in the millions per week.
3. SolarWinds, disclosed December 2020: attackers compromised the build pipeline for Orion and inserted the SUNBURST backdoor into signed releases. Roughly 18,000 customers installed it; a much smaller number were then targeted. The signature was valid because the build itself was subverted.
4. Codecov, April 2021: a flaw in a Docker image creation process leaked credentials, which were used to modify the Bash Uploader script. The change sat undetected from 31 January to 1 April 2021 and exfiltrated environment variables, meaning secrets, from users' CI runs.
5. xz-utils, March 2024: CVE-2024-3094, CVSS 10.0. An account building trust since 2021 became a co-maintainer and placed a backdoor targeting OpenSSH in release tarballs of versions 5.6.0 and 5.6.1. Andres Freund found it on 29 March 2024, from half a second of extra CPU time on SSH logins.
6. Read that last sentence again. The most sophisticated open-source supply chain attack yet found was caught by one engineer noticing a performance anomaly, not by any scanner.

■ 44.13.4 PLAIN – what is really happening inside

1. Dependency confusion, explained defensively, was published by Alex Birsan in February 2021 after he reached more than 35 companies including Apple and Microsoft, earning over 130,000 dollars in bug bounties.
2. The mechanism is resolution order. Many package managers, given a name, consult both an internal registry and the public one and prefer the highest version number. Publishing a very high version publicly wins.
3. The fixes are configuration, not vigilance: scope internal packages under a namespace you own, configure the client to use exactly one source per scope, and reserve your internal names publicly so nobody else can take them.
4. Lock files record the exact resolved version and a hash of each package's contents. `package-lock.json`, `poetry.lock`, `Cargo.lock`, `go.sum`.
5. Install in continuous integration with the command that refuses to update the lock file, such as `npm ci` rather than `npm install`.
6. Pinning to exact versions removes surprise updates but creates a second duty: something must tell you when a pinned version becomes vulnerable.

7. An **SBOM**, software bill of materials, is a machine-readable list of every component and version in a build. Its value is answering “are we affected” in minutes rather than weeks, which was the actual pain of Log4Shell.
8. Signing and provenance answer a different question: not what is inside, but who built it, from which source, on which machine.

■ 44.13.5 TECHNICAL – the engineer’s version

1. SBOM formats: SPDX, an ISO standard as ISO/IEC 5962:2021, and CycloneDX from OWASP. United States Executive Order 14028 of 12 May 2021 pushed SBOMs into federal procurement, which is why they spread.
2. Sigstore, launched in 2021 under the Open Source Security Foundation, provides keyless signing: a short-lived certificate is issued against an OpenID Connect identity, the signature is recorded in the public Rekor transparency log, and the key is discarded. cosign is the common tool.
3. SLSA, Supply-chain Levels for Software Artifacts, defines build integrity levels. Level 3 requires a hardened, non-falsifiable build service that produces signed provenance describing exactly what it built and from where.
4. npm provenance, generally available since 2023, publishes signed statements linking a published package to the exact commit and workflow that built it.
5. Reproducible builds mean identical source yields byte-identical output, so independent parties can rebuild and compare. Debian has driven this since 2013 and reports well over 90 percent reproducibility for its main archive.
6. Practical checklist for a team, in priority order: commit and honour lock files; run `npm audit`, `pip-audit`, `cargo audit` or `Dependabot` in CI; pin base container images by digest, not tag; scope internal packages; generate an SBOM per release; require signed provenance for anything you deploy.
7. Software Supply Chain Failures is A03 in the OWASP Top 10:2025, promoted and broadened from the 2021 category on vulnerable and outdated components.
8. Tools to observe: `syft` to generate an SBOM, `grype` or `trivy` to scan it, `cosign verify` to check signatures, and `osv-scanner` against the Open Source Vulnerabilities database.

■ 44.13.6 WORDS – remember these

Dependency confusion — the public registry answering an internal name — a resolution-order attack on package managers.

Typosquatting — a package named almost like yours — malicious package relying on a typing or memory error.

Lock file — the exact batch numbers you accepted — a record of resolved versions and content hashes for reproducible installs.

SBOM — the ingredient list — machine-readable inventory of components, in SPDX or CycloneDX format.

Provenance — who built it, from what — signed attestation linking an artifact to its source and build process.

Sigstore — signing without long-lived keys — keyless signing with certificate issuance and a public transparency log.

Reproducible build — same source, same bytes — property allowing independent verification that a binary matches its source.

44.14 Defence in depth for a real system

■ 44.14.1 PLAIN – in simple words

1. **Defence in depth** means assuming each control will fail and putting another one behind it.
2. **Network segmentation** divides a network into zones so a compromised laptop cannot reach the database directly.
3. A **firewall** decides which traffic may pass between zones. Default deny, with a short list of allowed paths, is the only sane setting.
4. **Zero trust** means dropping the idea that being inside the network proves anything. Every request is authenticated and authorized on its own merits.
5. **Secure boot** makes the machine refuse to load unsigned early software.
6. A **TPM** is a small chip that stores keys and measurements the main processor cannot forge.
7. **Measured boot** records a hash of each stage into the TPM, so the machine can later prove which software it actually booted.
8. **Disk encryption** protects data on a powered-off device. It does nothing at all once the machine is running and unlocked.
9. **Sandboxing** and containers limit what a compromised process can reach.
10. And the highest-value activity of all, which almost nobody does properly: **patching**, quickly and everywhere.

■ 44.14.2 PLAIN – a picture in your head

1. A castle is the traditional image, and it is the wrong one, because it suggests one wall with everything soft inside.
2. Use a ship instead. A ship has watertight compartments.
3. A hole in one compartment floods that compartment. The ship stays up.
4. Segmentation is the bulkheads. Least privilege is each compartment holding only what belongs in it.
5. Zero trust is the rule that a door between compartments needs a valid reason every single time, not just because you are already aboard.

Where this comparison breaks: bulkheads are physical and permanent. Network segments are configuration, and configuration drifts. The temporary firewall rule opened for a migration in 2022 is still open, and nobody knows why.

■ 44.14.3 PLAIN – a worked example

1. What disk encryption actually protects against, stated honestly.

Situation	Protected?
Laptop stolen, powered off	Yes
Laptop stolen, sleeping	Often not
Malware while you use it	No
Cloud provider reads disk	Yes
Someone with your password	No

2. That table is the whole point. FileVault on macOS, BitLocker on Windows and LUKS on Linux all defend the same narrow, real, valuable case: physical loss of a powered-off device.

3. The key is normally sealed to the TPM or Secure Enclave and released only if measured boot matches. That is why changing firmware settings can trigger a recovery key prompt: the measurements changed, so the seal did not open.
4. Sandboxing and containers, honestly. A container is a process with restricted namespaces and control groups, sharing one kernel.
5. That is a real boundary and a useful one. It is not the same boundary as a virtual machine, because one kernel flaw crosses it.
6. Where isolation must be strong, use virtual machines, or lightweight hypervisor-backed sandboxes such as Firecracker or gVisor.

■ 44.14.4 PLAIN – what is really happening inside

1. Zero trust in practice is three concrete changes, not a product.
2. One: identity is checked at every request, for users and for services alike.
3. Two: device health is part of the decision. Is the disk encrypted, is the operating system current, is the endpoint agent reporting?
4. Three: access is per-application rather than per-network, so being on the VPN does not grant everything on the subnet.
5. Google's BeyondCorp, published from 2014 onwards, is the reference implementation. NIST SP 800-207, published August 2020, is the reference architecture.
6. Patching is the highest-value activity because most successful intrusions use a flaw that was fixed months earlier.
7. Measure two numbers: how long from a fix being published to it being installed on internet-facing systems, and what fraction of your fleet is current. Days and percentages, not intentions.
8. CISA's Known Exploited Vulnerabilities catalogue, started November 2021, is the practical priority list. It contains only flaws with confirmed exploitation, and it is short enough to act on.
9. The reason patching does not happen is never ignorance. It is fear of breakage, and the answer is automated testing and staged rollout, which is an engineering investment, not a security one.

■ 44.14.5 TECHNICAL – the engineer's version

1. Layers, with a concrete control at each.

Layer	Control	Example
Network	Default-deny egress	Firewall, VPC rules
Host	Secure and measured boot	TPM 2.0, UEFI
Process	Mandatory access control	SELinux, AppArmor
Data	Encryption at rest	LUKS, KMS keys

2. UEFI Secure Boot verifies signatures on the bootloader and kernel. Measured boot extends PCR registers in the TPM with hashes of each stage, enabling remote attestation.
3. TPM 2.0 has been an ISO standard as ISO/IEC 11889 since 2015 and is a Windows 11 requirement. Apple's equivalent is the Secure Enclave.
4. Container hardening: run as a non-root user, drop all Linux capabilities and add back only what is needed, set a read-only root filesystem, apply a seccomp profile restricting system calls, and never mount the Docker socket into a container.
5. Kubernetes specifics: Pod Security Admission at restricted, network policies that default to deny, and short-lived projected service account tokens rather than long-lived secrets.

6. Egress filtering is underrated. Most malicious activity needs to call out. A default-deny outbound policy with a small allowlist blocks a large share of post-compromise activity and makes SSRF far less useful to an attacker.
7. Secrets belong in a manager such as HashiCorp Vault, AWS Secrets Manager or the cloud equivalent, retrieved at run time, with automatic rotation.
8. Commands to observe: `nft list ruleset` or `iptables -L -n` for firewall state, `tpm2_pcrread` for boot measurements, `cryptsetup status` for LUKS, `docker inspect` for container privileges, and `kubectrl auth can-i --list`.

■ 44.14.6 WORDS – remember these

Defence in depth — assume each layer fails — overlapping independent controls so no single failure is fatal.

Segmentation — watertight compartments — dividing networks so compromise does not spread laterally.

Zero trust — being inside proves nothing — architecture where every request is authenticated and authorized on its own merits.

TPM — the small chip that keeps secrets honestly — hardware root of trust storing keys and boot measurements, ISO/IEC 11889.

Measured boot — a signed record of what actually loaded — hashing each boot stage into TPM registers for later attestation.

Egress filtering — controlling what may call out — outbound network policy limiting destinations a host may reach.

Patch window — how long you stay exploitable — time between a fix being published and being installed.

44.15 Detection and response

■ 44.15.1 PLAIN – in simple words

1. You will not prevent everything. So you must be able to notice, and then act, without making it worse.
2. **Logging** is recording what happened. Useful logs answer who, what, when, from where, and did it succeed.
3. A **SIEM** collects logs from everywhere into one searchable place and raises alerts on patterns.
4. **Intrusion detection** watches network traffic or host behaviour for known bad patterns or unusual activity.
5. A **honeypot** is a deliberately attractive fake system. Nobody has a legitimate reason to touch it, so any interaction is a real signal.
6. The response process has six steps, and they are always taught in this order: prepare, identify, contain, eradicate, recover, learn.
7. The first hour matters most, and the most common expensive mistake is rushing to clean up before you understand what happened.

■ 44.15.2 PLAIN – a picture in your head

1. Think of a shop after a break-in.
2. The instinct is to sweep up the glass and reopen. That destroys the evidence and tells you nothing about how they got in.
3. The professional sequence is: stop the bleeding, photograph everything, then clean.
4. Containment is boarding up the broken window. Eradication is changing every lock they might have copied.
5. Recovery is reopening. Learning is the meeting where you find out the alarm had been silenced for three months.

Where this comparison breaks: a burglar leaves. An intruder in a network is often still present, watching your response, including your incident channel. That is why serious responses move to an out-of-band communication channel.

■ 44.15.3 PLAIN – a worked example

1. The first hour of a suspected breach, in order.

0-10 min	Believe it. Start a timestamped written log.
10-20 min	Assemble: security, an owner who can decide, legal.
20-30 min	Contain: isolate hosts from the network, but DO NOT power them off. Memory is evidence.
30-40 min	Preserve: snapshot disks, export logs to a separate account the attacker cannot reach.
40-50 min	Revoke: rotate credentials, tokens, sessions, starting with anything highly privileged.
50-60 min	Scope: what did that identity touch, and when? Decide on notification duties and timers.
2. Two rules inside that hour. Do not power a machine off if memory may hold evidence; isolate the network interface instead.
3. Do not discuss the incident on the systems you suspect are compromised. Move to phones or a separate messaging service.
4. What good logging looks like, as fields rather than prose.

Field	Example
Who	user or service identity
What	action and target object
When	UTC, ISO 8601, synced
From	source address, device

5. Add the outcome, success or failure, and a request identifier that follows the request across services. Without that identifier you cannot reconstruct anything in a distributed system.

■ 44.15.4 PLAIN – what is really happening inside

1. Log what matters and only what matters: authentication successes and failures, privilege changes, access to sensitive data, configuration changes, and outbound connections to new destinations.
2. Never log secrets, passwords, full card numbers, tokens or session identifiers. Logs are widely readable and widely backed up.
3. Ship logs off the machine immediately. A log stored only on the compromised host is a log the attacker can edit.

4. Retention is a real decision. Median attacker dwell time before detection has fallen over the last decade but is still measured in days to weeks, so thirty days of logs is often not enough to answer basic questions.
5. Detection tuning is the hard part. An alert that fires fifty times a day is not an alert; it is background noise that teaches people to ignore it.
6. Honeypots and honeytokens are cheap and high signal. A fake credential placed in a document that nobody should use, wired to an alert, tells you the moment someone is looking.
7. Preparation is the step that actually determines outcome: contact lists, decision authority, pre-approved isolation actions, a written plan, and one rehearsal per year.
8. Rehearse with a tabletop exercise. Read a scenario aloud and make people say what they would do. It reliably exposes the missing phone number, the unclear authority, and the backup nobody has tested restoring.

■ 44.15.5 TECHNICAL – the engineer’s version

1. The six-phase cycle is NIST SP 800-61, currently revision 3, published April 2025, which restructures incident response around the Cybersecurity Framework 2.0 functions. The older SANS six-step naming remains common.
2. Detection technologies: network intrusion detection with Suricata or Zeek, host detection with Wazuh or OSSEC, and endpoint detection and response agents for process-level telemetry.
3. Detection rules are shared as content: Sigma for log rules, YARA for file and memory patterns, and Suricata rules for network traffic.
4. Log pipelines in common use: the Elastic Stack, Splunk, Grafana Loki, or a cloud-native pairing such as CloudWatch with Athena. The important property is a separate trust boundary from the systems being logged.
5. Time synchronization is not optional. Without NTP and a single time zone, usually UTC, correlation across systems is guesswork.
6. Forensic preservation order, from most to least volatile: CPU registers and caches, memory, network state, running processes, disk, then archived logs. Capture in that order.
7. Notification timers start whether or not you are ready. GDPR Article 33 requires notifying a supervisory authority within 72 hours of becoming aware. India’s DPDP Rules 2025 require intimation without delay and a fuller report to the Data Protection Board within 72 hours.
8. Tools to observe and preserve: `journalctl`, `auditd` with `ausearch`, `dmesg`, Velociraptor for remote forensics, and `dd` or cloud snapshot APIs for disk images. Record hashes of every image you take.

■ 44.15.6 WORDS – remember these

SIEM — one searchable place for all logs — security information and event management with correlation and alerting.

IDS — a watcher for known bad patterns — intrusion detection system, network or host based.

Honeypot — bait with no legitimate use — deliberately exposed decoy whose every interaction is a signal.

Dwell time — how long they were inside unnoticed — interval between initial compromise and detection.

Containment — stopping the spread without destroying evidence — isolating affected systems while preserving volatile state.

Eradication — removing their access completely — eliminating footholds, credentials and persistence mechanisms.

Tabletop exercise — a rehearsal with no machines — discussion-based drill testing decisions and contacts.

44.16 Privacy and the law, briefly

■ 44.16.1 PLAIN – in simple words

1. Security is about stopping people who should not have access. Privacy is about limiting what you collect and what you do with it, even when you have every right to hold it.
2. You can be perfectly secure and badly privacy-invading at the same time.
3. **Personal data** is anything relating to an identified or identifiable person. That includes identifiers such as device numbers and addresses, not only names.
4. The core ideas of modern data protection law are short.
5. Have a lawful reason to process. Tell people plainly. Collect only what you need. Keep it only as long as you need it. Keep it secure. Let people see, correct and delete their data.
6. **Data minimization** is the one engineers control directly. Every field you do not collect is a field that cannot leak.
7. If a breach exposes personal data, you usually have a legal duty to tell a regulator, and often the affected people, on a clock measured in hours.

■ 44.16.2 PLAIN – a picture in your head

1. Security is the lock on the filing cabinet.
2. Privacy is the decision about which forms go into the cabinet in the first place, who may open it and why, and when the papers are shredded.
3. A shop with an excellent safe that photographs every customer and keeps the images forever is secure and privacy-invading.
4. A shop that photographs nobody has nothing to lose in a burglary.

Where this comparison breaks: paper is finite and visible. Digital data is copied silently, spreads into backups, analytics systems and third-party tools, and the copy in the log aggregator is usually the one nobody deleted.

■ 44.16.3 PLAIN – a worked example

1. Two regimes a working engineer meets most often.

Point	GDPR	India DPDP
In force	25 May 2018	Act 2023, Rules 2025
Breach to regulator	72 hours	72 hours
Top penalty	20M EUR or 4%	Rs 250 crore
Regulator	National authorities	Data Protection Board

2. GDPR fines have two tiers: up to 10 million euro or 2 percent of worldwide annual turnover for most obligations, and up to 20 million euro or 4 percent for the most serious. The higher of the fixed amount and the percentage applies.

3. India's Digital Personal Data Protection Act was passed in August 2023. The Digital Personal Data Protection Rules were notified on 14 November 2025, with the substantive obligations, including breach reporting and security safeguards, phased in over about eighteen months from that date.
4. Under the DPDP Act, failure to take reasonable security safeguards carries a penalty of up to 250 crore rupees, and failure to notify a breach up to 200 crore rupees.
5. Under the DPDP Rules, affected individuals must be informed without delay, and a fuller report reaches the Data Protection Board within 72 hours.

■ 44.16.4 PLAIN - what is really happening inside

1. The engineering consequences of these laws are concrete, not abstract.
2. You need a data inventory: what personal data exists, where it lives, who can reach it, and how long it stays. Most organizations cannot answer this.
3. You need deletion that actually works, including in backups, search indexes, analytics and every third-party processor.
4. You need to answer access requests, which means being able to find one person's data across systems.
5. You need to record consent where consent is your lawful basis, including when it was given, for what, and how it can be withdrawn.
6. Pseudonymization replaces identifiers with tokens but remains personal data, because it can be reversed with the mapping table.
7. Anonymization means the person can no longer be identified by anyone, and it is much harder than it looks. Combining a few "anonymous" fields such as postcode, birth date and sex re-identifies most individuals.
8. The cheapest compliance strategy is the same as the cheapest security strategy: do not collect it, and if you must, delete it on a schedule that runs automatically.

■ 44.16.5 TECHNICAL - the engineer's version

1. GDPR is Regulation (EU) 2016/679, applicable from 25 May 2018. Article 5 sets the principles, Article 6 lawful bases, Articles 12 to 22 data subject rights, Article 25 data protection by design and by default, Article 32 security of processing, Articles 33 and 34 breach notification.
2. Article 33 requires notification to the supervisory authority within 72 hours of becoming aware, unless the breach is unlikely to result in risk. Article 34 requires telling the individuals when the risk is high.
3. The largest single GDPR fine to date is 1.2 billion euro against Meta in May 2023, concerning transfers of European user data to the United States.
4. Other regimes worth knowing by name: the California Consumer Privacy Act as amended by CPRA, Brazil's LGPD, and sector rules such as HIPAA for United States health data and PCI DSS, currently version 4.0.1, for card data.
5. PCI DSS is a contractual standard, not a law, but it is enforced through card scheme agreements and behaves like one in practice.
6. Technical measures named in practice: encryption at rest and in transit, pseudonymization, access logging, retention policies enforced by automation, and documented data processing agreements with every processor.
7. Privacy and security overlap at Article 32 but are not the same duty. A system can satisfy every security control and still be unlawful because it collects data it has no basis to collect.

■ 44.16.6 WORDS – remember these

Personal data — anything about an identifiable person — includes indirect identifiers such as device and network addresses.

Data minimization — do not collect what you do not need — principle limiting collection to what is adequate and relevant.

Breach notification — the legal clock after an incident — duty to inform regulators and individuals, typically within 72 hours.

Pseudonymization — identifiers swapped for tokens — reversible with a separately held mapping, still personal data.

Anonymization — genuinely no longer a person — irreversible, and much harder than it looks in practice.

Lawful basis — your legal reason to process — one of the grounds required before processing personal data.

Data protection by design — build it in, not on — the requirement to embed protection into system design by default.

44.17 Security for the individual developer

■ 44.17.1 PLAIN – in simple words

1. Most of what protects you is a short list of habits, not expensive tools.
2. Keep your machine current. Turn on automatic updates for the operating system and the browser.
3. Encrypt the disk. FileVault, BitLocker or LUKS, with a real passphrase.
4. Use a password manager, and put a hardware key or passkey on the accounts that matter most: email first, then code hosting, then cloud.
5. Email first, because email resets everything else.
6. Never commit a secret. Not a token, not a password, not a private key, not a connection string.
7. Assume that anything committed once is compromised forever, even after you delete it, because history is public and cloned within minutes.
8. Keep your dependencies current and know what they are.
9. Read code with security in mind, which is a specific skill, not general attentiveness.
10. And if you find a flaw in somebody else's software, report it responsibly rather than publishing it or exploiting it.

■ 44.17.2 PLAIN – a picture in your head

1. Think of a workshop. Locked door, tools accounted for, dangerous materials in a cabinet rather than on the bench.
2. Committing a secret is leaving the spare key taped to the outside of the door, then painting over it and believing it is gone.
3. Automated scanners crawl public repositories continuously. Published measurements repeatedly show credentials being found and used within minutes of being pushed.
4. So the moment of exposure is not when someone reads your code. It is when the commit lands.

Where this comparison breaks: a taped key can be removed. A committed secret is in every clone, every fork, every cache, and every scraper's archive. The only real remedy is revocation and rotation, not deletion.

■ 44.17.3 PLAIN – a worked example

1. Practical checklist, grouped, in the order worth doing it.

Area	Action
Machine	Auto-update, disk encryption
Accounts	Manager, passkey on email
Secrets	Manager, never in git
Dependencies	Lock files, audit in CI
Code	Review checklist, static scan

2. Secrets handling in practice. Keep values in environment variables loaded from a file that is listed in `.gitignore`, or in a real secret manager.
3. Commit a `.env.example` with the names and no values, so the next person knows what is needed.
4. Install a pre-commit hook that blocks obvious secret patterns, and enable push protection on the hosting side as a second net.
5. If a secret is exposed, the order is fixed and not negotiable.
 1. Revoke the credential immediately.
 2. Issue a replacement and deploy it.
 3. Check logs for use of the old one.
 4. Only then clean the history, if worth it.
6. Rewriting history is the last step and the least important. The credential is already public; rotating it is what actually ends the exposure.

■ 44.17.4 PLAIN – what is really happening inside

1. Reading code for security means asking a different set of questions from normal review.
2. Where does untrusted input enter, and where does it end up? Trace it from the boundary to every interpreter it touches.
3. Is authorization checked on every path to this object, including the new endpoint added last week?
4. Are errors handled on every branch, and does the failure path fail closed rather than open?
5. Is anything constructed by string concatenation that should be parameterized: SQL, shell commands, HTML, file paths, URLs?
6. Is any cryptography home-made, and is any random value coming from a non-cryptographic generator?
7. Do log statements or error messages contain secrets or personal data?
8. Two kinds of tool help, and they see different things.
9. **Static analysis** reads code without running it. It finds patterns and scales to every file, at the cost of false positives.
10. **Dynamic analysis** runs the system and observes it. It finds real behaviour including configuration problems, but only on the paths it exercises.
11. **Fuzzing** is dynamic analysis with automatically generated inputs, and it is the most effective way to find memory safety and parsing bugs.
12. None of these replaces a person who understands what the application is supposed to protect.

■ 44.17.5 TECHNICAL – the engineer's version

1. Secret scanning: `gitleaks`, `trufflehog`, GitHub secret scanning with push protection, and `detect-secrets` as a pre-commit hook.

2. Static analysis: `semgrep` with community rule packs, `CodeQL`, `bandit` for Python, `gosec` for Go, `cargo clippy` for Rust lints, `brakeman` for Rails.
3. Dependency scanning: `npm audit`, `pip-audit`, `cargo audit`, `osv-scanner`, `Dependabot` or `Renovate` for automated update pull requests.
4. Dynamic analysis: `OWASP ZAP` for web scanning, `nuclei` for templated checks, `trivy` for container and infrastructure-as-code scanning.
5. Fuzzing: `libFuzzer` and `AFL++` for native code, `go-fuzz` and Go's built-in fuzzing since Go 1.18, `Atheris` for Python, and `OSS-Fuzz` for open-source projects that qualify.
6. Coordinated disclosure, the correct process when you find a flaw in software you do not own.
7. Look for a `security.txt` file at `/.well-known/security.txt`, standardized as RFC 9116 in April 2022, or a `SECURITY.md` in the repository, or a bug bounty programme.
8. Report privately with enough detail to reproduce. Propose a disclosure timeline. Ninety days is the widely used norm, following Google Project Zero's policy since 2014, with a shorter clock if active exploitation is observed.
9. Do not test against systems you do not own or have written authorization for. In many jurisdictions, including under India's Information Technology Act 2000 and the United States Computer Fraud and Abuse Act, unauthorized access is a criminal offence regardless of intent.
10. Request a CVE identifier through the vendor if they are a CVE Numbering Authority, or through MITRE otherwise. Coordinate publication with the fix.
11. If the vendor does not respond, escalate to a national CERT, such as CERT-In in India or CISA in the United States, rather than publishing.
12. Keep a written record of every contact attempt with dates. If disclosure ever becomes contested, that record is what protects you.

■ 44.17.6 WORDS - remember these

Secret scanning — automated hunting for committed credentials — pre-commit and server-side detection of key patterns.

Rotation — replacing a credential — issuing a new secret and invalidating the old one, the only real remedy for exposure.

Static analysis — reading code without running it — pattern and dataflow analysis over source, SAST.

Dynamic analysis — testing the running system — behavioural testing against a deployed instance, DAST.

Fuzzing — throwing generated inputs at it — automated input generation to find crashes and parsing flaws.

Coordinated disclosure — report privately, publish together — agreed process between finder and vendor with a timeline.

`security.txt` — where to send a vulnerability report — RFC 9116 file at a well-known path listing security contacts.

44.18 Famous breaches and what each teaches

■ 44.18.1 PLAIN - in simple words

1. Six incidents, chosen because they are well documented and because each fails in a different way.
2. Read them as a list of things that were possible, not as a list of villains.
3. In every one of them, the technical flaw was known and fixable beforehand.

■ 44.18.2 PLAIN – a picture in your head

1. Think of aviation accident reports. Nobody reads them for blame.
2. They are read because one crew's bad day becomes every other crew's checklist item.
3. Security has the same discipline available and uses it far less often.

Where this comparison breaks: aviation has a legal duty to publish investigations. Most breach reports are written by lawyers, and the technically interesting details are usually the ones removed.

■ 44.18.3 PLAIN – a worked example

1. The six, with root cause and lesson.

Incident	Year	Root cause
Target	2013	Supplier access, flat network
Equifax	2017	Unpatched Apache Struts
WannaCry	2017	Unpatched SMB, no segmentation
SolarWinds	2020	Compromised build pipeline
Colonial Pipeline	2021	VPN password, no MFA
Change Healthcare	2024	Citrix portal without MFA

2. Target, December 2013: credentials belonging to a refrigeration contractor reached the payment network because the network was flat. About 40 million card records and 70 million customer records were exposed.
3. Equifax, 2017: a fix for Apache Struts flaw CVE-2017-5638 was available on 7 March 2017. Intrusion began around 13 May and was discovered on 29 July. About 147 million people were affected; the settlement was up to 700 million dollars.
4. WannaCry, 12 May 2017: ransomware using the EternalBlue SMB exploit, for which Microsoft had shipped a patch on 14 March 2017. It reached about 200,000 machines in 150 countries and disrupted parts of the NHS.
5. SolarWinds, disclosed December 2020: the build system was subverted, so the malicious code carried a valid vendor signature. Roughly 18,000 customers installed the trojanized update.
6. Colonial Pipeline, 7 May 2021: a legacy VPN account with a leaked password and no multi-factor authentication. Fuel supply to the United States east coast was disrupted and a 4.4 million dollar ransom was paid.
7. Change Healthcare, February 2024: a Citrix remote access portal without multi-factor authentication. It became the largest health data breach ever reported in the United States, at 192.7 million individuals, with a reported 22 million dollar ransom payment.

■ 44.18.4 PLAIN – what is really happening inside

1. The lessons, one line each, in the same order.
2. Target: a supplier's access is your access. Segment the network so a contractor's account cannot see a payment terminal.
3. Equifax: the gap between a patch existing and a patch being installed is the whole vulnerability. Measure that gap in days.
4. WannaCry: a patch you did not deploy is a patch you did not have. And a flat internal network turns one infection into all of them.
5. SolarWinds: a valid signature proves who built it, not that the build was honest. Protect the build system as tightly as production.

6. Colonial Pipeline: every remote access path needs multi-factor authentication, including the old one nobody uses.
7. Change Healthcare: the same lesson again, three years later, at a much larger scale. This is the single most repeated root cause in the record.
8. The pattern across all six is dull and consistent. Known flaw, known fix, missing basic control, insufficient segmentation.
9. Not one of them required a novel attack technique.

■ 44.18.5 TECHNICAL – the engineer’s version

1. Detection times are the other lesson. Equifax ran for roughly 76 days before discovery. The Codecov modification ran for 60 days. The SolarWinds implant ran for months before Mandiant identified it in December 2020.
2. Notable data-handling failures worth naming separately, because they are about storage rather than intrusion.
3. RockYou, December 2009: 32 million passwords stored in plain text. The resulting list is still the standard wordlist in password auditing tools.
4. LinkedIn, June 2012: 6.5 million password hashes leaked, SHA-1 with no salt. The full set later proved to be about 117 million accounts.
5. Adobe, October 2013: about 153 million records, passwords encrypted with 3DES in ECB mode rather than hashed, with plaintext password hints alongside. Identical passwords produced identical ciphertext.
6. Yahoo, breaches in 2013 and 2014 disclosed in 2016 and 2017: 3 billion accounts, with MD5 hashing on a portion of them.
7. Each of those is a direct illustration of section 44.3. Plain text, fast unsalted hashes, and encryption used where hashing was required.
8. For breach data, the checkable public sources are the Have I Been Pwned service, the annual Verizon Data Breach Investigations Report published each spring since 2008, and the United States Department of Health and Human Services breach portal for health-sector figures.

■ 44.18.6 WORDS – remember these

Root cause — the thing that actually let it happen — the earliest correctable failure, not the final symptom.

Dwell time — how long they stayed unnoticed — interval from compromise to detection, often weeks.

Lateral movement — spreading sideways once inside — using one foothold to reach further systems.

Flat network — everything can reach everything — an unsegmented network where one compromise exposes all.

Patch gap — the window between fix and install — the period during which a known, fixed flaw remains exploitable.

Third-party risk — their weakness becomes yours — exposure arising from suppliers, contractors and integrations.

44.98 Common wrong ideas

1. Wrong: we are too small to be a target. Right: most attacks are automated and indiscriminate. Scanners find every internet-facing service within hours, and ransomware crews prefer small organizations because they pay faster and have weaker controls.

2. Wrong: the padlock in the browser means the site is safe. Right: it means the connection is encrypted to whoever holds a valid certificate for that name. A phishing site can obtain one for free in minutes. It says nothing about who runs the site or what they do with your data.
3. Wrong: antivirus protects me. Right: signature-based scanning misses new, repacked and fileless malware, and cannot judge whether a legitimate administrative tool is being misused. It is one useful layer among several, not a boundary.
4. Wrong: open source is less secure because anyone can read the code. Right: visibility helps defenders more than attackers, since attackers can analyze closed binaries too. The real open-source risk is different: unpaid, unsupported maintainers of critical components, which is exactly what the xz-utils backdoor exploited in 2024.
5. Wrong: a strong password is enough. Right: length and randomness do not help if the password is phished, reused on a breached site, or stolen by malware. Phishing-resistant multi-factor authentication is the control that changes the outcome.
6. Wrong: encryption solves security. Right: encryption protects confidentiality in transit and at rest. It does nothing about broken access control, injection, social engineering, or an attacker using your own application while it is unlocked and running.
7. Wrong: we passed a penetration test, so we are secure. Right: a test is a sample of one moment by one team within a fixed scope. It shows what was found, not what exists.
8. Wrong: internal systems do not need the same protection. Right: that assumption is exactly what turned the Target and WannaCry incidents into catastrophes. Assume the attacker is already inside and design accordingly.
9. Wrong: hashing a password with SHA-256 is fine because SHA-256 is secure. Right: SHA-256 is secure as a hash and terrible as a password function precisely because it is fast. Use Argon2id, scrypt or bcrypt.
10. Wrong: we have backups, so ransomware cannot hurt us. Right: modern ransomware finds and deletes backups first and steals the data before encrypting, so publication remains a threat. Backups must be tested, versioned and beyond the reach of production credentials.

44.99 Chapter summary in 20 lines

1. Security is five promises about data: confidentiality, integrity, availability, authenticity and non-repudiation, always in trade-off.
2. It is a property of a whole system in a context, never a product you buy.
3. Threat modelling is three questions: what am I protecting, from whom, and what happens if I fail. Rank the answers by likelihood times impact.
4. A hash is a one-way fingerprint. MD5 fell to collisions in August 2004 and SHA-1 to the SHattered attack on 23 February 2017. Use SHA-2, SHA-3 or BLAKE3, and never a bare hash for passwords.
5. Store passwords with a deliberately slow, memory-hard function: Argon2id at 19 MiB with two passes, scrypt, or bcrypt at cost 10 or above, always with a unique per-user salt and ideally a separately stored pepper.
6. NIST SP 800-63B revision 4, published 31 July 2025, forbids composition rules and routine expiry. Length beats complexity, and a manager beats both.
7. Symmetric encryption must be authenticated. Use AES-GCM or ChaCha20-Poly1305, never ECB, and never reuse a nonce with the same key.
8. Asymmetric cryptography solves key agreement and gives signatures, which prove origin and integrity in a way encryption alone never does. Every real system is hybrid: asymmetric once, symmetric thereafter.
9. NIST approved ML-KEM, ML-DSA and SLH-DSA as FIPS 203, 204 and 205 on 13 August 2024,

and selected HQC as a fifth algorithm on 11 March 2025. Harvest-now-decrypt-later makes key exchange migration urgent today.

10. Authentication factors are not equal. SMS is weak, app codes are relayable, and only FIDO2 and passkeys resist phishing by binding to the origin.
11. Authorization is a separate question from authentication, checked on every request, at the data layer, with least privilege and deny by default.
12. Broken Access Control has been number one in the OWASP Top 10 in both the 2021 and the current 2025 edition, which was presented on 6 November 2025.
13. Injection is fixed by construction with parameterized queries and argument arrays, never by filtering. Cross-site scripting is fixed by contextual output encoding plus a strict Content Security Policy.
14. Memory safety bugs have been roughly seventy percent of the vulnerabilities at major vendors. Canaries, non-executable memory, ASLR and control flow integrity raise cost; memory-safe languages remove the class.
15. Network defence rests on TLS against interception, DNSSEC and encrypted DNS for name lookups, and rate limiting, filtering, anycast and scrubbing against floods that now exceed 30 terabits per second.
16. Most intrusions begin with a person or a stolen credential, not an exploit. Business email compromise alone cost 3.04 billion dollars in 2025.
17. Your dependencies are your attack surface. Use lock files, pin by digest, generate SBOMs, and require signed provenance. Remember that xz-utils in March 2024 was caught by one engineer noticing half a second of CPU time.
18. Defend in depth: segment, patch fast, encrypt disks, sandbox, and give everything the least privilege it can function with.
19. Prepare for the incident you will have. Log the right fields off-host, rehearse once a year, and in the first hour contain without destroying evidence, preserve, revoke and scope before you clean.
20. Every famous breach in section 44.18 had a known flaw and an available fix. The advantage is almost never in exotic defence; it is in doing the dull things quickly and everywhere.



PART H

Games and Machine Intelligence

How a game runs sixty times a second, and how a language model is built, judged and used.

Games — How They Are Made and How They Run

45.0 What this chapter gives you

1. You will be able to say exactly what makes a game different from every other program on your machine, in one sentence, and defend it.
2. You will be able to write a correct game loop with a fixed physics step, an accumulator and interpolation, and explain every line of it.
3. You will be able to do frame budget arithmetic in your head, and explain why “average frames per second” hides the thing that actually annoys you.
4. You will be able to describe how a real game project is staffed and funded, from a one-person hobby project to a three-hundred-person AAA title, with real money figures.
5. You will be able to compare Unity, Unreal Engine and Godot honestly on cost, language and licence, including the 2023 Unity fee episode.
6. You will be able to explain what an asset pipeline does, why a build takes hours, and what “cooking” content means.
7. You will be able to work through a collision resolution by hand and check that momentum is conserved.
8. You will be able to run the A-star pathfinding algorithm on a grid by hand, and explain why game AI is designed to be beaten.
9. You will be able to explain client-side prediction, server reconciliation, entity interpolation and lag compensation, and say precisely why you get shot after you reached cover.
10. You will be able to trace what happens between the double-click and the first frame of gameplay, in twenty-five steps with real timings.

45.1 What a game actually is as a program

■ 45.1.1 PLAIN – in simple words

1. Most programs are lazy. They sit still and wait for you to do something.
2. A text editor draws your document once, then does nothing until you press a key. If you walk away for an hour, it does nothing for an hour.
3. A game is the opposite. A game never sits still.
4. A game runs the same short list of jobs over and over, very fast, forever, whether or not you touched anything.
5. That list is: read the controls, work out what the world looks like now, draw it, show it. Then do it again.

6. One trip round that list is called a **frame**.
7. At 60 frames a second, the game does this 60 times every second. That is once every 16.67 thousandths of a second.
8. It keeps doing it even when you stand perfectly still in an empty room.
9. Why? Because the world keeps moving. Water flows, clouds drift, an enemy walks somewhere far away, a timer counts down.
10. The game cannot know in advance whether anything changed. Working that out would cost more than just redoing the work.
11. So it redoes all of it, every single time, on a strict clock.
12. That one decision, “redo everything on a clock”, explains almost every other odd thing about how games are built.

■ 45.1.2 PLAIN – a picture in your head

1. Think of two people who work in the same building.
2. The first is a receptionist. She sits at a desk and does nothing until someone walks in. Then she deals with them and goes back to waiting.
3. Her day is quiet. Her energy use is low. Nothing happens between visitors.
4. The second is a film projectionist in an old cinema.
5. He cannot wait for anything. Twenty-four times a second a new picture must be in front of the lamp, whether the film is exciting or boring.
6. If he is late by even a fraction of a second, the audience sees a flicker and the illusion of motion breaks.
7. The receptionist is an ordinary application. The projectionist is a game.
8. Now notice what this does to the projectionist’s job. He cannot go and make tea. He cannot take a long phone call. He has a deadline every 41 thousandths of a second, all day.
9. Everything about how he arranges his booth is decided by that deadline.

Where this comparison breaks: the projectionist is only showing pictures that already exist on the film. A game invents each picture from scratch, having first worked out what the world looks like at that instant. It is closer to a projectionist who must also draw each frame in the time between showing them. Also, real film runs at a fixed 24 frames per second and cannot be late. A game can be late, and when it is, you see it as a stutter rather than a flicker.

■ 45.1.3 PLAIN – a worked example

1. Take one hour of a person using a word processor, and one hour of the same person playing a game.
2. In the word processor, the screen is redrawn when something changes: a letter typed, a scroll, a menu opened.
3. A fast typist hits about 400 keys a minute, so about 24,000 keys an hour.
4. Even if each key caused one redraw, that is 24,000 redraws in an hour.
5. In the game at 60 frames a second: $60 \times 60 \times 60 = 216,000$ frames an hour.
6. That is nine times more drawing, and the drawing itself is far heavier: millions of triangles instead of a page of text.
7. Now watch what the machine does.

Program state	Word processor	Game at 60 fps
Screen updates per hour	~24,000	216,000

Program state	Word processor	Game at 60 fps
CPU while idle	near 0 percent	one core busy
GPU while idle	near 0 watts	150 to 450 watts
Deadline per update	none	16.67 ms

8. The word processor has no deadline at all. If a redraw takes 200 milliseconds, nobody dies.
9. The game has a hard deadline, 60 times a second, and missing it is visible.
10. This is why your laptop fans spin up for a game and stay silent for a document, even though both are “just a program”.

■ 45.1.4 PLAIN – what is really happening inside

1. Both kinds of program start the same way. The operating system loads them and calls a starting function.
2. The lazy program then enters a **message loop**. It asks the operating system: “is there anything for me?” and, if not, it goes to sleep.
3. Going to sleep is the key. The program is removed from the list of things that need the processor. It uses no time at all.
4. The operating system wakes it only when a real event arrives: a key, a mouse click, a window resize, a timer.
5. When it wakes, it handles the one event, redraws only the part of the window that changed, and goes back to sleep.
6. A game also has a message loop, because the operating system still sends it events. But it never sleeps in it.
7. Instead it asks “is there anything for me?”, takes whatever is there without waiting, and immediately moves on to its own work.
8. Its own work is: read the current state of every input device, advance the simulated world by a small slice of time, and build a new picture.
9. Then it asks the graphics system to show that picture, and starts again.
10. The only place it ever waits is at the very end, when it hands the finished picture to the display and waits for the screen’s next refresh.
11. That single wait is what keeps a game at exactly 60 or 120 frames a second instead of running as fast as the hardware allows.

■ 45.1.5 TECHNICAL – the engineer’s version

1. An event-driven application on Windows blocks in GetMessage, which does not return until a message is in the thread’s queue. The thread is in a wait state and consumes no scheduler quantum.
2. It redraws in response to WM_PAINT, and the system tracks an **update region** so only invalidated rectangles are repainted. On macOS the equivalent is setNeedsDisplay and drawRect; on X11 it is Expose.
3. A real-time application instead uses PeekMessage with PM_REMOVE, which returns immediately whether or not a message is waiting, then falls through to its own frame code. This is the standard game shape and has been since the Win32 era.
4. The frame ends with a present: IDXGISwapChain::Present in Direct3D, vkQueuePresentKHR in Vulkan, [CAMetalDrawable present] in Metal, SDL_GL_SwapWindow or eglSwapBuffers in the portable stacks.

5. The swap chain holds two or three back buffers. With a sync interval of 1, Present blocks until the display's vertical blanking interval, which is what pins the frame rate to the refresh rate.
6. With variable refresh rate displays (VESA Adaptive-Sync, marketed as AMD FreeSync from 2015 and NVIDIA G-Sync from 2013) the panel waits for the game rather than the other way round, within a supported range, typically 48 to 144 Hz or wider.
7. The consequences of the loop shape are measurable. A game pins at least one CPU core near 100 percent and keeps the GPU in its highest power state. A discrete GPU that idles at 10 to 25 W will draw 150 to 450 W under load.
8. The other big consequence is memory behaviour. An event-driven application can allocate freely, because a 5 ms garbage collection pause is invisible. A game cannot: a 5 ms pause inside a 16.67 ms budget is 30 percent of the frame, and a 30 ms pause is a visible hitch.
9. This is why game code pre-allocates pools, avoids per-frame heap allocation, and why C# game code is written to keep the generation 0 heap quiet rather than in idiomatic allocating style.

Property	Event-driven app	Real-time game
Blocking call	GetMessage	none, PeekMessage
Redraw trigger	invalidated region	every frame
Frame deadline	none	16.67 ms at 60 Hz
Allocation policy	free	pooled, per-frame zero

10. Tools that show the difference: Windows Task Manager and powermetrics on macOS for power draw, perf top on Linux for the busy loop, PresentMon on Windows for per-frame present timings, and RenderDoc for a captured frame.

■ 45.1.6 WORDS – remember these

1. Frame — one complete picture — one iteration of the game loop producing one presented back buffer.
2. Game loop — the list of jobs done over and over — the input, update, render, present cycle running at the target frame rate.
3. Event-driven — waits until poked — blocks on a message queue and repaints only invalidated regions.
4. Present — showing the finished picture — handing a back buffer to the display engine, optionally synchronized to vertical blank.
5. Swap chain — the queue of pictures waiting to be shown — a set of back buffers rotated between the renderer and the display controller.
6. Vertical blank — the pause between screen refreshes — the interval in which a buffer swap can occur without tearing.

45.2 A history of video games as technology

■ 45.2.1 PLAIN – in simple words

1. Video games did not start as a business. They started as people playing with expensive laboratory equipment.
2. In 1958 a physicist wired an analogue computer to a small round screen so that visitors to his laboratory's open day would have something to do.
3. In 1962 students at a university did something similar on a room-sized computer, and the program spread to every machine of that type.

4. The first machines you could buy for a home, in 1972, had no computer inside at all. They were built from simple switching parts wired together.
5. Then chips got cheap. Arcade cabinets became small computers with one game burned into them.
6. Then home consoles became small computers with a slot for a cartridge.
7. Then home computers became fast enough to draw a three-dimensional world, and then extra chips were added just for drawing.
8. Then the internet arrived, and games stopped being finished things and started being services that keep changing.
9. Then phones arrived, and more people played games than ever before, mostly for free, paying in small amounts inside the game.
10. Every one of those steps was a technical step first, and a business step second.

■ 45.2.2 PLAIN – a picture in your head

1. Think about how photography grew up.
2. First there were one-off chemical experiments by scientists, making a single image with enormous effort.
3. Then there were heavy plate cameras owned by professionals with studios.
4. Then a company put film in a small box and said “you press the button, we do the rest”, and ordinary people took pictures.
5. Then colour arrived, then instant prints, then video, then digital, then a camera in every pocket connected to the whole world.
6. Games followed the same shape, about eighty years later and much faster.
7. The laboratory oscilloscope is the chemical experiment. The arcade cabinet is the professional studio. The home console is the box with film in it. The phone is the camera in every pocket.

Where this comparison breaks: photography captures something that already exists. A game must create everything, so each step forward needed not just better hardware but also new mathematics and new ways of organizing a team. And unlike photography, games have kept the old forms alive: people still make and play games that would run on 1985 hardware.

■ 45.2.3 PLAIN – a worked example

1. Compare one specific thing across the decades: how many moving pictures per second the machine had to produce, and what it had to remember.

Machine and year	Working memory	Screen
Odyssey, 1972	none	3 white dots
Atari 2600, 1977	128 bytes	160x192, 4 colours
NES, 1983	2 KB	256x240, 25 colours
PlayStation, 1994	2 MB + 1 MB video	320x240 to 640x480
PlayStation 5, 2020	16 GB shared	3840x2160

2. Read the memory column again. 128 bytes on the Atari 2600 is not a typing error. That is 128 individual bytes for all game state.
3. The whole screen would not fit in that memory, so the machine did not have a screen buffer at all.
4. Instead the processor had to change the graphics chip’s registers while the television’s electron beam was sweeping across the screen, line by line.
5. Programmers called this **racing the beam**. You had about 76 processor cycles per scan line to set up the next line.

6. From 128 bytes to 16 gigabytes is a factor of 134 million in about 43 years.

■ 45.2.4 PLAIN – what is really happening inside

1. Each era solved one specific bottleneck, and the games of that era look the way they do because of that one bottleneck.
2. The 1972 machines had no processor, so the game was fixed in wiring. You could not have a game with rules that changed.
3. The 1977 to 1983 machines had a processor but almost no memory, so the world had to be tiny and mostly regenerated each frame.
4. The 1983 to 1990 machines added dedicated graphics hardware that could move small pictures around cheaply, so games became about sprites and scrolling.
5. The early 1990s machines were fast enough to compute perspective in software, so first-person games appeared, with clever mathematics used to avoid work rather than to do it.
6. From 1996 a separate chip did the triangle filling, so the limit moved from “how many pixels can we fill” to “how many triangles can we send”.
7. From about 2001 those chips became programmable, so the limit moved again to “what can we compute per pixel”.
8. From about 2013 the limit for many games became not the hardware but the cost of making enough content to fill it.
9. That last shift is the reason a modern game costs two hundred million dollars and a 1985 game cost a few hundred thousand.

■ 45.2.5 TECHNICAL – the engineer’s version

Year	Milestone	Technical advance
1958	Tennis for Two	Real-time analogue graphics
1962	Spacewar, PDP-1	Stored-program interactive
1971	Computer Space	First coin-operated cabinet
1972	Magnavox Odyssey	Home console, discrete logic
1972	Pong arcade	TTL logic, no CPU, no code
1977	Atari VCS / 2600	Cartridge ROM, 6507 CPU
1978	Space Invaders	Microprocessor arcade game
1983	Market crash	Quality control matters
1983	Famicom in Japan	Sprite and scroll hardware
1985	NES in North America	Lockout chip, licensing model
1988	Mega Drive	16-bit 68000, bigger sprites
1990	Super Famicom	Mode 7 affine transforms
1992	Wolfenstein 3D	Ray-cast pseudo-3D at speed
1993	Doom	BSP trees, modding, LAN play
1994	PlayStation	Hardware triangle rasterizer
1996	3dfx Voodoo Graphics	Consumer 3D acceleration
1999	Dreamcast online	Built-in modem, online play
2002	Xbox Live	Unified console online service
2003	Steam	Digital distribution, patching
2008	Apple App Store	Mobile mass market
2017	Fortnite Battle Royale	Live service at scale
2020	PS5 and Xbox Series X	NVMe streaming, RT hardware
2024	PS5 Pro	Machine-learning upscaling
2025	Nintendo Switch 2	DLSS on a handheld

1. Tennis for Two was shown on 18 October 1958 at Brookhaven National Laboratory in Upton, New York, built by William Higinbotham with technician Robert V. Dvorak. It ran on a Donner Model 30 analogue computer with a five-inch oscilloscope as the display. Each player had an aluminium box with a knob to set the angle and a button to hit the ball.
2. Spacewar was written in 1962 by Steve Russell and others at MIT on a DEC PDP-1 with 9 kilowords of core memory and a Type 30 point-plotting display. It was copied to essentially every PDP-1 installation, which makes it the first game with a distribution story.
3. The Magnavox Odyssey, designed by Ralph Baer at Sanders Associates from the 1967 “Brown Box” prototype, shipped in September 1972. It had no processor, no memory and no software: transistor and diode logic produced three white spots, and coloured plastic overlays taped to the television supplied the backgrounds.
4. Pong, designed by Allan Alcorn at Atari and installed in November 1972, was also built from discrete transistor-transistor logic. There is no program to disassemble because there is no program.
5. The Atari Video Computer System, later the 2600, launched 11 September 1977 with a MOS 6507 at about 1.19 MHz, 128 bytes of RAM, and the Television Interface Adaptor. It had no frame buffer, so the CPU had to modify the TIA registers within each of the 262 scan lines of an NTSC field.
6. The 1983 crash is well documented in numbers: United States home video game revenue fell from roughly 3.2 billion dollars in 1983 to roughly 100 million dollars in 1985, a drop of about 97 percent. Contributing factors were oversupply, poor quality third-party titles, and cheap home computers. Atari buried roughly 728,000 cartridges in a New Mexico landfill in September 1983, confirmed by an excavation in 2014.
7. Nintendo’s answer was a hardware lock. The Famicom launched 15 July 1983 in Japan; the re-designed Nintendo Entertainment System launched in New York on 18 October 1985. Its 10NES system used a Checking Integrated Circuit in the console and an identical one in the cartridge, sharing a 4 MHz clock. The console’s chip sends a challenge and the cartridge’s chip must return a matching pseudo-random stream. On mismatch the console pulls the CPU and picture processing unit reset lines low with a 1 Hz square wave, producing the flashing screen. Different part numbers enforced regions: 3193 and 6113 for NTSC, 3197 for PAL-A, 3195 for PAL-B, 3196 for Asia.
8. The honest version: the lockout chip was not security in the modern sense. It was a business control. It let Nintendo decide who published on its platform, cap the number of titles per publisher, and manufacture all cartridges itself. Every console since has had some version of that control, and it is the reason the certification process in section 45.14 exists.
9. Doom shipped on 10 December 1993. Its renderer precomputed a binary space partitioning tree at level build time so that walls could be drawn in correct depth order without a depth buffer. Walls were drawn as vertical texture columns; floors and ceilings as horizontal spans called visplanes, with a hard limit of 128 of them. All arithmetic was 16.16 fixed point. It could not do rooms above rooms, sloped floors, or a proper look up and down. It also shipped as shareware, kept its data in an editable WAD file, and supported four-player networked deathmatch over IPX, which created the modding and competitive scenes at the same time.
10. 3dfx Voodoo Graphics arrived in late 1996 as an add-in card that did only 3D and passed 2D through from the existing card. GLQuake in January 1997 was the demonstration that made the category mainstream.
11. The PlayStation launched 3 December 1994 in Japan and 9 September 1995 in North America. Its geometry engine had no floating-point unit and its rasterizer had no depth buffer, so it used fixed-point maths and sorted polygons into ordering tables, which is why its textures wobble and polygons sometimes pop through each other.

12. Console online gaming became normal in stages: the Dreamcast shipped with a 56k modem in 1998 in Japan and 1999 elsewhere, and Xbox Live launched on 15 November 2002 with a broadband-only, subscription, unified-identity model that everyone else eventually copied.
13. As of August 2026 the current landscape is: PlayStation 5 and PlayStation 5 Pro, Xbox Series X and Series S, Nintendo Switch 2 released 5 June 2025, a PC market dominated by Steam, a handheld PC category led by the Steam Deck, and a mobile market larger by player count than all of them together.

■ 45.2.6 WORDS - remember these

1. Racing the beam — changing the picture while the screen draws it — writing video registers in sync with horizontal scan timing on a frame-buffer-less system.
2. Sprite — a small movable picture — a hardware-composited image layer with its own position registers, independent of the background.
3. Lockout chip — the part that says which cartridges are allowed — a challenge-response micro-controller pair gating the console reset line.
4. BSP tree — a way of pre-sorting a level — a binary space partitioning structure allowing back-to-front or front-to-back traversal from any point.
5. Shareware — try the first part free — a distribution model splitting a game into a freely copyable episode and paid episodes.
6. Fixed point — fractions stored as whole numbers — a numeric format such as 16.16 with an implicit binary scaling factor, used where no FPU exists.

45.3 The game loop in detail

■ 45.3.1 PLAIN - in simple words

1. The loop has four jobs, always in this order.
2. **Input:** find out what the player is doing right now. Which keys are down, where the mouse moved, how far the trigger is pulled.
3. **Update:** move the world forward by a small slice of time. Apply gravity, move characters, run the enemy brains, check what hit what.
4. **Render:** build the picture that shows the world in its new state.
5. **Present:** hand that picture to the screen.
6. Then start again from input.
7. There is one hard question in the middle of this: how big is that “small slice of time”?
8. The obvious answer is “however long the last frame took”. That is called a **variable timestep**, and it is wrong for physics.
9. The reason is that physics maths gives different answers for different slice sizes. Big slices lose detail and small slices keep it.
10. So if your machine is fast and mine is slow, a jump that clears a gap on your machine can fail on mine. That is not acceptable.
11. The fix is to always advance physics by exactly the same slice, for example one sixtieth of a second, no matter how long the frame took.
12. If the frame took longer, you run the physics slice more than once. If it took less, sometimes you do not run it at all.
13. That is called a **fixed timestep**, and it is what almost every serious game does.

■ 45.3.2 PLAIN – a picture in your head

1. Imagine filling a bucket from a tap, and every time the bucket is full you must pour it into a barrel.
2. The tap is real time flowing past. The bucket is a store of time you have not simulated yet. The bucket's size is your fixed step, say 16.67 thousandths of a second.
3. Each time round the loop, you measure how much water came out of the tap since last time and add it to the bucket.
4. Then, while the bucket has at least one bucketful in it, you empty one bucketful into the barrel. That emptying is one physics step.
5. Sometimes you empty it twice, because the frame was slow and two bucketfuls collected. Sometimes you empty it zero times, because the frame was fast.
6. The leftover water in the bucket is real time that has passed but has not been simulated. It is always less than one full bucket.
7. That leftover is not wasted. You use it to decide how far between the last two physics positions to draw the picture. Half a bucket left means draw halfway between.
8. That last trick is called **interpolation**, and it is why a game with a 60-step physics simulation can look perfectly smooth at 144 frames per second.

Where this comparison breaks: water is continuous and time in a computer is not. Real clocks are read as whole counts of a hardware timer. Also, a real bucket cannot overflow into an infinite queue; a game must put a limit on how much time it will try to catch up, or a slow frame causes a longer frame, which causes a longer one, and the game locks up. That failure is called the **spiral of death** and the fix is to clamp the measured frame time.

■ 45.3.3 PLAIN – a worked example

1. Fixed step is 16.67 ms, which is 1/60 of a second. The screen refreshes at 144 Hz, so a frame arrives every 6.94 ms.
2. Start with the bucket empty. Follow ten frames.

Frame	Real time added	Bucket after	Steps run
1	6.94 ms	6.94 ms	0
2	6.94 ms	13.88 ms	0
3	6.94 ms	4.15 ms	1
4	6.94 ms	11.09 ms	0
5	6.94 ms	1.36 ms	1
6	6.94 ms	8.30 ms	0
7	6.94 ms	15.24 ms	0
8	6.94 ms	5.51 ms	1

3. Over eight frames, three physics steps ran. That is $3 \times 16.67 = 50$ ms of simulated time for $8 \times 6.94 = 55.5$ ms of real time, with 5.5 ms still in the bucket. The books balance.
4. Notice that the picture changed on all eight frames, even the five where no physics ran. Those five frames were drawn by interpolating.
5. On frame 4 the bucket held 11.09 ms out of a 16.67 ms step, so the interpolation factor was $11.09 / 16.67 = 0.665$. Draw the world 66.5 percent of the way from the previous physics position to the current one.
6. Now the frame budget arithmetic, which you should know by heart.

Target rate	Budget per frame	Ticks per second
30 fps	33.33 ms	30
60 fps	16.67 ms	60
90 fps	11.11 ms	90
120 fps	8.33 ms	120
144 fps	6.94 ms	144
240 fps	4.17 ms	240

- The budget is 1000 divided by the target rate. That budget covers everything: input, physics, AI, animation, audio submission, all the render commands, and the driver's own work.
- At 144 fps you have 6.94 ms total. If your shadow pass alone takes 3 ms you have already spent 43 percent of the frame on shadows.

■ 45.3.4 PLAIN – what is really happening inside

- Here is the real loop, with the fixed step, the accumulator and the interpolation. Every line is explained underneath.

```
const double dt = 1.0 / 60.0;    /* fixed step: 16.667 ms */
double accumulator = 0.0;        /* unsimulated time store */
double t = 0.0;                  /* total simulated time */
double currentTime = now();       /* clock at loop start */

while (running) {
    double newTime = now();
    double frameTime = newTime - currentTime;
    if (frameTime > 0.25) frameTime = 0.25; /* clamp */
    currentTime = newTime;

    accumulator += frameTime;

    poll_input();

    while (accumulator >= dt) {
        previous_state = current_state;
        simulate(&current_state, t, dt);
        t += dt;
        accumulator -= dt;
    }

    double alpha = accumulator / dt;
    State view = lerp(previous_state, current_state, alpha);

    render(view);
    present();
}
```

- dt is the fixed simulation step. It never changes at run time. Every physics calculation in the game is written knowing this exact number.
- accumulator holds real time that has passed but has not yet been fed to the simulation. It is always between 0 and dt at the bottom of the loop.
- t is total simulated time. Effects, animations and networked timestamps are keyed off this, not off the wall clock.
- currentTime remembers when the previous loop iteration started, so we can subtract and get the elapsed time.

6. `now()` must be a monotonic high-resolution clock. On Windows that is `QueryPerformanceCounter`; on Linux `clock_gettime(CLOCK_MONOTONIC)`; on macOS `mach_absolute_time`. Never use wall-clock time, because it can jump backwards when the machine syncs with a time server.
7. `frameTime` is how long the previous whole frame took, in seconds.
8. The clamp to 0.25 seconds is the spiral-of-death guard. If the process was suspended, or a big file loaded, or you dragged the window, `frameTime` might be 8 seconds. Without the clamp the loop would try to run 480 physics steps in one frame, take even longer, and never recover. With the clamp the game simply loses that time. Slow motion is better than a freeze.
9. `poll_input` reads the devices once per frame, before any simulation, so that every physics step inside this frame sees the same input.
10. The inner while is the fixed-step loop. It runs zero, one or more times, consuming one `dt` of accumulator each time.
11. `previous_state = current_state` saves the state before stepping, so we have two snapshots to interpolate between.
12. `simulate` advances the world by exactly `dt`. It never sees the real frame time and never needs to.
13. `alpha` is the leftover fraction, between 0 and 1. It says how far past the last physics step the display moment is.
14. `lerp` is linear interpolation: $a + (b - a) * \alpha$. Applied to positions it is straightforward; applied to rotations you use spherical interpolation of quaternions instead.
15. `render` draws the interpolated view, not the simulation state. This is the line most beginners get wrong, and getting it wrong causes visible judder when the frame rate is not an exact multiple of the tick rate.
16. `present` hands the buffer to the display and, with vertical sync on, blocks until the next refresh.

■ 45.3.5 TECHNICAL - the engineer's version

1. The accumulator pattern above is the one popularized by Glenn Fiedler's article "Fix Your Timestep", first published in 2004 and widely revised since. It is a convention, not a standard, but it is close to universal.
2. Fixed step is required whenever the integration scheme is not step-size-invariant, which is every scheme used in games. Semi-implicit Euler with a 33 ms step and with a 4 ms step give different trajectories, and constraint solvers converge differently at different step sizes.
3. Typical simulation rates: 60 Hz is the default for most engines; 30 Hz for large-world console titles that trade physics fidelity for CPU; 120 Hz or 240 Hz for racing, fighting and VR titles; 128 Hz for competitive shooters on the server.
4. Substepping is separate: a solver may internally split one 16.67 ms step into 4 substeps of 4.17 ms to improve stability for fast objects, while the game still ticks at 60 Hz.
5. **Frame time** is the wall-clock duration of one frame in milliseconds. **Frames per second** is 1000 divided by the mean frame time. Frame time is the primary measurement; frames per second is a derived, compressed and lossy summary of it.
6. Why the average lies. Take 100 frames: 99 of them at 8.00 ms and one at 100 ms. Total elapsed is $99 \times 8 + 100 = 892$ ms. Mean frame time is 8.92 ms, so the reported average is 112.1 fps. But the player saw one frame held on screen for a tenth of a second, which reads as a clear hitch.
7. This is why reviewers and profilers report percentiles. The **1 percent low** is, by the most common convention, the average frame rate of the slowest 1 percent of frames; some tools instead report the frame time at the 99th percentile. The two definitions give different numbers and the industry has not settled on one. When comparing figures, check which was used.
8. In the example above, the 1 percent low is 10 fps by either definition, against a 112 fps average.

That gap is exactly the complaint people describe as “the frame rate is fine but it feels bad”.

9. **Frame pacing** is a separate problem from frame rate. A sequence of frame times of 8, 8, 8, 8 ms feels smoother than 4, 12, 4, 12 ms even though both average 8 ms, because the second sequence presents images at uneven intervals against an even display refresh.
10. The full click-to-photon latency chain, with realistic numbers for a PC at 60 Hz with vertical sync and a 2-frame render queue:

Stage	Typical latency
Mouse sample and USB	1 to 8 ms
Input poll to sim	0 to 16.7 ms
Simulate and render	16.7 ms
Render queue depth	16.7 to 33.4 ms
Display scanout and panel	5 to 20 ms

11. That totals roughly 40 to 95 ms, which is why competitive players disable vertical sync, cap frames just below the refresh rate, use variable refresh, and enable low-latency modes such as NVIDIA Reflex (2020) or AMD Anti-Lag, all of which mainly work by shortening the render queue.
12. Measuring tools: PresentMon and its front ends on Windows give per-frame present timings and a latency estimate; stat unit and Unreal Insights in Unreal Engine; the Unity Profiler; RenderDoc and PIX for GPU timings; perf and Tracy for CPU timeline capture.
13. The distinction between CPU frame time and GPU frame time matters. If the GPU is the bottleneck the CPU finishes early and waits; if the CPU is the bottleneck the GPU starves. Profilers report both, and the larger one is your frame time.

```

frame N          frame N+1          frame N+2
CPU [sim|render cmds] [sim|render cmds] [sim|...
GPU      [ draw N ]      [ draw N+1 ]      [ ...
DISP                [show N ]      [show N+1]

CPU builds frame N while GPU draws N-1: this is pipelining.
It raises throughput and adds one frame of latency.
```

■ 45.3.6 WORDS – remember these

1. Timestep — the slice of time simulated at once — the Δt passed to the integrator each simulation tick.
2. Fixed timestep — always the same slice — a constant Δt decoupled from frame duration, required for reproducible integration.
3. Accumulator — the store of unsimulated time — a scalar holding elapsed real time not yet consumed by fixed steps.
4. Interpolation — drawing between two known positions — blending previous and current simulation states by the leftover accumulator fraction.
5. Spiral of death — the game falls further and further behind — unbounded catch-up stepping where step cost exceeds real time elapsed.
6. Frame time — how long one frame took — wall-clock duration in milliseconds, the primary performance measurement.
7. 1 percent low — how bad the worst frames are — the mean frame rate of the slowest one percent of frames, or the 99th-percentile frame time.
8. Frame pacing — whether frames arrive evenly — the variance of frame intervals relative to the display refresh period.

45.4 How a game is built, as a project

■ 45.4.1 PLAIN – in simple words

1. A game is not written. It is built, the way a film is shot or a building is put up, by people with very different skills.
2. The **designer** decides what the player does and why it is fun. Levels, rules, numbers, pacing.
3. The **programmer** makes the machine do it. Gameplay code, engine code, graphics code, tools for everyone else, network code.
4. The **artist** makes what you see. Concept art first, then the actual models, textures, environments and characters.
5. The **animator** makes things move: walking, reloading, dying, a door opening, a face speaking.
6. The **audio** people make what you hear: footsteps, gunfire, wind, music, voices.
7. **Quality assurance**, usually called QA, plays the game deliberately and methodically to find what is broken, and writes it down precisely.
8. The **producer** keeps the whole thing on a schedule and decides what gets cut when there is not enough time. There is never enough time.
9. On a big game there are more roles: technical artist, user interface designer, writer, localization, build engineer, live operations, community.
10. The single most important fact about this list is that programmers are a minority. On a large modern game, roughly one person in five writes code.

■ 45.4.2 PLAIN – a picture in your head

1. Think of a restaurant opening.
2. Somebody decides what kind of restaurant it is and writes the menu. That is the designer.
3. Somebody builds the kitchen and makes the equipment work. That is the programmer.
4. Chefs cook. Decorators decorate. Someone chooses the music. Those are the artists, animators and audio people.
5. Before opening, friends come in and eat for free and say what was wrong. That is QA and playtesting.
6. Someone watches the money and the calendar and says “the dessert menu is cut, we open in three weeks”. That is the producer.
7. And a restaurant is never finished on opening night. The first month is fixing what real customers break. That is the launch patch and live operations.

Where this comparison breaks: a restaurant can open with a small menu and add dishes. A game usually has to ship one enormous simultaneous thing, and a missing piece in the middle stops everything. Also, a bad dish affects one diner; a bad save-file bug affects everyone at once, everywhere, at 3 a.m.

■ 45.4.3 PLAIN – a worked example

1. The phases, in order, with what each one actually means.
2. **Concept**: a few people, a few weeks to a few months. A document, some sketches, maybe a rough playable thing. Most concepts die here, and that is the cheapest place to die.
3. **Pre-production**: build the risky bits. Answer “can we even do this”. Small team, six months to two years on a big project.
4. **Vertical slice**: one small part of the game at full final quality. Ten minutes that look and feel exactly like the finished game. It exists to prove the vision and to unlock funding.
5. **Production**: the big, expensive middle. Make all the content. The team is at its largest here.

6. **Alpha:** feature complete. Every system exists, even if content is missing and it is full of bugs.
7. **Beta:** content complete. Everything is in. From here you only fix.
8. **Certification:** for consoles, the platform holder tests the build against a long checklist. See section 45.14.
9. **Gold and launch:** the build is locked and manufactured or uploaded.
10. **Live operations:** patches, seasons, events, balance changes, and sometimes years of new content.

Scale	Team size	Time	Budget range
Solo hobby	1	6 to 24 months	0 to 20k
Small indie	2 to 8	1 to 3 years	50k to 1M
Mid-size (AA)	30 to 80	2 to 3 years	5M to 30M
AAA	200 to 600	4 to 6 years	80M to 300M

11. Those budget figures are in United States dollars and exclude marketing, which on a AAA title is often as much again as development.

■ 45.4.4 PLAIN - what is really happening inside

1. The reason production is expensive is content, not code.
2. A gun in a modern shooter is not one job. It is a concept sketch, a high-detail model, a low-detail game model, several texture maps, a rig, reload and inspect and fire animations, muzzle flash effects, four or five layered sounds, a tuning pass on damage and recoil, a first-person and a third-person version, and an entry in the user interface.
3. That is roughly two to six person-weeks for one weapon, and a game may have forty.
4. Multiply that pattern across characters, environments, vehicles and cutscenes and you get the two hundred million dollars.
5. Code is a smaller share, but it gates everything: nobody can build a level until the level tools work.
6. This is why the tools programmers are quietly the most valuable people on a big team. A one-second improvement in the editor's save time, times 200 people, times 50 saves a day, is 2.8 hours of team time per day.
7. And this is why the schedule always slips at the end. Content work is parallel and estimable. Integration, bug fixing and certification are serial and are not.

■ 45.4.5 TECHNICAL - the engineer's version

1. Real documented budgets, from court filings and leaks rather than press releases, which is why they are trustworthy.

Game	Reported cost	Source and year
The Last of Us Part II	~220M, ~200 devs	Court exhibit, 2023
Horizon Forbidden West	~212M, ~300 devs	Court exhibit, 2023
Grand Theft Auto V	~265M with marketing	Reported, 2013
Cyberpunk 2077	~316M with marketing	CD Projekt, 2021
Stardew Valley	one person, 4.5 years	Developer, 2016

2. The Sony figures became public in June 2023 through poorly redacted documents filed in the United States Federal Trade Commission case about the Microsoft acquisition of Activision Blizzard. The Cyberpunk 2077 number is roughly 1.2 billion Polish zloty of combined development and marketing spend, converted at the rates of the time, and should be treated as approximate.

3. Team composition on a typical 250-person AAA project is roughly: 45 to 60 artists, 30 to 45 engineers, 20 to 30 designers, 20 to 30 animators, 10 to 15 audio, 30 to 60 QA, plus production, writing, user interface and localization. External outsourcing studios may add another 200 people who never appear on the internal headcount.
4. Costs per head, fully loaded with salary, benefits, hardware, software and office, run roughly 120,000 to 250,000 US dollars per person-year in North America and Western Europe, less elsewhere. A 250-person team for four years at 160,000 is 160 million dollars, which is where these budgets come from.
5. Now the honest part, because the reader asked for honesty.
6. **Crunch** means long mandatory overtime near milestones. Sixty to eighty hour weeks for months are documented at many studios. It is often unpaid for salaried staff. It correlates with burnout, health damage and people leaving the industry in their thirties.
7. The industry has argued about it publicly since the 2004 “EA Spouse” open letter, which led to overtime lawsuits and settlements at Electronic Arts. It recurred publicly around Rockstar Games in 2018 and CD Projekt Red in 2020.
8. The structural cause is not villainy. It is that games are fixed-date, fixed-scope products with an unpredictable final integration phase, sold in a market with a hit-driven revenue distribution. Cutting scope is the only real fix and it is the hardest decision to make.
9. **Layoffs** became the defining industry story of the 2020s. Trackers recorded roughly 10,500 games industry job losses in 2023 and roughly 14,600 in 2024, with further large numbers through 2025 and 2026. Studios have been closed within months of shipping well-reviewed games.
10. **Unionization** started to take hold in the same period, with organized bargaining units at several large studios from 2023 onward. Whether this changes crunch materially is still open.
11. Where experts disagree: some argue budgets are unsustainable and the AAA model must shrink; others argue the top few titles will keep growing and the middle will disappear. The evidence in 2026 supports both, because the middle has in fact been shrinking while the top has grown.

■ 45.4.6 WORDS – remember these

1. Vertical slice — a small piece at final quality — a fully polished representative segment used to validate the design and secure funding.
2. Alpha — every feature exists — feature complete, content incomplete.
3. Beta — everything is in — content complete, bug fixing only.
4. Gold master — the locked build — the final candidate submitted for manufacture or platform release.
5. Crunch — mandatory long hours — sustained overtime at milestone boundaries, often uncompensated for salaried staff.
6. Live operations — running the game after launch — the ongoing content, balance, event and monetization pipeline for a service game.

45.5 Game engines

■ 45.5.1 PLAIN – in simple words

1. An **engine** is the pile of reusable machinery that every game needs and that is not about your particular game.
2. It draws things. It plays sounds. It reads controllers. It works out what hit what. It loads files. It runs on Windows and consoles and phones.
3. It also gives you an **editor**: a program where you place objects, light a scene and press play, without writing code for any of it.
4. What it does not give you is a game. It gives you the ability to make one.
5. Studios use one instead of writing their own for a plain reason: writing your own costs several engineer-years before you can show anything.
6. Worse, you must then keep it working on five platforms as those platforms change, forever, with a team of maybe three people.
7. Big studios that do write their own do it because they need something the commercial engines do not do well: enormous open worlds, very specific physics, or an existing pile of tools twenty years deep.

■ 45.5.2 PLAIN – a picture in your head

1. Think about building a house.
2. You could make your own bricks, mill your own timber and wind your own wire. People did, once.
3. Today you buy standard bricks, standard wire in standard colours, standard window units and standard plumbing fittings.
4. The standard parts are not what makes your house yours. The plan is.
5. But the standard parts decide a lot: what shapes are cheap, what is awkward, and what will simply not fit.
6. An engine is the standard parts. It makes some games cheap to build and others painful, and you feel that pressure the whole way through.

Where this comparison breaks: bricks do not change their price after you have built half the house. Engine licences can, and in 2023 one of them tried to. Section 45.5.5 tells that story exactly.

■ 45.5.3 PLAIN – a worked example

1. What is actually inside an engine, listed as the parts you would otherwise have to write.

Part	What it does
Renderer	Turns a scene into pixels
Physics	Moves bodies, finds collisions
Audio	Mixes and positions sound
Input	Reads keyboard, mouse, pads
Scene system	Holds and finds objects
Asset pipeline	Imports and converts content
Scripting	Runs gameplay code safely
Editor	Lets non-programmers build
Build system	Packages for each platform
Networking	Replicates state to clients

2. A bare version of the first four is roughly 50,000 to 150,000 lines of code and two to four engineer-years for a competent small team.
3. A version with an editor, a working asset pipeline and console support is a decade of work by a real team. That is the honest scale.
4. Which is why a two-person studio using Godot ships in eighteen months, and a two-person studio writing an engine ships never.

■ 45.5.4 PLAIN – what is really happening inside

1. Two ways of organizing the objects in a game world dominate, and the difference is about how the computer's memory works.
2. A **scene graph** is a tree. A car is a node; its wheels are children of the car; a mirror is a child of a door which is a child of the car.
3. Moving the car moves everything under it automatically, because each child position is stored relative to its parent.
4. This is natural for humans and it matches how artists think. It is what Unity's GameObject hierarchy and Godot's node tree give you.
5. Its weakness is speed. Each object is a separate lump of memory with pointers to its children, scattered across the heap.
6. When the game walks 10,000 objects to move them, it jumps all over memory, and the processor spends most of its time waiting for memory rather than computing.
7. An **entity-component-system**, always abbreviated ECS, is the answer.
8. An **entity** is just a number, an identity with no data of its own.
9. A **component** is a small plain lump of data, such as a position or a health value, stored in one big array with all the other positions.
10. A **system** is a function that runs over every entity that has a particular set of components, walking those arrays straight through.
11. Because all the positions sit next to each other in memory, the processor can fetch them in order and predict what comes next. That is the whole trick, and it can be several times faster.

■ 45.5.5 TECHNICAL – the engineer's version

1. The cache arithmetic that makes ECS worth the awkwardness. A cache line on every current x86-64 and ARM64 processor is 64 bytes.
2. Suppose 10,000 objects, each a 256-byte object, and you want to add velocity to position: 12 bytes read, 12 bytes written per object.
3. In the object-oriented layout, each object touches at least one 64-byte line, usually two because the fields straddle. Memory traffic is at least $10,000 \times 64 = 640$ KB, in an access pattern the hardware prefetcher cannot predict.
4. In the component layout, positions are one contiguous 120 KB array and velocities another. Total traffic is 240 KB, perfectly sequential, fully prefetched. That is under half the traffic and far better latency hiding.
5. Blizzard's Overwatch is the best-documented shipped example. In Timothy Ford's GDC 2017 talk "Overwatch Gameplay Architecture and Netcode" the client is described as having about 46 systems and about 103 component types, with only three systems involved in gameplay networking.
6. Now the engines themselves, as of August 2026.

Engine	Language	Cost model
Unity 6	C#	Free under 200k revenue

Engine	Language	Cost model
Unreal Engine 5	C++ and Blueprints	Free under 1M lifetime
Godot 4	GDScript, C#, C++	MIT licence, free
In-house	usually C++	Salaries and years

7. **Unity** was announced at Apple’s Worldwide Developers Conference in June 2005 as a Mac-only engine. It now targets more than nineteen platforms. It scripts in C#; the old UnityScript was deprecated in August 2017. Unity Personal is free below 200,000 US dollars of annual revenue and funding. Unity Pro is required above that; its list price was 2,200 dollars per seat per year from 1 January 2025 and 2,310 dollars per seat per year from 12 January 2026. Unity Enterprise is required above 25 million dollars.
8. The 2023 episode, stated precisely. On 12 September 2023 Unity announced a **Runtime Fee**: a per-install charge on games above revenue and install thresholds, to begin 1 January 2024. The reaction was severe, because the fee applied retroactively to already-shipped games built under different terms, was based on installs rather than sales, and had no clear way to count them.
9. Unity revised the terms on 22 September 2023. Chief executive John Riccitiello retired on 9 October 2023. Matthew Bromberg became chief executive in May 2024. On 12 September 2024, exactly one year after the announcement, Unity cancelled the Runtime Fee outright and returned to seat-based subscriptions, doubling the Personal threshold from 100,000 to 200,000 dollars and raising Pro by 8 percent and Enterprise by 25 percent instead.
10. The lesson is not “Unity is bad”. It is that a licence is part of your technology stack, it is a contract with a company that has shareholders, and it can change under a project that is already three years in. Read the terms, keep an exit plan, and prefer terms fixed at the version you ship.
11. **Unreal Engine** licensing has moved the other way over time. In March 2014 Unreal Engine 4 was 19 dollars per month plus a 5 percent royalty. In March 2015 it became free with a 5 percent royalty above 3,000 dollars per calendar quarter. In May 2020, retroactive to 1 January 2020, the exemption became 1,000,000 dollars of lifetime gross revenue per title. Revenue from the Epic Games Store is royalty-free. Since 2024 there is also a seat licence, 1,850 dollars per seat per year, for companies over 1 million dollars doing non-game work such as film and architecture. Unreal Engine 5.7 shipped in November 2025.
12. **Godot** is MIT licensed and has been open source since February 2014. There is no fee, no royalty and no threshold, and the licence cannot be revoked for a version you already have. The latest stable release at the time of writing is 4.7.1 from 13 July 2026. It is developed by the Godot Foundation on donations, which is its own risk: a much smaller budget than either commercial competitor.
13. **In-house engines** still dominate the top end: RE Engine at Capcom, Decima at Guerrilla, Frostbite at Electronic Arts, Anvil at Ubisoft, id Tech at id Software, Creation at Bethesda, Source 2 at Valve, and Rockstar’s RAGE. The reason is usually a genre the commercial engines handle badly, plus twenty years of tooling that a team already knows.
14. Where experts disagree: some argue in-house engines are a strategic asset; others argue they are a tax that diverts engineers from the game. Several large publishers have moved to Unreal Engine 5 since 2022, which is evidence for the second view but not proof.

■ 45.5.6 WORDS – remember these

1. Engine — the reusable machinery — the platform layer, renderer, physics, audio, tools and pipeline shared across titles.
2. Scene graph — a tree of objects — a hierarchy of transforms where child transforms are expressed relative to their parent.

3. ECS — data in flat arrays with functions over them — entity-component-system architecture separating identity, plain data and behaviour for cache locality.
4. Cache line — the smallest chunk memory moves in — 64 bytes on current x86-64 and ARM64 processors.
5. Royalty — a share of revenue paid to the engine maker — a percentage of gross above a stated threshold, as in Unreal's 5 percent above 1 million.
6. Runtime fee — a charge per install — the per-installation model Unity announced in September 2023 and cancelled in September 2024.

45.6 Assets and the content pipeline

■ 45.6.1 PLAIN – in simple words

1. Everything in a game that is not code is an **asset**: models, pictures, sounds, animations, fonts, levels, text.
2. Assets are made in other programs. A model is made in a modelling tool, a texture in a painting tool, a sound in an audio tool.
3. Those programs save in formats designed for editing, not for playing. They are big, flexible and slow to read.
4. So there is a step in the middle that converts everything into the exact form the target machine wants. That step is the **asset pipeline**.
5. It compresses textures into a format the graphics chip can read directly. It simplifies models into several versions for different distances. It packs thousands of small files into a few big ones.
6. It also computes things in advance that would be too slow to compute while playing: how light falls on a static wall, where characters can walk, how sound travels in a room.
7. Doing all that for an entire game is why a full build takes hours.
8. In Unreal Engine this conversion step is called **cooking**, and the phrase “cook the content for PlayStation” means “convert every asset into the exact form that machine wants”.

■ 45.6.2 PLAIN – a picture in your head

1. Think of a professional kitchen before service.
2. The raw ingredients arrive as whole vegetables, whole fish, sacks of flour. Flexible, but useless during a busy service.
3. So in the morning the kitchen does **prep**: chopping, portioning, making stocks, part-cooking things that take an hour.
4. During service nothing takes an hour, because the hour already happened.
5. Prep is the asset pipeline. Raw ingredients are the source files. The prepped trays are the cooked assets.
6. And if the chef changes the menu at 5 p.m., all the prep for that dish is wasted and must be redone. That is why artists hate late changes.

Where this comparison breaks: a kitchen preps for one evening. A game preps for every machine it will ever run on, and each target wants different prep, so the same source is prepped four or five times in parallel.

■ 45.6.3 PLAIN – a worked example

1. Follow one texture from painting to screen.
2. An artist paints a 4096 by 4096 colour map in a painting program and saves a 32-bit-per-pixel file.
3. Raw size: $4096 \times 4096 \times 4$ bytes = 67,108,864 bytes, which is 64 mebibytes for one texture.

4. The pipeline generates **mipmaps**, smaller copies at half size, quarter size and so on down to one pixel. That adds about 33 percent, so about 85 mebibytes.
5. Then it compresses to a format the graphics chip can sample directly. On PC and console that is usually BC7, which is one byte per pixel.
6. BC7 size: $4096 \times 4096 \times 1 = 16$ mebibytes, plus mipmaps, about 21.3 mebibytes. That is a quarter of the raw size and it stays compressed in video memory, which is the point.
7. On a phone the same texture is compressed to ASTC instead, because mobile graphics chips do not read BC7.

Format	Target	Bits per pixel
RGBA8 uncompressed	none	32
BC1 (DXT1)	PC, console	4
BC7	PC, console	8
ETC2 RGBA	Android, GLES 3	8
ASTC 4x4	mobile, Apple	8
ASTC 6x6	mobile, Apple	3.56

8. Now do that for 4,000 textures. At 64 mebibytes raw each, the source is 256 gibibytes; cooked to BC7 with mipmaps it is about 85 gibibytes; after packaging and further lossless compression, perhaps 45 gibibytes on disk.
9. And the compression itself is not free. High-quality BC7 encoding is roughly 0.5 to 5 seconds per 4K texture on one core. Four thousand of them is 30 minutes to 5 hours of single-core work, which is why studios run distributed build farms and shared caches.

■ 45.6.4 PLAIN - what is really happening inside

1. The full pipeline, in order, for a 3D character.
2. **Model:** the artist builds a mesh, a set of points joined into triangles. A hero character might be 60,000 to 150,000 triangles.
3. **UV unwrap:** the artist flattens the surface onto a square so a two-dimensional picture can be painted onto it.
4. **Texture:** several pictures are painted, not one. Base colour, roughness, metalness, a normal map that fakes small bumps, sometimes ambient occlusion and emissive.
5. **Rig:** a skeleton of bones is built inside the mesh, and each vertex is assigned weights saying which bones move it and how much.
6. **Animate:** the animator poses the skeleton over time, or motion capture data is cleaned up and retargeted onto it. Each clip is a list of bone rotations at keyframes.
7. **Export:** everything is written to an interchange format, usually FBX, or increasingly glTF 2.0 or USD.
8. **Import:** the engine reads that file and converts it. Triangles are reordered for the graphics chip's vertex cache. Normals and tangents are computed. Animation curves are compressed, often to about a tenth of their size, by dropping keys within an error tolerance.
9. **Level of detail:** simplified copies are generated. A common ladder is 100 percent, 50 percent, 25 percent and 12 percent of the triangles, swapped by screen size.
10. **Bake:** things that never change are computed once. Light maps for static geometry, a navigation mesh for where characters can walk, occlusion data for what can see what.
11. **Package:** everything is written into a few large archive files, sorted so that things loaded together sit next to each other on disk, and compressed with a fast codec.
12. **Ship:** the archive plus the executable is what the player downloads.

■ 45.6.5 TECHNICAL – the engineer’s version

1. Tools in common use: Blender (free, GPL) and Autodesk Maya for modelling and animation; ZBrush for sculpting; Adobe Substance 3D Painter and Designer for textures; Houdini for procedural content and effects; Photoshop or Krita for two-dimensional art; Reaper, Pro Tools and iZotope RX for audio.
2. Interchange formats: FBX (Autodesk, proprietary but ubiquitous), glTF 2.0 (Khronos, 2017, open and runtime-friendly), USD (Pixar, open-sourced 2016, now standard for large scene assembly), Alembic for baked geometry caches, OBJ for simple static meshes.
3. Cooking, precisely, means: convert every source asset into a platform-specific, engine-specific binary that can be memory-mapped or streamed with minimal parsing, and drop all editor-only data.
4. Why full builds take hours, itemized for a large project:

Stage	Typical duration
Code compile and link	20 to 90 minutes
Shader compilation	1 to 6 hours
Texture compression	30 to 180 minutes
Lightmap baking	1 to 12 hours
Packaging and compress	20 to 90 minutes

5. Shader compilation is usually the worst offender because of **permutations**. One material shader with 12 independent boolean features is 2 to the power 12, which is 4,096 variants, and a real project has hundreds of such shaders. Large titles ship tens of thousands to hundreds of thousands of compiled pipeline states.
6. The countermeasures are a **derived data cache** (a shared server storing the result of every conversion keyed by a hash of the input plus the settings, so nobody converts the same asset twice), distributed compilation systems such as Incredibuild or FASTBuild, and incremental cooking that touches only changed assets. With a warm cache, an incremental cook is minutes rather than hours.
7. Runtime compression codecs matter as much as the format. Oodle Kraken, Zstandard and LZ4 are the common choices; consoles decompress in hardware, with the PlayStation 5 Kraken decompressor rated at roughly 5.5 gigabytes per second of compressed input and the Xbox Series X BCPack path similar in spirit.
8. Virtualized geometry changes the level-of-detail story. Unreal Engine 5’s Nanite, introduced in 2021, stores meshes as a hierarchy of triangle clusters and selects detail per cluster per pixel at run time, removing hand-authored LODs for static opaque geometry. It does not remove the need for LODs on skinned characters, foliage or translucent materials, which is a caveat often skipped in marketing material.
9. Virtual texturing solves the matching problem for textures: only the pages of a texture actually visible at the current screen resolution are resident in video memory, so a scene can reference far more texture data than fits.
10. Observation tools: Unreal’s `-run=DerivedDataCache` and cook logs, Unity’s Build Report and Addressables analysis, `ls -la` on the packaged archives, RenderDoc to see which textures a frame actually sampled, and the platform’s own memory profiler to see what is resident.

■ 45.6.6 WORDS – remember these

1. Asset — anything that is not code — content data authored in an external tool and consumed by the engine.
2. Cooking — converting content for one machine — transforming source assets into a platform-specific runtime binary form.
3. Mipmap — smaller copies of a texture — a precomputed pyramid of half-resolution levels used to avoid aliasing at distance.
4. Level of detail — simpler versions for far away — a set of decimated meshes selected by projected screen size.
5. Baking — computing it in advance — precomputing lighting, navigation or occlusion data at build time rather than at run time.
6. Derived data cache — a shared store of conversion results — a server keyed by input hash so that each conversion is performed once for the whole team.
7. Shader permutation — one variant of a shader — a compiled pipeline state for one specific combination of features and platform.

45.7 Physics and collision

■ 45.7.1 PLAIN – in simple words

1. A physics system answers two questions, sixty times a second: where is everything now, and did anything touch anything.
2. Each movable object is a **rigid body**: a shape that does not bend, with a mass, a position, a speed and a spin.
3. Forces such as gravity change its speed. Its speed changes its position. Doing that arithmetic step by step is called **integration**.
4. Finding touches is done in two passes, because checking every pair is far too slow.
5. The first pass, the **broad phase**, cheaply throws away pairs that are obviously nowhere near each other, using a grid or a tree of boxes.
6. The second pass, the **narrow phase**, does the real shape test on the few pairs that survived.
7. When two things do touch, the system pushes them apart and changes their speeds. How bouncy that is depends on a number between 0 and 1 called **restitution**.
8. Ropes, hinges, doors and ragdolls are the same machinery plus rules that say “these two bodies must stay joined here”. Those rules are constraints.

■ 45.7.2 PLAIN – a picture in your head

1. Imagine a large hall with 1,000 people walking around, and you must report every collision.
2. Checking every pair means $1,000 \times 999 / 2 = 499,500$ checks. At sixty times a second that is 30 million checks a second, for nothing.
3. So you chalk the floor into a grid of squares. Each person reports which square they are in.
4. Now you only compare people in the same square or in touching squares. Almost every pair is eliminated without looking at them.
5. That is the broad phase. The remaining few hundred pairs get the careful look, which is the narrow phase.

Where this comparison breaks: people are all about the same size, so one grid square size works. Game objects range from a bullet to a cathedral, and one grid size cannot serve both. Real engines therefore use trees of boxes that adapt, not a flat grid.

■ 45.7.3 PLAIN – a worked example

- Two balls hit head-on. Ball A weighs 2 kg and moves at +5 metres per second along x. Ball B weighs 1 kg and is still. Restitution is 0.5.
- The collision normal n points from A to B, so it is (1, 0, 0).
- Relative velocity along the normal: $v_{rel} = (v_B - v_A) \cdot n = 0 - 5 = -5$. Negative means they are approaching, so we must respond.
- Impulse magnitude uses this formula:

$$j = \frac{-(1 + e) \cdot v_{rel}}{1/m_A + 1/m_B}$$

- Put the numbers in: $j = -(1 + 0.5)(-5) / (1/2 + 1/1) = 7.5 / 1.5 = 5$. The unit is kilogram metres per second.
- Apply it: $v_A' = 5 - j/m_A = 5 - 5/2 = 2.5$ m/s, and $v_B' = 0 + j/m_B = 0 + 5/1 = 5.0$ m/s.
- Check momentum. Before: $2 \times 5 + 1 \times 0 = 10$. After: $2 \times 2.5 + 1 \times 5 = 10$. Conserved, as it must be.
- Check restitution. Separation speed after is $5 - 2.5 = 2.5$, and the approach speed was 5. The ratio is 0.5, which is exactly e .
- Check energy. Before: $0.5 \times 2 \times 25 = 25$ joules. After: $0.5 \times 2 \times 6.25$ plus $0.5 \times 1 \times 25 = 6.25 + 12.5 = 18.75$ joules. 6.25 joules were lost as heat and sound, which is what a restitution below 1 means.

■ 45.7.4 PLAIN – what is really happening inside

- The integration step. The naive version, called explicit Euler, is $x = x + v \cdot dt$ then $v = v + a \cdot dt$. It quietly adds energy on every step and a spring simulated this way will slowly explode.
- Games use **semi-implicit Euler** instead: update velocity first, then use the new velocity to update position. One line swapped, and the system is stable enough for gameplay.
- Verlet integration** stores the previous position instead of velocity: $x_{new} = 2 \cdot x - x_{prev} + a \cdot dt^2$. It handles distance constraints very well, which makes it the standard choice for cloth, rope and hair.
- The broad phase in a real engine is usually a bounding volume hierarchy: a tree where each node is a box containing its children's boxes. Testing a ray or a box against it costs about log of the object count.
- The narrow phase uses exact tests for simple pairs, such as sphere against sphere, and a general algorithm for convex shapes.
- Contacts are not resolved one at a time in isolation, because a box resting on the floor has four contacts that must agree. The solver runs over all contacts several times per step, nudging each towards consistency.
- That is why a tall stack of boxes in a game jitters. The solver has a fixed number of iterations and simply runs out of time before the stack agrees.

■ 45.7.5 TECHNICAL – the engineer's version

- Broad-phase structures in use: uniform grids and spatial hashes for same-size objects; sweep and prune, which keeps objects sorted along an axis and exploits frame-to-frame coherence; quadtrees and octrees; and dynamic AABB trees, which is what Box2D, Bullet, PhysX and Jolt all use.
- Narrow phase: analytic tests for sphere, capsule and axis-aligned box pairs; the separating axis theorem for oriented boxes and convex polygons; and GJK, published by Gilbert, Johnson and Keerthi in 1988, plus the expanding polytope algorithm to recover penetration depth, for general

convex shapes. Concave meshes are decomposed into convex pieces or treated as static triangle soups.

3. Constraint solving is almost always **sequential impulses**, also called projected Gauss-Seidel, popularized by Erin Catto in Box2D and his GDC presentations from 2005 onward. Typical settings are 4 to 10 velocity iterations and 1 to 4 position iterations per step. Baumgarte stabilization or a split-impulse pass removes penetration without injecting energy.
4. Joint types: hinge (revolute), ball and socket (spherical), slider (prismatic), fixed, distance, and six-degree-of-freedom with per-axis limits and motors. A **ragdoll** is a set of rigid bodies matched to the skeleton's bones with joints whose limits mimic human range of motion, plus a blend that hands control from animation to physics on death or impact.
5. Continuous collision detection exists because a bullet moving 600 metres per second travels 10 metres in a 16.67 ms step and will tunnel straight through a wall. Solutions are conservative advancement, swept shape tests, or simply raycasting bullets instead of simulating them, which is what most shooters do.
6. Common engines and their defaults:

Engine	Physics	Default rate
Unity 6	PhysX 4 or Unity Physics	50 Hz
Unreal Engine 5	Chaos	30 or 60 Hz
Godot 4	Godot Physics or Jolt	60 Hz
Custom	Bullet, Jolt, Box2D	60 Hz

7. Determinism, stated precisely. IEEE 754-2019 fully specifies addition, subtraction, multiplication, division and square root: given the same inputs, rounding mode and precision, every conforming machine returns the same bits.
8. It does not specify $\sin$, $\cos$, $\exp$, $\log$ or pow , so two maths libraries can legally differ in the last bit. That is the first source of divergence.
9. The others are: compilers reordering floating-point expressions or contracting $a*b+c$ into a fused multiply-add with different rounding; fast-math flags; historic x87 80-bit intermediates versus SSE 64-bit; different SIMD widths taking different code paths; and multi-threaded solvers where the order contacts are summed changes between runs.
10. The consequence for multiplayer: you cannot assume two machines simulating the same inputs reach the same state. A one-bit difference in one frame becomes a visibly different world within seconds, because contact resolution amplifies small differences.
11. If you need determinism, you must either use fixed-point arithmetic, or lock down compiler flags, library versions, thread counts and instruction sets across every platform you ship on. This is why deterministic lockstep networking, described in section 45.11, is hard.

■ 45.7.6 WORDS - remember these

1. Rigid body — a solid that does not bend — a body with mass, inertia tensor, linear and angular velocity, and no internal deformation.
2. Integration — stepping the maths forward — advancing state by dt using a scheme such as semi-implicit Euler or Verlet.
3. Broad phase — the cheap first filter — spatial culling that produces a candidate pair list, typically via a dynamic AABB tree.
4. Narrow phase — the careful shape test — exact intersection tests producing contact points, normals and penetration depths.

5. Restitution — how bouncy it is — the coefficient e relating separation speed to approach speed along the contact normal.
6. Impulse — an instant change of momentum — the quantity j applied along the contact normal to resolve a collision.
7. Tunnelling — moving straight through a wall — a fast body crossing thin geometry within one timestep, fixed by continuous collision detection.

45.8 Game AI

■ 45.8.1 PLAIN – in simple words

1. Game AI is the code that decides what a character that is not you does next.
2. It has two halves: **decide what to do**, and **work out how to get there**.
3. The simplest way to decide is a **state machine**. The guard is in one state at a time: idle, patrol, investigate, chase, attack, flee. Events move it between states.
4. When there are twenty states, the arrows between them become unmanageable, so bigger games use a **behaviour tree**: a tree of small tasks read from the top, left to right, until one succeeds.
5. Another way is a **utility system**: score every possible action with a number and do the highest-scoring one.
6. The most elaborate is a **planner**: describe the goal, describe what each action requires and produces, and let the machine search for a sequence.
7. The second half, getting there, is **pathfinding**, and the standard algorithm has been the same since 1968.

■ 45.8.2 PLAIN – a picture in your head

1. Think about how you decide what to do at work.
2. Some of it is a state machine: you are in a meeting, or at your desk, or at lunch, and only certain moves make sense from each.
3. Some of it is a checklist read top down: is anything on fire, no; is anyone waiting for me, no; then do the next task on the list. That is a behaviour tree, and the order encodes priority.
4. Some of it is scoring: three things could be done, and you weigh urgency against effort and pick one. That is a utility system.
5. And occasionally you plan: to get the report signed you need the data, which needs the export, which needs access. Working backwards from the goal is what a planner does.
6. All four are used in games, often in the same game, for different characters.

Where this comparison breaks: you have a goal of your own. A game character does not. Its goal is that you enjoy fighting it, which is not the same thing as winning, and that changes every design decision underneath.

■ 45.8.3 PLAIN – a worked example

1. A-star pathfinding, by hand, on a five by five grid. S is the start, G is the goal, # is a wall. Moves are up, down, left and right, each costing 1.

y=4	.	.	.	.	.
y=3	.	.	#	.	.
y=2	S	.	#	.	G
y=1	.	.	#	.	.
y=0	.	.	.	.	.
	x0	x1	x2	x3	x4

2. A-star keeps a score for each square: $f = g + h$.
3. g is the real cost from the start to that square, counted so far.
4. h is a guess of the cost from that square to the goal. Here it is the Manhattan distance, $|4 - x| + |2 - y|$, which never overestimates because you cannot reach the goal in fewer steps than that.
5. Always expand the open square with the lowest f .
6. Start: S at (0,2) has $g=0$, $h=4$, $f=4$.
7. Expand S. Neighbours (0,1) and (0,3) get $g=1$, $h=5$, $f=6$. Neighbour (1,2) gets $g=1$, $h=3$, $f=4$.
8. Lowest f is (1,2) at 4, so expand it. (2,2) is a wall. (1,1) and (1,3) get $g=2$, $h=4$, $f=6$.
9. Now four squares are tied at $f=6$. Ties are broken towards higher g , which pushes the search forwards rather than sideways.
10. Expanding those gives (1,0) and (1,4) at $f=8$, and later (0,0) and (0,4) at $f=8$.
11. From (1,4) the search runs along the top: (2,4) $g=4$ $h=4$ $f=8$, (3,4) $g=5$ $h=3$ $f=8$, (3,3) $g=6$ $h=2$ $f=8$, (3,2) $g=7$ $h=1$ $f=8$, and finally the goal at $g=8$, $h=0$, $f=8$.
12. The answer is an eight-step path around the top of the wall. Note the heuristic said 4 at the start and the truth was 8. That is allowed, and that gap is exactly why the search had to look around.

Algorithm	Squares expanded	Path length
Breadth-first / Dijkstra	~22 of 22	8
A-star, Manhattan h	~13 of 22	8
Greedy best-first	~9 of 22	may exceed 8

13. A-star found the same shortest path as Dijkstra while looking at about forty percent fewer squares. Greedy search looks at fewer still but can return a worse path, because it ignores the cost already paid.

■ 45.8.4 PLAIN - what is really happening inside

1. A-star holds two collections: an **open set** of squares discovered but not yet expanded, and a **closed set** of squares already expanded.
2. The open set is a priority queue ordered by f , so taking the best square is fast.
3. Each square records which square it came from. When the goal is reached, you follow those links backwards to build the path.
4. Grids are convenient for teaching and terrible for large 3D worlds, because the number of squares grows with area and the paths look like staircases.
5. Real 3D games use a **navigation mesh**: the walkable floor is covered with a few thousand convex polygons instead of a million squares.
6. A-star then runs over polygons, and a second step called the funnel algorithm pulls the resulting corridor tight into a straight-line path.
7. Finally, **steering** turns that path into motion: the character seeks the next corner, slows as it arrives, avoids other characters, and blends with its animation so the feet do not slide.

■ 45.8.5 TECHNICAL – the engineer’s version

1. A-star was published in 1968 by Peter Hart, Nils Nilsson and Bertram Raphael as “A Formal Basis for the Heuristic Determination of Minimum Cost Paths” in IEEE Transactions on Systems Science and Cybernetics. It is optimal if the heuristic is **admissible**, meaning it never overestimates, and needs no node reopening if the heuristic is also **consistent**.
2. Heuristics by grid type: Manhattan for four-way movement; octile, $dx + dy + (\sqrt{2} - 2) * \min(dx, dy)$, for eight-way; Euclidean for navigation meshes. Using Euclidean on a four-way grid is admissible but weaker, so the search expands more nodes.
3. Variants shipped in real games: hierarchical A-star over a coarse graph then a fine one; jump point search for uniform grids; theta-star for any-angle paths; D-star Lite for replanning in changing worlds; and flow fields, where one Dijkstra pass from the goal produces a direction for every cell, which is how real-time strategy games move 500 units at once without 500 searches.
4. The standard open-source navigation toolkit is Recast and Detour by Mikko Mononen: Recast voxelizes the level and extracts walkable convex polygons; Detour does the queries, path corridors and local avoidance. Unity and Unreal both ship navigation mesh baking built on the same ideas.
5. Decision architectures with their shipped examples: hierarchical finite state machines everywhere; behaviour trees popularized by Damian Isla’s GDC 2005 talk on the Halo 2 AI; utility scoring used in The Sims from 2000; and goal-oriented action planning described in Jeff Orkin’s GDC 2006 paper “Three States and a Plan: The A.I. of F.E.A.R.”, where the state machine has only three states and a planner sequences them at run time.
6. Steering behaviours come from Craig Reynolds. His 1987 SIGGRAPH paper “Flocks, Herds, and Schools: A Distributed Behavioral Model” introduced boids with three rules: separation, alignment and cohesion. His 1999 GDC paper added seek, flee, arrive, pursue, evade, wander, obstacle avoidance and path following. Flocking that looks alive is those three rules and nothing else.
7. Now the honest part. Game AI is not optimizing for winning. It is optimizing for the player’s experience, and the two are usually opposed. A perfect aimbot enemy is trivial to write and miserable to play against.
8. Deliberate handicaps that ship in real games: enemies miss the first one or two shots on purpose; only one or two enemies attack at a time while the rest circle, a pattern used widely in third-person action games; attacks have long telegraph animations so you can react; enemies announce their intentions out loud, which is a user interface disguised as dialogue; vision cones and hearing radii are far worse than a human’s; racing games rubber-band opponents to keep the pack close.
9. The Pac-Man ghosts from 1980 remain the best teaching example. Four ghosts with four short, deterministic targeting rules read as four personalities, and none of them contains anything a modern programmer would call intelligence.
10. Separating fact from marketing: established fact is that shipped game AI is overwhelmingly hand-authored logic. Active research is reinforcement learning agents such as OpenAI Five for Dota 2 in 2018 and DeepMind’s AlphaStar for StarCraft II in 2019, which beat strong humans but are expensive to train, hard to tune and generally not fun opponents. Marketing claim is “AI-powered characters” in trailers, which in 2026 almost always means a language model generating dialogue lines, not the combat behaviour.

■ 45.8.6 WORDS – remember these

1. Finite state machine — one mode at a time — a set of states with guarded transitions, the simplest decision architecture.
2. Behaviour tree — a prioritized checklist as a tree — composites, decorators and leaf tasks ticked from the root each evaluation.
3. Utility AI — score everything and pick the best — action selection by evaluating scoring curves over world variables.
4. GOAP — describe the goal, let it plan — goal-oriented action planning, an A-star search over world states using action preconditions and effects.
5. Heuristic — an educated guess of remaining cost — the h term in A-star, admissible if it never overestimates.
6. Navigation mesh — the walkable floor as polygons — a convex polygon decomposition of traversable surfaces used as the pathfinding graph.
7. Steering — turning a path into movement — force-based local motion such as seek, arrive, separation and obstacle avoidance.

45.9 Rendering a game frame, specifically

■ 45.9.1 PLAIN – in simple words

1. In one line, the drawing pipeline is: points in space, moved into camera view, turned into triangles, filled in as pixels, shaded, depth-tested and written to a picture.
2. A game adds a lot of decisions around that, because it must do all of it in under 16.67 milliseconds.
3. First it works out what it does not need to draw at all. That is **culling**. Anything behind you or outside the camera's cone is thrown away immediately, and anything fully hidden behind a wall as well.
4. Second it groups what is left so the processor talks to the graphics chip as few times as possible. Each of those conversations is a **draw call** and they are expensive on the processor side.
5. Third it sorts. Solid things are drawn nearest first, so far-away pixels are rejected cheaply. See-through things must be drawn furthest first, so the blending is right.
6. Then it draws shadows, then the world, then the see-through things, then a stack of full-screen effects, and finally the user interface on top.

■ 45.9.2 PLAIN – a picture in your head

1. Think of a stage crew setting up a theatre scene in one minute.
2. First they check the script: what is actually visible from where the audience sits. Everything else stays in the wings. That is culling.
3. Then they carry things on in loads rather than one item at a time, because every trip costs time regardless of what is on it. That is batching.
4. They set the lights before the actors walk on, because moving lights during the scene is expensive. That is the shadow pass.
5. Glass and gauze go on last and in the right order, because you can see through them and the order changes what you see.
6. And the surtitles are projected on top of everything at the end, in their own crisp resolution. That is the user interface.

Where this comparison breaks: the stage crew works once and the scene runs. The renderer redoes the entire setup from nothing, sixty times a second, with the camera in a new place each time.

■ 45.9.3 PLAIN – a worked example

1. A real frame breakdown, taken from a profiler on a mid-range PC at 2560 by 1440 with a 60 frames per second target. The numbers are typical rather than from one specific game.

Stage	Time (ms)	Share
Shadow map passes	2.4	14 percent
Depth prepass and G-buffer	3.1	19 percent
Lighting and reflections	3.8	23 percent
Transparency and effects	1.6	10 percent
Post-processing stack	2.7	16 percent
Upscale and interface	1.1	7 percent
Present and slack	1.9	11 percent

2. Total is 16.6 ms, just inside the 16.67 ms budget.
3. Read the first row again. Shadows cost 2.4 ms, which is more than the entire user interface and post-processing of many older games, and they are just one lighting feature.
4. This is why the graphics settings menu exists. Dropping shadow resolution from 2048 to 1024 per cascade typically saves 30 to 50 percent of that row.

■ 45.9.4 PLAIN – what is really happening inside

1. **Frustum culling** tests each object's bounding sphere or box against the six planes of the camera's viewing pyramid. A sphere fully outside any one plane is rejected. This is a handful of arithmetic operations per object.
2. **Occlusion culling** removes what is hidden behind other things. It is harder, because you must know what is in front before you know what is hidden. Common approaches are precomputed visibility per region, a low-resolution software depth buffer rendered on the processor, and graphics-chip queries whose answers arrive a frame or two late.
3. **Batching** merges objects that share a material into one draw call. **Instancing** goes further: send one mesh and a list of 5,000 positions, and the chip draws all 5,000 in one call.
4. **Sorting** solid objects front to back lets the depth test discard hidden pixels before shading them. On modern hardware a depth prepass, which draws depth only and then shades with the depth test set to equal, guarantees every pixel is shaded exactly once.
5. **Forward rendering** shades each object against every light that touches it. Cost grows as objects times lights. It handles transparency and hardware anti-aliasing naturally.
6. **Deferred rendering** first writes surface properties into several full-screen buffers, together called the **G-buffer**, then computes lighting once per screen pixel. Cost is decoupled from scene complexity, which is why it won for scenes with many lights.
7. Deferred rendering's costs are memory bandwidth and transparency. See-through surfaces cannot be written into a single G-buffer pixel, so they are drawn afterwards in a separate forward pass.

■ 45.9.5 TECHNICAL – the engineer's version

1. G-buffer bandwidth arithmetic. At 1920 by 1080 with four 32-bit render targets plus a 32-bit depth buffer, that is $5 \times 4 \text{ bytes} \times 2,073,600 \text{ pixels}$, about 41.5 megabytes written per frame, or 2.5 gigabytes per second at 60 frames per second, before the lighting pass reads it all back.
2. At 3840 by 2160 the same layout is about 166 megabytes per frame and about 10 gigabytes per second of writes. This is why 4K deferred rendering is a bandwidth problem before it is a compute problem.

3. Modern engines mostly use clustered or tiled deferred and forward variants, which bin lights into screen tiles or view-space clusters so each pixel only evaluates the lights that reach it. Unreal Engine 5 and Unity's High Definition Render Pipeline both do this.
4. Draw call budgets. Under Direct3D 11 the per-call driver cost made 1,000 to 3,000 calls per frame a common ceiling. Direct3D 12, Vulkan and Metal cut that cost sharply, and with instancing, bindless resources and GPU-driven indirect draws, 10,000 to 30,000 batched calls per frame is normal in 2026.
5. Transparency ordering has no cheap correct answer. Per-object sorting fails for intersecting or concave transparent geometry. The options are alpha testing, which is opaque and sortable; per-triangle sorting, which is expensive; depth peeling, which costs one pass per layer; or weighted blended order-independent transparency, published by McGuire and Bavoil in 2013, which is approximate but cheap and widely shipped.
6. Shadows in practice are cascaded shadow maps for the sun: the view frustum is split into 3 or 4 depth ranges, each rendered from the light into a 1024 to 2048 square depth texture. Each cascade is another full scene traversal of shadow casters. Point lights need six faces. Filtering ranges from simple percentage-closer filtering to variance and moment methods.
7. A typical post-processing stack, in execution order: depth of field, motion blur, bloom, auto-exposure, tone mapping (commonly an ACES curve), colour grading through a 3D lookup texture, temporal anti-aliasing, upscaling, sharpening, then film grain and vignette.
8. Upscalers such as DLSS, FSR and XeSS are inserted after the main scene and before the user interface, so the interface renders at native output resolution and stays sharp. Getting that order wrong makes text blurry, and it is a common bug in early builds.
9. The interface is drawn last, usually as batched textured quads from a texture atlas with a signed distance field font, in a separate orthographic pass, sometimes at a different resolution from the scene.
10. Capture and profiling tools: RenderDoc for a full frame capture on almost any API; PIX on Windows for Direct3D 12 timing; NVIDIA Nsight Graphics and Radeon GPU Profiler for hardware counters; Xcode's Metal debugger on Apple platforms; Unity's Frame Debugger; Unreal's stat gpu, stat unit and ProfileGPU console commands.

■ 45.9.6 WORDS - remember these

1. Frustum culling — skip what is off-screen — rejecting bounding volumes against the six camera planes.
2. Occlusion culling — skip what is hidden — rejecting objects proven to be behind other opaque geometry.
3. Draw call — one instruction to draw something — a submitted command with a bound pipeline state, costing processor time regardless of triangle count.
4. Instancing — one command, many copies — drawing N copies of a mesh from one call with per-instance data.
5. G-buffer — the sheet of surface facts — the set of render targets holding albedo, normal, roughness, metalness and depth in deferred rendering.
6. Deferred rendering — light once per pixel, not per object — decoupling shading from geometry by writing surface data first.
7. Post-processing — the full-screen effects at the end — the ordered chain from depth of field through tone mapping to anti-aliasing and upscaling.

45.10 Audio in games

■ 45.10.1 PLAIN – in simple words

1. Sound is a pressure wave in air, and a computer stores it as a long list of numbers, typically 48,000 numbers per second per channel.
2. A game does not play files. It plays **events**: “footstep on gravel”, “pistol fire”, “door creak”. One event may pick one of eight recordings at random and change its pitch slightly so it does not sound repetitive.
3. Those recordings are packed into **sound banks**, loaded and unloaded in groups, usually one bank per level or per character.
4. A sound in the world has a position. The mixer works out how loud it should be from the distance, and how much goes to each speaker from the angle.
5. If a wall is in the way, the game fires a ray from the source to your ears, and if it is blocked, it turns the volume down and removes the high frequencies, because that is what walls do to sound.
6. All the sounds are grouped into **buses**: music, effects, dialogue, interface. Each bus has its own volume, and rules such as “duck the music by 6 decibels while anyone is speaking”.
7. Music in a modern game is not one track. It is layers that fade in as danger rises, and blocks that swap at musical boundaries.

■ 45.10.2 PLAIN – a picture in your head

1. Imagine a live sound engineer at a theatre with a mixing desk.
2. Every microphone and every playback source is a channel on the desk. Channels are grouped into buses: cast, orchestra, effects.
3. She rides the faders as the scene changes, pulls the music down when somebody speaks, and adds reverb when the scene moves to a cathedral.
4. She also has a limit: the desk has 64 channels. If a 65th sound is needed, something quiet must be dropped.
5. A game audio engine is that desk, run by rules instead of a person, updated sixty times a second.

Where this comparison breaks: the theatre engineer knows the script. The game engine does not know what will happen next, so every rule must be written to handle any order of events, including 200 sounds at once.

■ 45.10.3 PLAIN – a worked example

1. Audio memory arithmetic, which is what decides whether a sound is kept decompressed in memory or streamed from disk.
2. Uncompressed 48 kHz, 16-bit stereo is $48,000 \times 2 \times 2 = 192,000$ bytes per second. One minute is 11.5 megabytes. One hour is 691 megabytes.
3. The same audio in Vorbis at 128 kilobits per second is 16,000 bytes per second: 0.96 megabytes per minute, 57.6 megabytes per hour, a saving of about 12 to 1.
4. So the rule of thumb is: short and frequent sounds are decompressed into memory once and played from there; long music and dialogue are streamed from storage and decoded on the fly.

Content	Typical handling	Cost
Footsteps, gunfire	Decompressed in RAM	~50 KB each
Ambience loops	Compressed in RAM	~1 MB each
Music tracks	Streamed from disk	2 buffers

Content	Typical handling	Cost
Voice lines	Streamed, bank per scene	2 buffers

5. A console game commonly budgets 100 to 500 megabytes of memory for audio, which is a real constraint against a script with 40,000 recorded lines.

■ 45.10.4 PLAIN – what is really happening inside

1. The audio system does not run on the frame clock. It runs on its own high-priority thread with its own deadline.
2. That deadline is the buffer size. A 512-sample buffer at 48 kHz must be filled every $512 / 48,000$ seconds, which is 10.67 milliseconds. A 256-sample buffer is 5.33 milliseconds.
3. If the audio thread misses that deadline, the hardware plays whatever was in the buffer, and you hear a click or a stutter. There is no equivalent of a dropped frame that merely looks bad.
4. This is why audio must never wait for the game. The game posts events into a queue, and the audio thread reads them when it wakes.
5. Each buffer fill is: gather the active voices, decode any compressed ones, apply per-voice volume, pitch and filters, sum them into buses, apply bus effects such as reverb and compression, and write the result out.
6. Voice limits exist because that work is per voice. A budget of 32 to 128 simultaneous voices is normal, and beyond it the quietest and most distant sounds are stolen.

■ 45.10.5 TECHNICAL – the engineer’s version

1. Distance attenuation. The physical law is inverse square, which is minus 6 decibels per doubling of distance. Games rarely use it directly; they use authored curves with a minimum distance, inside which volume is constant, and a maximum distance, beyond which the sound is culled entirely.
2. Occlusion versus obstruction: **occlusion** is a full barrier between source and listener and attenuates both the direct and reverberant paths; **obstruction** blocks the direct path only while sound still arrives via reflections. Both are usually implemented as raycasts driving a low-pass filter cutoff and a gain.
3. Spatialization ranges from cheap stereo panning, through vector-based amplitude panning for surround layouts, to head-related transfer function convolution for binaural headphone output. Shipping implementations include Sony’s Tempest 3D AudioTech on PlayStation 5, Dolby Atmos, Windows Sonic and Valve’s Steam Audio.
4. Adaptive music has two standard techniques. **Vertical layering** plays several stems at once and cross-fades them with game intensity. **Horizontal re-sequencing** switches between composed blocks at musical boundaries, using short transition segments so the change lands on a beat.
5. Middleware exists because building all of this well takes years. The two dominant products are FMOD Studio from Firelight Technologies and Wwise from Audiokinetic, which Sony acquired in 2019. Both give composers and sound designers an editor that produces data the game loads, so audio changes do not require a programmer or a rebuild.
6. Wwise pricing per title, from the published rate card, as an example of how middleware is sold:

Tier	Budget cap	First platform
Indie	250k USD	Free
Pro	2M USD	7,000 USD
Premium	none	22,000 USD

Tier	Budget cap	First platform
Platinum	none	40,000 USD

- Additional platforms cost extra on the paid tiers, which is why a small team shipping on five platforms must budget for audio middleware as a real line item rather than an afterthought.
- Streaming considerations: each streamed voice needs at least two buffers so one can be filled while the other plays, plus a seek-and-decode budget. On consoles with a hardware decompressor, audio streaming competes with texture streaming for the same input-output bandwidth, and audio must win, because a late texture is a blurry wall and a late sound is a click.
- Diagnostic tools: the FMOD Studio and Wwise live profilers connect to a running game and show every active voice, bus level and memory pool; Unreal's `stat audio` and `au.Debug` commands; Unity's Audio Profiler module; and `pw-top` or `pactl` on Linux for the system side.

■ 45.10.6 WORDS – remember these

- Sound bank — a group of sounds loaded together — a packaged set of audio assets and event metadata with a shared lifetime.
- Voice — one sound currently playing — an active mixer channel consuming decode and processing budget.
- Attenuation — getting quieter with distance — a gain curve over distance, physically minus 6 dB per doubling.
- Occlusion — a wall in the way — attenuation and low-pass filtering applied when the direct path to the listener is blocked.
- Bus — a group fader — a mix group through which many voices are summed before further processing.
- Middleware — audio tools bought in — FMOD or Wwise, giving designers an authoring environment decoupled from engine code.

45.11 Multiplayer and netcode

■ 45.11.1 PLAIN – in simple words

- Here is the problem, and it cannot be solved, only hidden.
- Information takes time to travel. Light in a glass fibre goes about 200,000 kilometres a second, which is fast but not instant.
- Mumbai to Singapore and back is roughly 60 milliseconds of round trip in practice. Mumbai to Frankfurt is roughly 120. Mumbai to the United States east coast is roughly 240.
- So when you press fire, the server learns about it at best 30 milliseconds later, and you learn its answer 30 milliseconds after that.
- Meanwhile your screen must keep drawing at 60 frames a second. It cannot wait for the answer.
- So the game lies to you, carefully. It shows you what it believes will happen, and corrects itself later when the truth arrives.
- Everything in netcode is a variation on that one trick, plus the question of who is allowed to decide the truth.
- The answer to that question is: a server that nobody plays on, owned by the game company. That is an **authoritative server**, and it won because any other arrangement lets a player's own machine cheat.

■ 45.11.2 PLAIN – a picture in your head

1. Imagine two people playing chess by post, with letters taking three days.
2. If each waits for the other's letter before thinking, a game takes a year.
3. So each of them keeps a board at home and moves their own piece immediately, guessing what the opponent will do.
4. When the letter arrives, they compare. If the guess was right, nothing changes. If it was wrong, they must undo their guessed moves and redo them from the true position.
5. That undo-and-redo is exactly what your game does, sixty times a second, with the postal delay measured in milliseconds instead of days.
6. And there is a referee whose board is the only one that counts. If your board disagrees with the referee's, yours is wrong by definition.

Where this comparison breaks: chess players can wait, and a game cannot. Also, in chess both boards are complete. In a game the referee deliberately does not tell you about pieces you cannot see, because if it did, you could cheat.

■ 45.11.3 PLAIN – a worked example

1. The full sequence for one shot, with real numbers. Round trip time is 80 milliseconds, so one-way delay is 40 milliseconds. The server runs at 64 ticks per second, so a tick is 15.6 milliseconds. Your interpolation delay is 100 milliseconds.

```
t=0    you press fire; client predicts the shot at once
t=0    input #4471 stamped and sent over UDP
t=40   server receives input #4471
t=40   server rewinds all hitboxes by 40 + 100 = 140 ms
t=40   server tests the ray against the rewind world
t=47   next server tick packages the result
t=87   you receive "hit confirmed, input #4471 processed"
t=87   client reconciles: state matches, nothing visible
```

2. Now the same event from the victim's point of view. On their screen they ran behind a wall at their local time $t=-20$ milliseconds, that is, 20 milliseconds before you pressed fire.
3. The server still counted the hit, because it rewound the world to where you saw them, and there they were in the open.
4. How far can that be? At a running speed of 6 metres per second, 140 milliseconds is 0.84 metres. That is most of a body width, which is exactly why it feels like being shot through a wall.
5. The 140 milliseconds is not padding. It is 40 milliseconds of network delay plus the 100 milliseconds of deliberate interpolation delay your client uses to smooth other players' movement.

■ 45.11.4 PLAIN – what is really happening inside

1. **Client-side prediction.** Your client never sends "I am at position X". It sends "I held forward and pressed fire, and this is input number 4471". Then it immediately applies that input locally so the screen responds now.
2. It keeps every unacknowledged input in a list.
3. **Server reconciliation.** The server replies with "here is your true state after I processed input 4471". Your client snaps to that state and then replays inputs 4472 onwards on top of it.
4. If the prediction was correct, the replay lands exactly where you already were and you see nothing. If it was wrong, you see a small correction.

5. **Entity interpolation.** Other players are not predicted. Instead your client deliberately renders them slightly in the past, between the last two snapshots it received, so their movement is smooth even though the snapshots arrive unevenly.
6. **Lag compensation.** The server records where every hitbox was for the last half second or so. When your shot arrives, it rewinds to what you were actually looking at, tests the hit there, then restores the present.
7. That is the entire architecture. Prediction hides your own latency. Interpolation hides everyone else's jitter. Lag compensation makes aiming fair. The cost is paid by the person being shot.

■ 45.11.5 TECHNICAL – the engineer's version

1. The techniques above were set out publicly in Yahn Bernier's 2001 paper for Valve, "Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization", building on the prediction work in QuakeWorld from 1996. Almost every shooter since uses this model.
2. Blizzard's Overwatch, described in Timothy Ford's GDC 2017 talk, runs 16 millisecond command frames, about 62.5 per second, tightened to about 7 milliseconds in tournament configuration. The client clock runs ahead of the server by roughly half the round trip plus one buffered command frame, so at 160 milliseconds of latency it predicts about 96 milliseconds ahead. Above about 220 milliseconds round trip it stops predicting hits.
3. **Tick rate** is how often the server simulates. 64 ticks is one every 15.625 milliseconds; 128 ticks is one every 7.8125 milliseconds. Halving the tick interval halves the average quantization error on event timing, from about 7.8 to about 3.9 milliseconds, and doubles both server processor cost and downstream bandwidth. That is a money decision, not a technical one, which is why third-party services ran 128-tick Counter-Strike servers while official matchmaking ran 64.
4. Counter-Strike 2, released in September 2023, introduced **sub-tick** input: the client timestamps each input with its exact moment inside the tick, and the server uses that timestamp rather than snapping to the tick boundary. Valorant instead runs 128-tick servers. Whether sub-tick fully matches a true 128-tick server is genuinely disputed by players and analysts, and there is no neutral published measurement that settles it.
5. **Deterministic lockstep** is the other architecture. Only player commands are transmitted; every machine runs an identical simulation. Bandwidth is independent of the number of units, which is why real-time strategy games use it. The canonical write-up is Mark Terrano and Paul Bettner's GDC 2001 paper "1500 Archers on a 28.8", about Age of Empires.
6. Lockstep requires bit-identical simulation across every machine, for the reasons given in section 45.7: unspecified transcendental functions, compiler reordering, fused multiply-add, and non-deterministic thread ordering will all desynchronize it. It also needs a command delay of two to five turns, typically 100 to 250 milliseconds, so that all commands arrive before the turn executes. That delay is why strategy games feel slightly sluggish by design.
7. Rollback netcode for fighting games, popularized by Tony Cannon's GGPO from 2006, is lockstep plus speculation: assume the opponent pressed nothing, and when the real input arrives, restore the saved state and re-simulate the intervening frames. It needs a fast full state save and restore every frame, which constrains how the game is written.
8. **Bandwidth.** A worked example: 64 players, 20 snapshots per second, 200 bytes per snapshot after delta compression and relevancy culling, gives $64 \times 20 \times 200 = 256,000$ bytes per second, about 2.05 megabits per second outbound per server, before packet headers.

Genre	Snapshot rate	Down per client
Competitive shooter	60 to 128 Hz	200 to 800 kbps
Battle royale, 100 players	20 to 30 Hz	150 to 500 kbps
MMO	5 to 20 Hz	30 to 150 kbps
Strategy, lockstep	5 to 10 Hz	5 to 30 kbps

9. **Delta compression** sends only what changed since the last snapshot the client acknowledged, a model established by Quake 3 in 1999. Combined with quantization it is dramatic: a position stored as three 32-bit floats is 12 bytes; quantized to 16-bit fixed point over a bounded world it is 6 bytes; as a delta from the previous value with variable-length bit packing it is often 1 to 3 bytes.
10. **Relevancy culling**, sending a client only what it could plausibly perceive, saves bandwidth and is also the single most effective anti-cheat measure that exists, because a wallhack cannot draw data it never received. Riot's Fog of War in Valorant, shipped in 2020, is the best-known example.
11. **Why UDP**. TCP, standardized in RFC 793 in 1981 and now RFC 9293 from 2022, guarantees an ordered reliable byte stream. If packet 5 is lost, packets 6, 7 and 8 sit in the kernel buffer until 5 is retransmitted. That is head-of-line blocking, and it costs at least one round trip. For position updates it is exactly wrong: by the time packet 5 arrives it is stale and you already hold newer truth.
12. UDP, RFC 768 from 1980, has no ordering, no retransmission and no congestion control. Games build their own reliability layer and choose per message: unreliable for positions, since the next update supersedes this one; reliable and ordered for chat and inventory; reliable but unordered for one-off events. QUIC, RFC 9000 from 2021, offers independent streams over UDP and is used for some game backend traffic, but action games generally still roll their own.
13. Practical UDP detail: since there is no connection, NAT devices time out idle mappings, commonly after 30 to 120 seconds, so games send small keep-alive packets.
14. **Matchmaking** is a multi-objective search over skill, latency, party size, role and queue time, widening its tolerances as the wait grows. Skill systems descend from Arpad Elo's chess rating from the 1960s, through Glicko, to Microsoft's TrueSkill from 2007, which models skill as a distribution with an uncertainty that shrinks as it observes you.
15. **NAT traversal** matters because most home connections, and every carrier-grade NAT connection, have no reachable public address. STUN, currently RFC 8489 from February 2020, which obsoletes RFC 5389 of 2008 and RFC 3489 of 2003, lets a host discover the public address and port its NAT assigned. TURN, currently RFC 8656 from February 2020, relays traffic when a direct path cannot be established. ICE, RFC 8445 from 2018, gathers all candidate addresses and systematically tests pairs.
16. Hole punching works when both sides send outbound packets simultaneously so each NAT creates a mapping the other can use. It fails when both ends use symmetric NAT, which allocates a different external port per destination. A relay is then the only option, and a single-digit to low-double-digit percentage of connection attempts need one in practice.
17. **Relays** such as Steam Datagram Relay carry game traffic over a provider's private backbone. The benefits are that the server's real address is never exposed, which prevents denial-of-service attacks on individual players; routing is often better than the public internet; and NAT problems disappear.
18. The unfixable part: a player far from any datacentre cannot be given a competitive experience. No amount of netcode removes 240 milliseconds. This is a real and often-ignored fairness issue for players in regions without local server presence.

■ 45.11.6 WORDS – remember these

1. Authoritative server — the referee whose word counts — a server that simulates canonically and treats client messages as inputs, never as state.
2. Client-side prediction — acting before permission — applying local input immediately and retaining it for later replay.
3. Reconciliation — correcting the guess — snapping to the server state and replaying unacknowledged inputs on top of it.
4. Entity interpolation — drawing others slightly in the past — rendering remote entities between two received snapshots with a fixed delay buffer.
5. Lag compensation — rewinding to what the shooter saw — server-side historical hitbox rewind by the shooter's latency plus interpolation delay.
6. Tick rate — how often the server thinks — simulation steps per second, 64 ticks being 15.625 ms and 128 ticks 7.8125 ms.
7. Delta compression — send only what changed — encoding a snapshot as a difference from the last one the client acknowledged.
8. Deterministic lockstep — send commands, not state — identical simulation on every machine, requiring bit-exact reproducibility.

45.12 Cheating and anti-cheat

■ 45.12.1 PLAIN – in simple words

1. Cheating in an online game means making your copy of the program do something the designers did not intend, to gain an advantage.
2. An **aimbot** reads the game's memory to find where other players are, then points your weapon at them for you.
3. A **wallhack** draws other players through walls. It works only because your machine was told where they are.
4. A **speed hack** changes the clock the game reads, so your character moves faster than everyone else's.
5. **Packet manipulation** means interfering with the messages themselves: a lag switch drops your outgoing packets so others cannot hit you.
6. Some cheating is not technical at all: paying someone to play your account, deliberately losing to face weaker opponents, or trading wins.
7. The one rule that explains all defences is this: your machine belongs to the attacker. Anything the game checks on your machine, the attacker can remove.
8. So the only real defence is to check on the server, and to never send your machine information it is not entitled to.

■ 45.12.2 PLAIN – a picture in your head

1. Think of an examination hall.
2. Asking candidates to mark their own papers is the client trusting itself. Some will cheat, and you will never know.
3. Collecting the papers and marking them centrally is server-side validation. Now the cheating must happen in the room, not in the marking.
4. Not printing the answers on the back of the paper is relevancy culling. You cannot copy what you were never given.
5. Invigilators watching for suspicious behaviour are statistical detection.

6. And searching candidates at the door, going through their pockets, is kernel-level anti-cheat. It works, and people reasonably object to it.

Where this comparison breaks: an exam lasts three hours. Anti-cheat software that runs at boot is searching your pockets every day whether or not there is an exam, which is precisely the objection people raise.

■ 45.12.3 PLAIN – a worked example

1. How server-side validation actually looks, for movement.
2. The client sends: “input 4471, held forward, delta time 15.6 milliseconds”.
3. The server does not read a position. It runs the same movement code the client ran, from the state it already holds.
4. Maximum run speed is 5.2 metres per second. In 15.6 milliseconds the maximum possible movement is $5.2 \times 0.0156 = 0.081$ metres.
5. If the client claims a delta time of 100 milliseconds while only 15.6 have passed on the server’s clock, that is a speed hack, and it is rejected by comparing the client’s claimed time budget against elapsed server time.
6. The same pattern applies everywhere: fire rate against the weapon’s cooldown, damage against the weapon’s table, line of sight against the server’s own geometry, and inventory against the server’s own database.

Cheat	Defence that works
Aimbot	Statistical aim analysis
Wallhack	Do not send hidden positions
Speed hack	Server-side time budget
Damage edit	Server owns the damage table
Item duplication	Atomic server transactions

■ 45.12.4 PLAIN – what is really happening inside

1. Statistical detection looks at how you aim rather than at your files.
2. Human aim has tremor, overshoot, correction and a reaction time that never drops below about 150 milliseconds for a genuinely unexpected event.
3. A machine has none of those. Its angular snap to a target is too fast, too precise, and too consistent across distances.
4. Detectors measure the distribution of flick times, the time between a target becoming visible and the first shot, the hit rate as a function of range, and the smoothness of the mouse path.
5. The output is a probability, not a verdict, which is why bans are issued in waves rather than instantly: batching hides which specific detection fired, so cheat authors cannot binary-search their way around it.
6. False positives are the hard part. At a 0.1 percent false positive rate across 10 million players, 10,000 innocent people are banned. This is why detections are usually required to be independently corroborated.

■ 45.12.5 TECHNICAL – the engineer’s version

1. Categories in more detail: memory-reading external cheats; injected internal cheats hooking the rendering or input path; triggerbots; radar overlays; automated recoil control macros in hardware mice; lag switches and packet delay; and account services such as boosting and win trading.

2. Two categories defeat all software-only defences. **DMA cheats** use a second computer with a PCIe direct memory access card reading the gaming machine's memory over the bus, so nothing runs on the machine being watched. **Screen-reading cheats** capture the video output with a capture card, run object detection on a second machine, and drive a hardware device that emulates a mouse. Both are a live problem in 2026 and neither is detectable by inspecting the game machine's software.
3. Kernel-level anti-cheat runs a driver at ring 0 so it can see and block things user-mode code cannot: other drivers loading, handles opened to the game process, hooked system calls, and tampered kernel structures. Shipped examples include Riot Vanguard, introduced with Valorant in 2020 and extended to League of Legends in 2024; Easy Anti-Cheat, now owned by Epic Games; BattlEye; and Activision's Ricochet, introduced in 2021.
4. The honest debate, both sides in full. For: cheats themselves run in the kernel, and a user-mode watcher simply cannot see a kernel-mode cheat, so without a kernel presence the defender has already lost the privilege race. Games with kernel anti-cheat measurably have fewer visible cheaters.
5. Against: it is a signed driver with total access to your machine, shipped by a game company, usually loading at boot whether or not you play. A bug in it is a system-wide vulnerability. This is not hypothetical: the mhyprot2.sys driver shipped with Genshin Impact was abused by ransomware operators to disable antivirus software, and the July 2024 CrowdStrike incident showed what a single faulty kernel-mode update does at scale.
6. It also excludes people. Kernel anti-cheat generally does not work under Proton on Linux or the Steam Deck unless the developer explicitly enables the vendor's Linux support, which many competitive titles decline to do.
7. The platform direction is changing. Microsoft announced the Windows Resiliency Initiative in November 2024 and from 2025 began moving endpoint security products out of kernel mode into a supported user-mode platform, with security vendors involved in the design. As of August 2026 this is a live transition rather than a completed one, and how anti-cheat fits into it is not yet settled.
8. Hardware-backed identity raises the cost of ban evasion. Valorant requires TPM 2.0 and Secure Boot on Windows 11, which lets bans attach to something harder to change than an account or a network address.
9. Machine learning detection is real and shipped: Valve described VACnet, a classifier trained on human-reviewed cases, at GDC in 2018, and comparable systems exist at other publishers. It is a filter that raises human review efficiency, not an oracle.
10. The arms race is structural and permanent. Cheat development is a paid industry with subscription pricing, private builds to avoid signature detection, and customer support. Anti-cheat cannot win; it can only make cheating expensive, unreliable and short-lived enough that most people do not bother.

■ 45.12.6 WORDS – remember these

1. Aimbot — the machine aims for you — automated target acquisition driven by memory-read entity positions or screen-space detection.
2. Wallhack — seeing through walls — rendering entity positions the client received but should not have been told about.
3. Server-side validation — the server checks everything — simulating client inputs authoritatively and rejecting physically impossible claims.
4. Relevancy culling — do not send what cannot be seen — filtering network state by visibility, which prevents wallhacks at the source.
5. Kernel-level anti-cheat — a watcher with full system access — a ring 0 driver monitoring for tampering, with the privilege and stability trade-off that implies.
6. Ban wave — punishing in batches — delayed enforcement that obscures which detection triggered, slowing the cheat authors' feedback loop.

45.13 What actually happens when you double-click a game

■ 45.13.1 PLAIN – in simple words

1. The short version of the ordinary part: the operating system reads the file's header, works out how much memory the program needs, copies the code and data into that memory, fills in the addresses of the system libraries the program uses, and jumps to its first instruction.
2. That is true for every program. A game then does a great deal more before you see anything.
3. It usually starts a **launcher** first, which checks for updates, verifies that your files have not been altered, and logs you in.
4. It starts the anti-cheat service, if there is one.
5. Then it starts the real game program, which must find a graphics card, ask the driver for a device, and set up the chain of pictures it will draw into.
6. Then it must turn its shader source into machine code for your specific graphics chip. On the first run this can take minutes, and if it is done lazily while you play, you get the famous first-run stutter.
7. Then it loads a small amount of content for the menu, and only then does the loop from section 45.3 start running.
8. Choosing a level starts a second, much bigger load, and finally you play.

■ 45.13.2 PLAIN – a picture in your head

1. Think of an aircraft before a flight.
2. Opening the door is the double-click. Nothing is running yet.
3. Ground power, then auxiliary power, then engines: that is the loader, the launcher and the engine start.
4. The pre-flight checklist is the device and swap chain creation. Every item must be confirmed before the next.
5. Loading passengers and fuel is asset loading. It is the slowest part and it scales with how far you are going.
6. Taxiing is the main menu: you are moving, everything is running, but the real work has not started.
7. Take-off is entering the level.

Where this comparison breaks: an aircraft does the same checklist every time. A game's first run is dramatically different from its second, because the shader cache, the file system cache and the driver cache are all empty the first time.

■ 45.13.3 PLAIN – a worked example

1. The twenty-five step trace, with realistic timings for a Windows PC with an NVMe solid-state drive, a mid-range graphics card and a broadband connection. Cold means first ever run; warm means a later run.

Step	What happens	Warm time
1	Shell creates the process	20 ms
2	Loader maps the launcher	30 ms
3	Launcher checks for updates	200 to 2000 ms
4	File integrity spot check	500 ms to 30 s
5	Anti-cheat service starts	50 to 500 ms
6	Launcher spawns the game	50 ms
7	Loader maps the game image	100 to 400 ms
8	Runtime and static init	20 to 100 ms
9	Read config and command line	5 to 20 ms
10	Memory pools reserved	10 to 50 ms
11	Job system threads start	5 to 20 ms
12	Archives mounted, index read	50 to 300 ms
13	Enumerate adapters, device	100 to 500 ms
14	Create swap chain buffers	50 to 200 ms
15	Allocate render targets	50 to 200 ms
16	Load or compile shader states	2 s warm, minutes cold
17	Audio device and mixer graph	50 to 200 ms
18	Enumerate input devices	10 to 100 ms
19	Sockets, DNS, account login	200 to 1500 ms
20	Load front-end bundle	500 to 3000 ms
21	Main menu, loop now running	first frame
22	You pick a level, load list	10 ms
23	Stream, decompress, upload	3 to 30 s
24	Warm-up frames force work	1 to 5 s
25	Gameplay loop settles	steady state

2. Warm total to the main menu is roughly 4 to 8 seconds. Cold total can be several minutes, almost all of it in step 16.
3. Step 16 is the one worth understanding. Shader source is compiled at build time into an intermediate form, but the final translation into your exact graphics chip's instructions can only happen on your machine, because the developer does not know which chip you have.
4. A game may have 20,000 to 200,000 pipeline states. If each takes 20 milliseconds to build, 50,000 of them is about 17 minutes of one core, or roughly one minute spread across sixteen cores.
5. If the game builds them lazily instead, the first time a new effect appears the frame stops for 20 to 200 milliseconds. That is the stutter. Consoles never have it, because the chip is known and the compilation happened at build time.

■ 45.13.4 PLAIN – what is really happening inside

1. A **save file** is a written-down copy of the parts of the world the designer decided matter.
2. It is never the whole world. Nobody stores every leaf position. It stores your position, your inventory, quest flags, which doors are open, which enemies are dead, and a version number.
3. Turning live objects into a flat sequence of bytes is called **serialization**, and reading it back is **deserialization**.

4. The formats used, in rough order of how common they are:

Format	Example use	Trade-off
Chunked binary	Most console games	Fast, opaque
Tagged binary	Minecraft NBT	Flexible, compact
JSON, often gzipped	Many indie games	Readable, large
SQLite database	Mobile, big worlds	Robust, heavier
Engine key-value	Unity PlayerPrefs	Settings only

5. Minecraft's NBT, standing for Named Binary Tag, is a tagged tree of typed values, gzip-compressed, with world data grouped into region files that each hold a 32 by 32 block of chunk columns.
6. The correct way to write a save is: write to a temporary file, flush it to the storage device, then rename it over the old one. Rename is atomic on every mainstream file system, so a power cut leaves either the old save or the new one, never half of each. Games that write in place corrupt saves.
7. Saves must carry a version number, because patch 1.20 must still load a save written by 1.00, which means keeping an upgrade path for every old version. This is why some games break saves at a major update and why players are right to be annoyed about it.
8. Online games sign or checksum saves and usually store them server-side, so that editing them is pointless. Single-player games often do not, which is why save editors exist and are harmless.

■ 45.13.5 TECHNICAL - the engineer's version

1. On Windows the image is a PE file: the loader maps sections according to the section table, applies base relocations if the preferred base is taken, resolves the import address table, runs TLS callbacks, then calls the entry point. On Linux it is an ELF file handled by `ld.so`. Section 45.13 does not repeat that machinery; it only notes where it sits in the sequence.
2. Graphics initialization in Direct3D 12 is: create a DXGI factory, enumerate adapters, `D3D12CreateDevice`, create command queues, then `CreateSwapChainForHwnd` with 2 or 3 buffers and a chosen swap effect, usually flip-model. In Vulkan it is `vkCreateInstance`, physical device selection, `vkCreateDevice`, then a surface and a `VkSwapchainKHR`.
3. Pipeline state objects bundle shaders, blend, depth, rasterizer and vertex layout into one immutable object, which is why they are compiled up front rather than assembled per draw. Both APIs support a pipeline cache blob that can be saved to disk and reloaded, which is what makes the warm run fast. Steam additionally distributes precompiled Vulkan pipeline caches for Proton titles so a Steam Deck rarely compiles at play time.
4. Asset streaming on modern consoles uses hardware decompression and a direct storage path: the PlayStation 5 Kraken decompressor is rated at about 5.5 gigabytes per second of compressed input, and Microsoft's DirectStorage on Windows, with GPU decompression from version 1.1 in 2022, brings a similar path to PC.
5. Useful observation commands: Process Monitor to watch every file the game opens in order; `strace -f` or `ltrace` on Linux; Windows Performance Recorder or `xperf` for a boot-to-first-frame trace; `nvidia-smi` or GPU-Z to watch video memory fill during load; and the engine's own load-time trace, such as Unreal Insights with the LoadTime channel.

■ 45.13.6 WORDS – remember these

1. Launcher — the small program that runs first — an updater, integrity checker and authentication front end that spawns the real executable.
2. Swap chain — the queue of pictures — the set of back buffers created at startup and rotated between renderer and display.
3. Pipeline state object — a frozen graphics configuration — an immutable bundle of shaders and fixed-function state compiled ahead of a draw.
4. Shader cache — the compiled results kept for next time — an on-disk store of driver-compiled pipeline blobs keyed by hardware and driver version.
5. Serialization — turning live objects into bytes — writing state in a versioned, replayable form, and reading it back.
6. Atomic rename — the safe way to save — writing to a temporary file and renaming over the target so no partial file is ever visible.

45.14 Consoles versus PC

■ 45.14.1 PLAIN – in simple words

1. A PC is a different machine for every person. A console is the same machine for everybody.
2. That single difference is worth more than it sounds.
3. On a console the developer knows exactly how fast the processor is, exactly how much memory there is, exactly how fast the storage is, and exactly which graphics chip will run the shaders.
4. So the developer can tune to that machine, compile shaders in advance, use the memory to the last megabyte, and rely on a fixed loading speed.
5. On a PC the same code must work on thousands of combinations, through a driver that has to guess, with unknown amounts of memory, and with shaders compiled on your machine while you wait.
6. In exchange for that constraint, the console maker takes control: your game must pass their tests before it can be sold.
7. That is why a console game with weaker numbers on paper can look better than a PC game on stronger hardware.

■ 45.14.2 PLAIN – a picture in your head

1. Think of a tailor.
2. A bespoke suit is measured for one body. It fits perfectly and it fits nobody else. That is a console game.
3. Off-the-peg suits come in twelve sizes and must fit everybody roughly. They need more fabric in places, and they hang a little loose. That is a PC game.
4. The bespoke suit is not made of better cloth. It just wastes none.

Where this comparison breaks: a PC owner can buy far better cloth. A 1,500 dollar PC in 2026 genuinely outperforms any console, and gets modding, backwards compatibility and higher frame rates too. The console advantage is efficiency per unit of hardware, not raw capability.

■ 45.14.3 PLAIN – a worked example

1. The 2026 landscape, with United States prices as of August 2026. Note that prices in this generation went up rather than down, which is historically unusual.

Machine	Graphics	Price
PlayStation 5 (2020)	36 CU, 10.3 TFLOPS	649.99
PlayStation 5 Pro (2024)	60 CU, 16.7 TFLOPS	899.99
Xbox Series X (2020)	52 CU, 12.1 TFLOPS	649.99
Xbox Series S (2020)	20 CU, 4 TFLOPS	399.99
Nintendo Switch 2 (2025)	1,536 Ampere cores	449.99

2. PlayStation raised prices in August 2025 and again on 2 April 2026. Xbox raised prices in October 2025. The Nintendo Switch 2 launched at 449.99 on 5 June 2025 and has held that price.
3. Memory tells the same story as compute. The PlayStation 5 has 16 gigabytes of GDDR6 at 448 gigabytes per second, shared between processor and graphics chip. The Pro keeps 16 gigabytes but raises bandwidth to 576 gigabytes per second and adds 2 gigabytes of DDR5 for the system. The Switch 2 has 12 gigabytes of LPDDR5X at about 102 gigabytes per second docked.
4. Unified memory is the quiet advantage. On a PC, data must be copied across the PCI Express bus from system memory to video memory. On a console there is one pool and no copy.

■ 45.14.4 PLAIN – what is really happening inside

1. Certification is a checklist, not a review of whether the game is good.
2. Sony calls its list the Technical Requirements Checklist. Microsoft calls its list the Xbox Requirements. Nintendo's process is called Lotcheck.
3. The checks are behavioural. What happens if the controller battery dies mid-save. What happens if the network drops during a purchase. What happens if the user signs out. Does the game suspend and resume within the required time. Does it use the platform's exact button names. Does it declare its save data size correctly. Does it survive an eight-hour soak test.
4. A submission takes on the order of one to three weeks per platform per build, and a failure means fixing and resubmitting. Studios therefore lock content weeks before launch to leave room for one or two failures.
5. Those durations are studio experience rather than published figures, and they vary by platform, by title and by relationship.
6. Age ratings are a separate process again: ESRB in North America, PEGI in Europe, CERO in Japan, USK in Germany, and the IARC questionnaire for digital storefronts.

■ 45.14.5 TECHNICAL – the engineer's version

1. The concrete reasons a fixed target wins, in order of importance: precompiled shaders, so no run-time compilation and no first-run stutter; unified memory, so no bus copies; low-overhead graphics APIs closer to the hardware than the PC equivalents; a known memory budget minus a known system reservation; guaranteed storage bandwidth with hardware decompression; and a single optimization target, so every hour of tuning benefits every player.
2. The APIs differ: PlayStation uses its own GNM and GNMx; Xbox uses Direct3D 12 with console-specific extensions and a fixed driver; Nintendo Switch and Switch 2 use NVN. All three expose more control and less driver interpretation than desktop Direct3D 12 or Vulkan.
3. System reservations are real and published in developer documentation rather than in marketing: the Xbox Series X reserves about 2.5 gigabytes of its 16, and the Switch 2 reserves about 3 gigabytes of its 12 by reported figures, leaving the rest to the game.
4. Handhelds are the growth area. The Steam Deck, released 25 February 2022 with an OLED model in November 2023, uses an AMD APU with a 4-core Zen 2 processor and 8 RDNA 2 compute units,

about 1.6 teraflops, 16 gigabytes of LPDDR5, and a 1280 by 800 display, in a 4 to 15 watt power envelope.

5. SteamOS is Arch-based Linux, and Windows games do not run on Linux natively. **Proton** makes them run. It is not emulation: the processor instructions are native x86-64 and execute at full speed. Only the calls into Windows are translated.
6. Proton is Wine, the open reimplementation of the Windows API begun in 1993, plus DXVK which translates Direct3D 9, 10 and 11 into Vulkan, plus VKD3D-Proton which translates Direct3D 12 into Vulkan, plus FAudio for XAudio2, plus Steam-specific shims. Shader bytecode is translated too, from DXBC and DXIL into SPIR-V.
7. Typical overhead is 0 to 15 percent, and a minority of titles run faster than on Windows because of driver differences. The thing Proton cannot fix is kernel anti-cheat, which must be explicitly enabled for Linux by the game's developer.
8. Valve announced a **Steam Machine** in November 2025 for release in 2026: a small SteamOS box with a semi-custom 6-core AMD Zen 4 processor up to 4.8 GHz, an RDNA 3 graphics part, 16 gigabytes of system memory plus 8 gigabytes of dedicated graphics memory, running the same Proton stack. At the time of writing in August 2026 the price had not been announced.
9. The honest summary: consoles buy predictability and efficiency; PCs buy ceiling and freedom. Which is better depends entirely on whether you value a guaranteed experience or a configurable one, and reasonable people split on it.

■ 45.14.6 WORDS - remember these

1. Fixed platform — one machine for everyone — a single known hardware configuration allowing exact tuning and ahead-of-time compilation.
2. Certification — the platform holder's checklist — a technical requirements pass covering behaviour, terminology, storage and stability.
3. Unified memory — one pool shared by processor and graphics — no PCI Express copies between system and video memory.
4. System reservation — the slice the console keeps — memory and processor time reserved for the operating system and background features.
5. Compatibility layer — translating one platform's calls to another — Proton, which maps Windows and Direct3D calls to Linux and Vulkan without emulating the processor.
6. Lotcheck — Nintendo's certification — the platform's technical and content compliance review before release.

45.15 Emulation

■ 45.15.1 PLAIN – in simple words

1. An **emulator** is a program that pretends to be a different computer.
2. The game it runs was compiled for a processor your machine does not have, talking to chips your machine does not have.
3. So the emulator reads the old machine's instructions one at a time and does whatever they were supposed to do, using your machine's instructions.
4. It also pretends to be the old machine's graphics chip, sound chip, cartridge slot and controller ports.
5. There are two ways to run the instructions. **Interpretation** reads and acts on each instruction every time it is encountered. Simple, accurate, slow.
6. **Recompilation** translates a whole block of old instructions into new instructions once, keeps the translation, and reuses it. Complicated, much faster.
7. There is a third shortcut. Instead of emulating the old machine's system software, the emulator can spot a call to a known system function and do the equivalent thing natively. That is **high-level emulation**.

■ 45.15.2 PLAIN – a picture in your head

1. Interpretation is a live interpreter at a conference, translating sentence by sentence, every time, including sentences the speaker repeats.
2. Recompilation is having the speech translated once and printed, so the repeated parts are read from the page.
3. High-level emulation is noticing that the speaker just said a standard phrase, and saying the equivalent standard phrase in the target language rather than translating it word by word.
4. Accuracy is the trade in each case. The live interpreter catches every nuance. The printed version cannot adapt if the speaker goes off script.

Where this comparison breaks: a speech does not care about timing to the microsecond. Old consoles absolutely did, and many games depend on exactly when a chip responded, not just on what it returned.

■ 45.15.3 PLAIN – a worked example

1. Why emulating a slow machine can need a fast one.
2. The Super Nintendo ran at about 3.58 MHz. A modern processor runs at about 4 GHz, roughly 1,100 times faster in clock terms alone, and does far more per clock.
3. Yet a fully cycle-accurate SNES emulator such as byuu's bsnes and higan needed roughly a 3 GHz machine to reach full speed.
4. The reason is not the instructions. It is the synchronization. A cycle accurate emulator must interleave the processor, the picture processing unit, the sound processor and the cartridge coprocessors many times per emulated instruction, and each switch costs far more than the instruction itself.
5. A less accurate emulator that runs each chip in blocks and synchronizes rarely is ten to fifty times faster and breaks a small number of games that rely on exact timing.

Approach	Speed cost	Accuracy
Cycle-accurate interpreter	500 to 1500x	Highest

Approach	Speed cost	Accuracy
Block interpreter	20 to 100x	Good
Dynamic recompiler	2 to 10x	Varies
High-level API emulation	1 to 3x	Lowest

6. Those ratios are approximate and depend heavily on the console being emulated.

■ 45.15.4 PLAIN – what is really happening inside

1. An interpreter's inner loop is: read the instruction at the program counter, decode it, look up a handler, run the handler, update the registers and flags, advance the program counter, add the instruction's cycle cost to a counter, and check whether any hardware event is due.
2. That is 10 to 100 host instructions to do the work of one guest instruction.
3. A dynamic recompiler instead scans forward from the program counter until a branch, translates that whole block into host machine code once, stores it in a cache keyed by address, and jumps into it.
4. The block then runs at close to native speed on every later visit.
5. Self-modifying code and code overwritten by a memory bank switch force the cached block to be thrown away and retranslated, which is why emulating older machines with heavy bank switching is harder than it looks.
6. Emulating the graphics chip is where the biggest shortcut lives. If the emulator translates the old chip's drawing commands into modern graphics calls instead of simulating its pixel pipeline, it can also render at four or eight times the original resolution, which the original hardware could never do.

■ 45.15.5 TECHNICAL – the engineer's version

1. Terms, precisely. **Interpretation** decodes per execution. **Threaded interpretation** removes dispatch overhead by chaining handler addresses. **Dynamic binary translation**, also called dynarec or JIT, compiles per basic block or per trace. **Static recompilation** translates the whole binary ahead of time and mostly fails on general programs because indirect jumps make it impossible to tell code from data reliably, though per-title static recompilation projects have succeeded, notably for Nintendo 64 titles from around 2023.
2. **High-level emulation** replaces the guest system's own libraries with native implementations, which is essential when the firmware cannot legally be distributed. **Low-level emulation** runs the real firmware and is more accurate but requires the user to supply it.
3. Accuracy has axes: instruction accuracy, cycle accuracy, sub-instruction bus timing, sample-accurate audio, and scanline or dot accurate video. Choosing a lower point on any axis buys speed and breaks some titles.
4. The legal position, stated neutrally and without advice.
5. Writing an emulator has been upheld in United States courts. In *Sony Computer Entertainment v. Connectix*, decided by the Ninth Circuit in 2000, intermediate copying of the PlayStation BIOS for reverse engineering was held to be fair use. *Sony v. Bleem*, also 2000, held that using screenshots in comparative advertising was fair use.
6. Copying game data you do not own is copyright infringement essentially everywhere. Whether dumping a cartridge you do own is lawful varies by country, and in several jurisdictions the act of circumventing copy protection to make the dump is itself unlawful regardless of ownership, under the United States Digital Millennium Copyright Act of 1998 and the European Union Copyright Directive of 2001.

7. Firmware and BIOS images are copyrighted works in their own right; distributing them is infringement whatever the emulator's status.
8. Enforcement in practice: in March 2024 Nintendo settled with Tropic Haze, developer of the Switch emulator Yuzu, for 2.4 million United States dollars, and the project shut down. Nintendo's complaint centred on circumvention of encryption rather than on emulation as a concept.
9. Where people disagree: emulator authors and archivists argue the tool is lawful, general-purpose and necessary; rights holders argue that emulators of current hardware exist principally to enable piracy. Both claims can be true of different projects at the same time.
10. Preservation is the strongest argument, and it has numbers behind it. A study published in July 2023 by the Video Game History Foundation with the Software Preservation Network found that about 87 percent of video games released in the United States before 2010 are out of commercial circulation entirely.
11. The mechanisms of loss are concrete: digital storefronts close, as the Wii Shop Channel did in 2019 and the Nintendo 3DS and Wii U eShops did in March 2023; optical discs suffer disc rot; cartridge save batteries last ten to twenty years; and flash memory loses charge over decades.
12. Museums and libraries have argued for broader legal exemptions to preserve and provide access to games. Some exemptions exist for on-site preservation work; remote access for researchers remains contested.

■ 45.15.6 WORDS – remember these

1. Emulator — a program pretending to be another machine — software modelling a foreign instruction set and peripheral hardware.
2. Interpretation — decode every instruction every time — a fetch-decode- execute loop in software, simple and slow.
3. Dynamic recompilation — translate once, run many times — just-in-time compilation of guest basic blocks into host machine code.
4. High-level emulation — reimplement the system calls — replacing guest firmware routines with native host implementations.
5. Cycle accuracy — matching the original timing exactly — modelling per-clock behaviour including inter-chip contention.
6. ROM — a copy of the original game data — a dump of cartridge or disc contents, copyrighted material unless the rights holder permits otherwise.

45.16 Making your first game, practically

■ 45.16.1 PLAIN – in simple words

1. Build something tiny. Then build something slightly less tiny.
2. The reason is not that you cannot handle a big idea. It is that finishing is a separate skill from building, and it is the rarer one.
3. A tiny game teaches you the whole pipeline: input, a loop, collision, a win condition, a lose condition, sound, a menu, a build, and publishing.
4. A big game teaches you one slice of that and then dies.
5. The proven ladder is: Pong, then Breakout, then Snake, then Asteroids, then a one-level platformer, then something of your own.
6. Each of those should take days, not months. If it is taking months, the scope is wrong.
7. Then publish it. Publishing a bad small game teaches you more than not publishing a good big one.

■ 45.16.2 PLAIN – a picture in your head

1. Nobody learns to cook by attempting a five-course dinner for twenty.
2. You learn by making one dish, badly, then again, less badly.
3. The dinner party teaches you timing, which the single dish does not, so you do eventually need one. But you need it after ten dishes, not before.
4. A game jam is the dinner party: a fixed 48 hours, a theme announced at the start, and something submitted at the end regardless of quality.

Where this comparison breaks: a bad dish is eaten and forgotten. A published game stays on the internet under your name, which is exactly why some people never publish. Publish anyway. Nobody is looking as hard as you fear.

■ 45.16.3 PLAIN – a worked example

1. A thirty-day plan. It assumes about two hours a day, which is realistic alongside study or a job.

Days	Task	Deliverable
1 to 2	Engine, one tutorial	A window, a moving box
3 to 4	Core verb only	Move and jump feel good
5 to 7	One obstacle, win, lose	A 30-second playable loop
8 to 9	Playtest with 3 people	Written notes, a cut list
10 to 13	Fix the top three problems	The loop reads clearly
14 to 16	Add exactly one system	The game has some depth
17 to 18	Second playtest	Notes, cut list again
19 to 21	Sound and feedback pass	It feels good to touch
22 to 24	Five levels or a curve	Content, not features
25 to 26	Menu, pause, settings	Ship-shaped
27 to 28	Build, test on a clean PC	An installable build
29	Store page, screenshots	Page ready
30	Publish on itch.io	Released

2. Notice that only days 3 to 7 and 14 to 16 add features. Everything else is testing, fixing, content and shipping. That ratio is correct and it surprises everyone the first time.

■ 45.16.4 PLAIN – what is really happening inside

1. **Scope** is the number one killer of hobby projects, and the reason is arithmetic, not willpower.
2. Every feature costs its own build time plus its interaction with every other feature. The number of interacting pairs grows roughly as the square of the feature count.
3. Five features have ten pairs to make work together. Fifteen features have 105. That is why the second half of a project feels ten times harder than the first.
4. So the discipline is a **cut list**: when you have a new idea, write it down on a “version two” list, and carry on with what you were doing.
5. Rough multipliers for common additions, approximate but directionally right: multiplayer multiplies total work by about three; a save and load system by about 1.3; procedural generation by about two; a narrative mode by more than any of them.
6. The classic failure is choosing a massive online role-playing game or an open world as a first project. It has ended more hobby projects than any technical difficulty.
7. Prototype with untextured boxes. If a game is not fun as grey boxes, art will not save it. Art makes a good game feel better; it cannot make a boring game interesting.

8. Playtest by handing someone the controls and saying nothing. Every time you want to explain something, that is a design bug. Write it down instead of speaking.

■ 45.16.5 TECHNICAL – the engineer’s version

1. Choosing a tool by goal, as of August 2026:

Goal	Tool	Why
Understand the machinery	raylib or SDL3	No magic, all yours
Ship a 2D game quickly	Godot 4	Free, small, 2D-first
Reach mobile and consoles	Unity 6	Widest platform support
Highest-end 3D visuals	Unreal Engine 5	Best default look
Web game in a browser	Godot or a JS engine	HTML5 export

2. Code-first libraries worth knowing: SDL3 and raylib in C, LOVE in Lua, pygame in Python, MonoGame in C#, Bevy in Rust, and three.js in JavaScript. These give you a window, input, drawing and sound, and nothing else, which is exactly the point when learning.
3. Free asset sources with usable licences: Kenney’s public domain 2D and 3D packs; OpenGameArt; Poly Haven and AmbientCG for CC0 textures, models and environment maps; Quaternius for CC0 low-poly models; Adobe’s Mixamo for free rigged characters and animations; Freesound for Creative Commons audio; and Google Fonts. Read every licence and keep an attributions file from day one, because reconstructing it later is miserable.
4. “Juice” is the disproportionately effective polish layer: screen shake, a two to six frame hit pause, particles on impact, layered sound, tweened interface movement, and squash and stretch on jumps. The reference talk is “Juice It Or Lose It” by Martin Jonasson and Petri Purho from 2012.
5. Where to publish, with real numbers:

Platform	Cost to publish	Revenue share
itch.io	Free	You choose, 10% default
Steam	100 USD per game	30%, 25% over 10M
Epic Games Store	Free, curated	12%
Google Play	25 USD once	15% under 1M per year
Apple App Store	99 USD per year	15% under 1M per year

6. The Steam Direct fee of 100 United States dollars per title is refundable once the title reaches 1,000 dollars of adjusted gross revenue. Steam’s share drops to 25 percent above 10 million dollars of lifetime revenue and 20 percent above 50 million.
7. Game jams are the most reliable way to finish anything. Ludum Dare has run since 2002, Global Game Jam since 2009, and the GMTK Jam is now among the largest. A 48-hour deadline enforces scope better than any amount of personal discipline.
8. A practical technical checklist before publishing: test on a machine that has never had your development tools installed; check that the game runs from a path with a space and a non-English character in it; handle alt-tab, minimize and display resolution change; provide key rebinding; check it on a 16:10 and an ultrawide display; and make sure the game can be quit with the keyboard alone.

■ 45.16.6 WORDS - remember these

1. Scope — how big the thing is — the total feature and content set, and the dominant risk in any small project.
2. Cut list — the ideas you deliberately postpone — a written backlog that protects the current milestone from new ideas.
3. Prototype — the ugly playable test — a minimal implementation built to answer one design question, then usually discarded.
4. Playtest — watching someone else play — structured observation of a naive player without designer intervention.
5. Juice — cheap polish that feels expensive — screen shake, hit pause, particles and easing applied to feedback moments.
6. Game jam — a short deadline with a theme — a fixed-duration event, typically 48 to 72 hours, used to force completion.

45.98 Common wrong ideas

1. Wrong: games are made by one genius programmer. Right: a modern game is made by a team where roughly one person in five writes code, and the largest groups are artists and quality assurance. Even the famous solo games, such as *Stardew Valley*, took a single person four and a half years and still used other people's engine, tools and audio libraries.
2. Wrong: more polygons means better graphics. Right: past a certain density extra triangles are smaller than a pixel and cost performance for no visible gain. What actually makes an image look good is lighting, material response, texture quality, animation and art direction. A 2015 game with good art direction routinely looks better than a 2026 game without it.
3. Wrong: the engine makes the game. Right: the engine provides the machinery. Every game in the same engine can look and feel completely different, because the design, the art, the tuning and the content are the game. An engine can make a bad game easier to build; it cannot make it good.
4. Wrong: lag is always your internet connection. Right: what you feel as lag is the sum of your input device, your frame rate, your render queue depth, your display, your distance to the server, the server's tick rate, and the game's interpolation delay. A player with a perfect connection on a distant, low-tick server will feel worse than a player with a mediocre connection nearby.
5. Wrong: a higher average frame rate always feels smoother. Right: evenly paced frames feel smoother than unevenly paced ones at the same average. 112 frames per second average with one 100-millisecond hitch feels worse than a steady 90.
6. Wrong: game physics is real physics. Right: it is a stable, cheap approximation tuned for feel. Jump arcs, friction and gravity in most games are deliberately unphysical because physical values feel bad to play.
7. Wrong: game AI is machine learning. Right: shipped game AI is almost entirely hand-authored state machines, behaviour trees, utility scoring and pathfinding, deliberately weakened so it is fun to fight. Learned agents exist in research and beat humans, and are not what is in your game.
8. Wrong: emulators are illegal. Right: writing and using an emulator has been upheld as lawful in United States courts. Copying game data or firmware you do not own is what is unlawful, and the two questions are separate.
9. Wrong: kernel anti-cheat means the game is secure. Right: it raises the cost of software cheating, and it cannot see a direct-memory-access cheat on a second computer or a screen-reading cheat driving a hardware mouse. Server-side validation and not sending hidden data remain the only defences that cannot be bypassed on the client.

10. Wrong: consoles are just locked-down PCs. Right: the fixed target changes what is possible. Pre-compiled shaders, unified memory, a known memory budget and one optimization target let a console extract far more from the same silicon than a PC of identical specification.

45.99 Chapter summary in 20 lines

1. A game is a program that redraws the world on a clock, whether or not anything changed, unlike an application that redraws only on an event.
2. That single decision produces the frame budget: 16.67 ms at 60 frames per second, 8.33 ms at 120, 6.94 ms at 144.
3. The loop is input, update, render, present, and it never sleeps except when waiting for the display.
4. Physics needs a fixed timestep because integration is not step-size invariant; an accumulator collects real time and spends it in fixed steps.
5. Rendering interpolates between the last two simulation states using the leftover accumulator fraction, which is what makes 60 Hz physics look smooth at 144 frames per second.
6. Frame time is the real measurement; average frames per second hides stutter, which is why 1 percent lows are reported.
7. Games began as laboratory curiosities in 1958 and 1962, became wired logic in 1972, gained processors and cartridges in 1977, crashed in 1983, and were rebuilt around platform control from 1985.
8. Doom in 1993 and 3D acceleration from 1996 moved the bottleneck from pixel fill to triangle throughput, and then to per-pixel computation.
9. A modern AAA game costs 80 to 300 million dollars and employs 200 to 600 people, of whom fewer than a quarter write code; content, not code, is the cost.
10. Engines supply the reusable machinery and an editor. Unity, Unreal and Godot differ chiefly in language, licence and default strengths, and a licence is part of your technology stack, as 2023 demonstrated.
11. Entity-component-system layouts win on speed because contiguous component arrays match how caches and prefetchers work.
12. The asset pipeline converts editor formats into platform-specific runtime formats, a process Unreal calls cooking; shader permutations and lightmap baking are why builds take hours.
13. Physics is broad phase, narrow phase, and impulse-based resolution; momentum is conserved and restitution controls how much energy is lost.
14. Floating-point results are not reproducible across machines in general, which is why deterministic lockstep networking is hard.
15. Game AI is state machines, behaviour trees, utility scoring, planners and A-star pathfinding, deliberately weakened so that fighting it is fun.
16. A rendered frame is culling, batching, sorting, shadow passes, a forward or deferred lighting pass, transparency, post-processing, then interface.
17. Latency cannot be removed, only hidden: client-side prediction, server reconciliation, entity interpolation and lag compensation together explain why you get shot after reaching cover.
18. The client can never be trusted; server-side validation and not sending invisible data are the only defences that cannot be patched out.
19. Launch is loader, launcher, anti-cheat, device and swap chain creation, shader compilation, streaming and finally the loop, which is why the first run stutters and consoles do not.
20. Start tiny, cut ruthlessly, playtest silently, and publish. Scope, not skill, is what ends most projects.

Machine Learning From Scratch — Fitting, Loss, Gradients and Neural Networks

46.0 What this chapter gives you

1. You will be able to say exactly what machine learning is, in one sentence, without using the word “intelligence”, and explain how it differs from ordinary programming with a concrete example of both.
2. You will be able to tell the honest history: the perceptron, two winters, the support vector machine years, and why 2012 changed everything.
3. You will be able to fit a straight line to five points by hand, define what a parameter is, and compute the error of your fit two different ways.
4. You will be able to explain gradient descent to someone who has never seen a derivative, then work four steps of it on paper with real numbers.
5. You will be able to use the words sample, feature, label, epoch, batch, iteration and step correctly, and never confuse them again.
6. You will be able to read a training curve and say whether the model is underfitting, overfitting, or learning properly.
7. You will be able to name the main classical methods, say when each is the right tool, and explain why neural networks are not always the answer.
8. You will be able to describe one artificial neuron exactly, compute its output by hand, and prove in two lines why a network with no activation function is worthless.
9. You will be able to explain backpropagation in plain language as blame assignment, then do a complete backward pass through a small network with real numbers.
10. You will be able to say what a trained model physically is, and why training costs so much more than using the result.

46.1 What machine learning actually is

■ 46.1.1 PLAIN – in simple words

1. In ordinary programming you know the rule, and you write it down.
2. You want to convert Celsius to Fahrenheit, so you write: multiply by 1.8, then add 32. Done. The rule was in your head and now it is in the file.
3. Machine learning is for the other case: you do not know the rule, but you have a pile of examples of the rule working.
4. You have ten thousand emails, each already marked “spam” or “not spam”. You cannot write the rule that separates them. Nobody can.

5. So you do something different. You write down a **shape** for a rule, with blank spaces in it where numbers should go.
6. Then you write a procedure that fills in those numbers, by trying, checking against the examples, and adjusting.
7. When the procedure stops, the blanks are full. Those numbers are the rule.
8. Nobody wrote them. Nobody can read them. But they work on the examples, and usually on new cases too.
9. That is the whole of machine learning. A rule-shaped thing with adjustable numbers, plus a procedure that tunes the numbers to match examples.
10. At heart this is curve fitting: finding the settings that make a formula pass close to a set of points.
11. Saying “it is just curve fitting” is not dismissive. It is accurate, and the accuracy is what lets you reason about it.
12. Curve fitting in a space of a billion adjustable numbers, on data covering most of what humans have written, does astonishing things. It is still curve fitting.

■ 46.1.2 PLAIN – a picture in your head

1. Picture a sound mixing desk with a hundred sliders on it.
2. Each slider changes the sound a little. You do not know what any single one does, and no slider has a label.
3. Someone plays you a recording of how the finished mix should sound.
4. Your job is to move the sliders until what comes out of the speakers matches the recording.
5. You cannot solve this by thinking. You have to move sliders and listen.
6. So you make a small move, listen, and ask one question: closer or further?
7. If closer, keep going that way. If further, go back the other way.
8. Do this over and over, for all hundred sliders, thousands of times, and the mix creeps towards the target.
9. At the end you have a hundred slider positions that produce the right sound. You still cannot say what any one slider does.
10. The sliders are the parameters. The recording is the training data. The listening is the loss. The nudging is the optimizer.

Where this comparison breaks: a real sound engineer knows what each slider does, so they aim. Nothing here aims. Also, real learning does not move one slider and listen. It computes, in one shot, the direction every slider should move, all at once. That computation is called backpropagation and it is the single most important idea in the chapter. Finally, a mix only has to sound right once. A model has to work on sounds it has never heard, which is a much harder demand and the source of nearly every practical problem in the field.

■ 46.1.3 PLAIN – a worked example

1. Take spam detection. First, the hand-written way.

```
def is_spam_handwritten(email):
    subject = email.subject.lower()
    if "free" in subject and email.link_count > 3:
        return True
    if "viagra" in email.body.lower():
        return True
    if email.sender not in email.user_contacts:
        if email.exclamation_count > 5:
```

```

        return True
    return False

```

2. Every line of that was decided by a person. You can read it, argue with it, and fix it. Those are real advantages and they are often undervalued.
3. Now the failure. A spammer writes “FR33” instead of “free”. The rule misses.
4. You add a rule for “FR33”. They write “F.R.E.E”. You add another rule.
5. This is a race you lose, because there are more spellings than there are hours in your life.
6. Now the learned way. You do not write rules. You write a shape.
7. The shape: give every word a number, called its weight. Add up the weights of the words present. If the total is above zero, call it spam.
8. That shape has one blank per word in the vocabulary. Say 50,000 blanks.
9. You feed in 10,000 emails already marked spam or not, and let the tuning procedure fill the blanks. Afterwards the numbers look like this:

Word	Learned weight	Effect
viagra	+4.10	strong spam signal
unsubscribe	+1.35	mild spam signal
meeting	-2.60	strong not-spam
the	+0.01	says nothing
fr33	+3.80	learned, never typed

10. Nobody told the system about “fr33”. It appeared in spam in the training data, so it got a large positive weight automatically.
11. That is the real difference. The hand-written rule needed a human to notice each trick. The learned rule needs only examples containing the trick.
12. Test one email with the words “free”, “meeting” and “unsubscribe”, with weights +2.20, -2.60 and +1.35. Total: $2.20 - 2.60 + 1.35 = +0.95$.
13. Above zero, so it says spam. It is probably wrong, because “meeting” should have carried more. That is a failure of the training data, not of the idea.
14. Both approaches fail. The difference is how they fail and how you fix them. You fix the first with a code change. You fix the second with more data.

■ 46.1.4 PLAIN – what is really happening inside

1. Every machine learning system has exactly four pieces. Learn these four and nothing later will surprise you.
2. Piece one, the **model**: the shape of the rule, with blanks. A straight line, a tree of yes-or-no questions, or a network with a billion blanks.
3. Piece two, the **parameters**: the numbers that go in the blanks. Before training they are random or fixed. After training they are the answer.
4. Piece three, the **loss**: a single number saying how wrong the current parameters are on the examples. Lower is better. Zero would be perfect.
5. Piece four, the **optimizer**: the procedure that changes the parameters to make the loss smaller.
6. Training is a loop over these four. Predict with current parameters. Measure the loss. Work out which way each parameter should move. Move them a little. Repeat, often millions of times.
7. Notice what has been swapped. In ordinary programming, the logic is the thing you write and the data is what flows through it.

8. In machine learning, the data is the thing you provide and the logic is what comes out. The data has become the specification.
9. That single swap explains almost everything strange about the field. Bad data gives a bad model, in exactly the way a bad specification gives bad software, and for exactly the same reason.
10. It also explains why debugging feels so different. There is no line to step through. There is only a distribution of examples, and a number.

■ 46.1.5 TECHNICAL – the engineer’s version

1. The term was coined by Arthur Samuel of IBM, in his 1959 paper “Some Studies in Machine Learning Using the Game of Checkers”, in the IBM Journal of Research and Development. His program ran on an IBM 701.
2. The standard formal definition is Tom Mitchell’s, from his 1997 textbook Machine Learning: a program learns from experience E with respect to task T and performance measure P , if its performance at T , measured by P , improves with E .
3. Formally: choose a hypothesis class H , a set of functions parameterized by a vector θ . Choose a loss function L . Find θ that minimizes the average loss over the training data.
4. That last step is called **empirical risk minimization**. “Empirical” because you minimize measured loss on data you have, not true loss on data you do not have. The gap between the two is the whole subject of generalization.
5. The linear spam model above is logistic regression on a bag-of-words feature vector. It was the industry standard for spam filtering from the late 1990s, alongside naive Bayes, which Paul Graham popularized with his 2002 essay A Plan For Spam.
6. Side by side:

Aspect	Ordinary program	Learned model
You supply	logic	labelled examples
System supplies	output	logic (as numbers)
Debug by	reading code	inspecting data
Fails by	wrong branch	wrong distribution

7. Established fact: the “rule-shaped thing plus optimizer” description above is complete and correct for essentially every supervised learning system in production today, including the largest language models.
8. Marketing claim: that these systems reason, understand, or think. Those words have no agreed technical definition here, and vendors use them because they are unfalsifiable. Treat them as advertising until defined.
9. Active research: whether large models build internal structures that deserve to be called world models or algorithms rather than statistics. The field of mechanistic interpretability studies this. It is genuinely open, and serious researchers disagree, so do not let anyone tell you it is settled.

■ 46.1.6 WORDS – remember these

1. Machine learning — filling in blanks in a rule from examples — fitting the parameters of a hypothesis class by minimizing empirical risk.
2. Model — the shape of the rule, with blanks — the hypothesis class, a parameterized family of functions.
3. Parameter — a number the system may change during training — a component of the weight vector θ , learned by optimization.

4. Loss — one number saying how wrong you are — a scalar objective function measuring disagreement between prediction and target.
5. Optimizer — the procedure that changes the numbers — the algorithm that updates theta to reduce the loss, usually a gradient method.
6. Curve fitting — finding settings that make a formula pass near the points — function approximation from finite samples. An accurate description, not an insult.

46.2 A short honest history

■ 46.2.1 PLAIN - in simple words

1. This field is older than most people think, and it has failed publicly at least twice. Knowing that keeps you sane during the hype.
2. In 1943 two researchers, Warren McCulloch and Walter Pitts, wrote down a simple mathematical model of a nerve cell. That is where it starts.
3. In 1957 and 1958 Frank Rosenblatt built the **perceptron**, a machine that learned to tell simple pictures apart by adjusting numbers.
4. The press coverage was wild, and far ahead of the reality. That is a pattern you will see again in your lifetime.
5. In 1969 two respected researchers proved the perceptron could not do certain very simple things. Funding collapsed. This is called the first AI winter.
6. The fix, an algorithm for training networks with more layers, was invented several times between 1970 and 1974, and mostly ignored.
7. In 1986 it was published clearly, in a famous journal, and the field came back. Then it faded again in the early 1990s. Second winter.
8. Through the late 1990s and 2000s a different family of methods, based on clean mathematics rather than networks, did better in practice.
9. In 2012 a neural network won a big image contest by such a large margin that the argument ended in a single afternoon.
10. The thing that made 2012 possible was not a new idea. It was cheap parallel arithmetic hardware, plus a very large labelled dataset.

■ 46.2.2 PLAIN - a picture in your head

1. Think of a seed planted in soil that is too poor to grow it.
2. The seed is the idea of learning by adjusting numbers. It was planted in 1958 and it was a good seed.
3. The soil is three things: computing power, data, and algorithms.
4. In 1958 there was almost no computing power, almost no data, and only a one-layer algorithm. The seed sprouted and died.
5. In 1986 the algorithm arrived. Compute was better, data was still thin. The seed sprouted higher and died again.
6. By 2012 all three were rich at once: graphics chips gave the compute, the internet gave the data, and the 1986 algorithm was still sitting there unchanged. The seed grew into a tree.
7. Almost nothing about the seed changed across those fifty-four years.

Where this comparison breaks: the algorithms did improve, and pretending otherwise is unfair to a great deal of careful work. Better activation functions, better initialization, better normalization and better optimizers each moved the field, and without them the 2012 result would not have trained at all.

The honest version: compute and data were the necessary conditions, and a decade of unglamorous engineering was what turned “possible in principle” into “trains on a Tuesday”.

■ 46.2.3 PLAIN – a worked example

1. The overclaim of 1958, in detail, because it teaches you how to read news about this field.
2. Rosenblatt built the Mark I Perceptron at the Cornell Aeronautical Laboratory in Buffalo, New York, funded by the Office of Naval Research.
3. It had a 20 by 20 grid of 400 light sensors, so it saw a 400-pixel picture.
4. Its adjustable numbers were physical. They were potentiometers, small dials, turned by electric motors during training. You could watch the machine learn by watching the dials move.
5. It could learn to tell simple shapes apart. It genuinely worked, and that was a real achievement.
6. In July 1958 The New York Times reported the Navy’s expectation that the machine would be able to “walk, talk, see, write, reproduce itself and be conscious of its existence”.
7. It could sort simple shapes. The gap between those two sentences is the entire lesson of this section.
8. Eleven years later, Marvin Minsky and Seymour Papert published the book *Perceptrons*, in 1969, showing what one layer could not do.
9. Their killer example was XOR: output 1 if exactly one of two inputs is 1. Here is the whole problem.

x2			
1		1	0
0		0	1
	+	-----	
		0	1
			x1

1 means "should output 1"

A single straight line must put both 1s on one side.
No straight line can. That is the entire objection.

10. A one-layer perceptron can only draw a straight dividing line. XOR needs two lines, or a bent one. So one layer cannot do XOR.
11. That was mathematically correct and it was fatal in practice, because nobody yet had a working way to train two layers.
12. Minsky and Papert did say multi-layer networks were more capable. The reception of the book was harsher than its text, which is common.

■ 46.2.4 PLAIN – what is really happening inside

1. Why did the training method take so long to arrive, when the mathematics is just repeated use of the chain rule from school calculus.
2. Because nobody was looking for it in the place it was found. The reverse accumulation method was worked out for a completely different purpose.
3. In 1970 Seppo Linnainmaa, in his master’s thesis at the University of Helsinki, described how to compute the derivative of a whole computation efficiently by walking backwards through it.
4. His subject was rounding error in numerical algorithms. He was not thinking about learning at all. The algorithm was correct and general anyway.
5. In 1974 Paul Werbos, in his Harvard PhD thesis, applied that idea directly to training neural networks. Very few people read it.
6. In 1986 David Rumelhart, Geoffrey Hinton and Ronald Williams published “Learning representations by back-propagating errors” in *Nature*. Same idea, clearly written, in a journal everyone read, with results.

7. The lesson is not about mathematics. It is that an idea has to arrive in a readable form, in a place the right people look, at a moment when the hardware can demonstrate it.
8. The second winter came after 1987 for practical reasons: networks of the day were slow to train, needed careful hand-tuning, and were beaten on real tasks by simpler methods with better theory.
9. What broke the deadlock in 2012 was throughput. A graphics chip does the same arithmetic operation on thousands of numbers at once, which is exactly the shape of a neural network's work.

■ 46.2.5 TECHNICAL – the engineer's version

1. The timeline, with the facts you should be able to quote.

Year	Event	Why it mattered
1943	McCulloch and Pitts model	first formal neuron
1949	Hebb, Organization of Behavior	learning as weight change
1957	Rosenblatt perceptron report	first learning machine
1958	NYT overclaim	first hype cycle
1960	Widrow and Hoff ADALINE	least-mean-squares rule
1962	Novikoff convergence proof	perceptron theory
1969	Minsky and Papert book	XOR objection
1970	Linnainmaa thesis	reverse-mode derivatives
1973	Lighthill report, UK	funding cut
1974	Werbos thesis	backprop for networks
1980	Fukushima Neocognitron	ancestor of the CNN
1982	Hopfield networks	physics enters the field
1986	Rumelhart, Hinton, Williams	backprop reaches everyone
1989	LeCun, digits at Bell Labs	first real CNN application
1989	Cybenko approximation proof	one layer is enough, in theory
1995	Cortes and Vapnik, SVM	networks fall out of fashion
1997	Hochreiter and Schmidhuber	LSTM for sequences
1998	LeCun LeNet-5	cheque reading in production
2001	Breiman random forests	strong, simple, tabular
2006	Hinton deep belief nets	the name "deep learning"
2007	NVIDIA releases CUDA	general compute on GPUs
2009	ImageNet dataset published	a large labelled image corpus
2012	AlexNet wins ILSVRC	the argument ends
2014	GANs; Adam optimizer	generation; default optimizer
2015	ResNet; batch normalization	very deep networks train
2016	AlphaGo beats Lee Sedol	4-1, March 2016
2017	Attention Is All You Need	the transformer
2020	GPT-3; scaling laws	size as a strategy
2022	ChatGPT, 30 November	public arrival
2024	Nobel Prize in Physics	Hopfield and Hinton

2. The 2012 result, precisely. AlexNet, by Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton, won the ImageNet Large Scale Visual Recognition Challenge with a top-5 error rate of 15.3 percent. The second-placed entry, which did not use deep learning, scored 26.2 percent.
3. A gap of nearly 11 percentage points in a contest usually decided by one is why the field turned in a single day.

4. The training set was about 1.2 million images across 1,000 categories. The network had roughly 60 million parameters, most of them in its fully connected layers.
5. It was trained on two NVIDIA GTX 580 cards with 3 GB of memory each, for about five to six days. The 3 GB limit is why the network was split across two cards at all. A hardware constraint shaped the architecture.
6. Chapter 22 explained what a graphics chip is and why it multiplies matrices so quickly. The short version, without repeating it: these chips run thousands of identical arithmetic operations in parallel, and a neural network is thousands of identical arithmetic operations. The fit is exact and it was an accident of history.
7. ImageNet itself, led by Fei-Fei Li and published at CVPR in 2009, was arguably the more important contribution. Without a large, clean, labelled dataset there is nothing to fit.
8. The support vector machine era rests on Corinna Cortes and Vladimir Vapnik's 1995 paper "Support-Vector Networks", building on the kernel trick from Bernhard Boser, Isabelle Guyon and Vapnik in 1992. From roughly 1995 to 2010 these methods usually beat neural networks on the benchmarks of the day, and had far better theory.
9. Hinton, Yoshua Bengio and Yann LeCun shared the 2018 Turing Award, presented in 2019. In October 2024 John Hopfield and Geoffrey Hinton received the Nobel Prize in Physics for foundational discoveries enabling machine learning with artificial neural networks.
10. Historical note on naming: "backpropagation" is a shortening of "backward propagation of errors", from the 1986 Nature paper title.

■ 46.2.6 WORDS - remember these

1. Perceptron — the first machine that learned by adjusting dials — a single linear threshold unit with the 1958 update rule, convergent on linearly separable data.
2. AI winter — a period when funding and interest collapsed — the funding contractions of roughly 1974 to 1980 and 1987 to 1993.
3. XOR problem — the simple task one layer cannot do — a function that is not linearly separable, the core objection of Perceptrons, 1969.
4. Backpropagation — the way to find how each number should change — reverse-mode automatic differentiation applied to a layered network.
5. ImageNet — the giant labelled picture collection — the dataset of about 14 million images, and its annual challenge, ILSVRC, run from 2010.
6. AlexNet — the network that won in 2012 and ended the argument — an eight-layer CNN, 60 million parameters, 15.3 percent top-5 error.

46.3 Fitting a line, the foundation of everything

■ 46.3.1 PLAIN - in simple words

1. Everything in this chapter is an elaboration of one small task: draw the best straight line through some dots on a graph.
2. Understand this properly and the rest is bookkeeping.
3. You have pairs of numbers. For each one, an input and an output.
4. Hours studied and exam mark. Size of a flat and its rent. Anything.
5. Plot them. They do not lie on a perfect line, because reality is messy, but they lean in a direction.
6. You want the line that leans the same way and sits as close to all the dots as it can.
7. A straight line is described completely by two numbers. Just two.
8. The first is how steep it is. Call it w . If w is 2, then every step right makes the line go up two.

9. The second is how high it sits. Call it b . It is the height of the line where the input is zero.
10. So the line is: output equals w times input, plus b . Written short: $y = wx + b$.
11. Those two numbers, w and b , are the only things you are allowed to change.
12. A number the system is allowed to change to fit the data is called a **parameter**. This is the definition that carries through the whole book.
13. A straight-line model has two parameters. A large language model has hundreds of billions. There is no difference in kind, only in count.

■ 46.3.2 PLAIN – a picture in your head

1. Picture a long straight stick and a scattering of nails hammered into a board at different heights.
2. You must hold the stick against the board so it passes as close as possible to all the nails at once.
3. You have exactly two freedoms. You can slide the stick up and down, and you can tilt it.
4. Sliding up and down is b . Tilting is w . There is nothing else you can do to a straight stick.
5. Now the useful part. Suppose the nails clearly climb from left to right and your stick is flat. You can see immediately that tilting it will help.
6. Suppose the stick is at the right tilt but all the nails are above it. You can see that sliding it up will help.
7. That “you can see which way to move it” feeling is exactly the gradient. You already have the intuition. Later we just compute it instead of eyeballing.

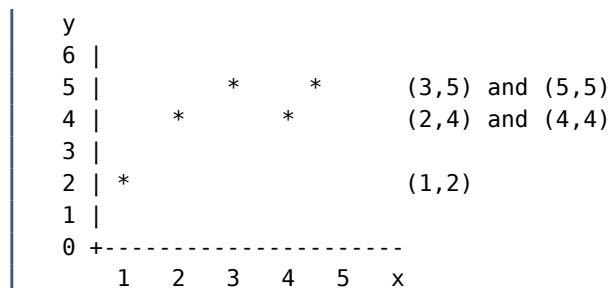
Where this comparison breaks: with two freedoms you can see the whole board and judge both at once. With a million freedoms you cannot see anything. You can only ask, at your current position, which way is downhill, and take a step. Your entire view of the landscape is the slope under your feet.

■ 46.3.3 PLAIN – a worked example

1. Here is the dataset used for the whole of this chapter. Five points. Learn them, because they come back three more times.

x	y
1	2
2	4
3	5
4	4
5	5

2. Plotted, they rise, but not smoothly. The fourth point dips.



3. First, a guess. Try $w = 1$ and $b = 0$, the line $y = x$.
4. Predictions: 1, 2, 3, 4, 5. Actual values: 2, 4, 5, 4, 5.

5. It is too low at the start and too low in the middle. Not good.
6. Now do it properly. The best straight line under the usual error measure has a formula, and we work it by hand.
7. Average of the x values: $(1+2+3+4+5) / 5 = 15 / 5 = 3$.
8. Average of the y values: $(2+4+5+4+5) / 5 = 20 / 5 = 4$.
9. Now for each point, take x minus 3, and y minus 4, and multiply them.

x-3	y-4	product	(x-3) squared
-2	-2	4	4
-1	0	0	1
0	1	0	0
1	0	0	1
2	1	2	4

10. Sum of the products: $4 + 0 + 0 + 0 + 2 = 6$.
11. Sum of the squares: $4 + 1 + 0 + 1 + 4 = 10$.
12. The slope is the first divided by the second: $w = 6 / 10 = 0.6$.
13. The height is the average y minus w times the average x: $b = 4 - (0.6 \times 3) = 4 - 1.8 = 2.2$.
14. So the best line is $y = 0.6x + 2.2$. Those two numbers are the trained model. That is the whole model. Two numbers.
15. Its predictions on our five inputs: 2.8, 3.4, 4.0, 4.6, 5.2.
16. Compare to the actual values 2, 4, 5, 4, 5. Close, never exact. That is normal and expected.

■ 46.3.4 PLAIN - what is really happening inside

1. Notice what happened there. We did not search. We solved.
2. For a straight line under squared error, there is a formula that lands on the exact best answer in one calculation. No stepping, no guessing.
3. This is called a closed-form solution. It exists for this problem because the problem is simple enough to solve with algebra.
4. Here is the important fact: for almost every model you will ever use, no such formula exists.
5. Add a bend to the curve, add a threshold, add a second layer, and the algebra stops closing. There is no formula to solve.
6. When you cannot solve, you search. You start somewhere, look at which way is downhill, and step. That is section 46.5.
7. So the line is the perfect teaching example, precisely because you can check the search against the exact answer.
8. We know the truth is $w = 0.6$, $b = 2.2$. When gradient descent creeps towards 0.6 and 2.2 later in this chapter, you will know it is working.
9. One more thing to notice. The model has no memory of the five points. After training, all that is left is 0.6 and 2.2. The data is thrown away.
10. That is true of every trained model, from this line to the largest network. The data shaped the parameters and then went home.

■ 46.3.5 TECHNICAL - the engineer's version

1. The method is ordinary least squares. Adrien-Marie Legendre published it in 1805. Carl Friedrich Gauss published in 1809 and claimed use from 1795. The priority dispute is real and unresolved.

- The closed form for simple linear regression: w equals the sample covariance of x and y divided by the sample variance of x , and b equals the mean of y minus w times the mean of x .
- In matrix form for many features, the normal equation is θ equals the inverse of X -transpose- X , times X -transpose- y .
- Computing that inverse directly is a mistake in practice. Numerically stable implementations use a QR decomposition or a singular value decomposition. The function `numpy.linalg.lstsq` uses SVD.
- Cost: the normal equation is order n times d squared, where n is samples and d is features. At d equal to 100,000 features the d -squared term is fatal, which is one practical reason iterative methods win at scale.

```
import numpy as np
x = np.array([1, 2, 3, 4, 5], dtype=float)
y = np.array([2, 4, 5, 4, 5], dtype=float)
A = np.vstack([x, np.ones_like(x)]).T
w, b = np.linalg.lstsq(A, y, rcond=None)[0]
print(w, b)           # 0.6  2.2
```

- Terminology, and be careful here. In statistics these two numbers are “coefficients” and the model is a “regression”. In machine learning they are “weights” and “bias”, and the model is a “linear layer with one output”. Same equation, two vocabularies, two research communities.
- Parameter** has a precise meaning: a value learned from data by the optimization procedure. A **hyperparameter** is a value you choose before training, which the optimizer never touches. The learning rate is a hyperparameter. The slope is a parameter. Confusing these two is the single most common vocabulary error in the field.
- Scale check for intuition:

Model	Parameters	Fits on
Straight line	2	paper
LeNet-5, 1998	about 60,000	any laptop
ResNet-50, 2015	25.6 million	one GPU
GPT-2 XL, 2019	1.5 billion	one GPU
GPT-3, 2020	175 billion	a cluster

■ 46.3.6 WORDS – remember these

- Parameter — a number the system may change to fit the data — a learned component of θ , updated by the optimizer.
- Hyperparameter — a setting you choose before training starts — a value outside θ , fixed during optimization, tuned on validation data.
- Weight — how strongly an input pushes the output — the multiplicative coefficient applied to an input feature.
- Bias — the fixed offset added regardless of input — the additive intercept term, allowing the fit to leave the origin.
- Slope — how steep the line is — the first derivative of output with respect to input, constant for a linear model.
- Closed form — an exact answer from one formula — an analytic solution, such as the normal equation, available only for simple models.

46.4 Loss: how you measure being wrong

■ 46.4.1 PLAIN – in simple words

1. To improve something you must first be able to score it.
2. The score here is called the **loss**. It is one number. It says how badly the current parameters do on the examples.
3. Lower is better. Zero would mean perfect on every example.
4. The obvious idea is to add up how far each prediction is from the truth.
5. But there is a trap. If one prediction is 3 too high and another is 3 too low, they cancel to zero, and the model looks perfect when it is not.
6. So you must remove the sign before adding. There are two natural ways.
7. First way: square each miss. A miss of 3 becomes 9. Squares are never negative, so nothing cancels. Average the squares. That is **mean squared error**.
8. Second way: drop the minus sign on each miss. A miss of minus 3 becomes 3. Average those. That is **mean absolute error**.
9. Both are sensible. They are not the same, and the choice changes what the model learns.
10. Squaring punishes big misses much harder. A miss of 10 counts a hundred times more than a miss of 1, not ten times more.
11. So squared error makes the model terrified of large mistakes, and willing to accept many small ones to avoid one large one.
12. Absolute error treats a miss of 10 as exactly ten misses of 1. It does not panic about outliers.
13. The loss is not a description of reality. It is your statement of what you care about. The model will optimize exactly what you asked for, including the parts you did not mean.

■ 46.4.2 PLAIN – a picture in your head

1. Imagine you are a delivery company and you must score your drivers.
2. Scheme one: fine a driver the square of the minutes late. Five minutes late costs 25. Twenty minutes late costs 400.
3. Under that scheme a driver will do almost anything to avoid one very late delivery, and will happily be three minutes late on many.
4. Scheme two: fine one unit per minute late, always. Twenty minutes late costs 20, exactly four times what five minutes costs.
5. Under that scheme a driver optimizes total minutes and will let one delivery run very late if it saves a little time on several others.
6. Neither scheme is wrong. They encode different values.
7. And here is the point: the drivers are not cheating. They are doing exactly what you asked. If the result is bad, your scoring scheme was bad.

Where this comparison breaks: drivers understand the rule and plan around it. A model has no plan and no understanding. It only follows the slope of the number you defined. That is worse in one way, because it will exploit loopholes no honest person would notice, and better in another, because it is not trying to deceive you. The honest version: a model does not “game” the loss. The loss is its entire universe. There is nothing else to game.

■ 46.4.3 PLAIN – a worked example

1. Take the five points again and our fitted line $y = 0.6x + 2.2$.
2. Compute every step of the mean squared error.

Point (x, y)	predicted	error	squared
(1, 2)	2.8	+0.8	0.64
(2, 4)	3.4	-0.6	0.36
(3, 5)	4.0	-1.0	1.00
(4, 4)	4.6	+0.6	0.36
(5, 5)	5.2	+0.2	0.04

3. Error means predicted minus true. Sign kept for now.
4. Sum of the squared column: $0.64 + 0.36 + 1.00 + 0.36 + 0.04 = 2.40$.
5. Divide by the number of points: $2.40 / 5 = 0.48$.
6. Mean squared error is 0.48. Written as a formula:

$$\begin{aligned} \text{MSE} &= (1/n) * \text{sum over } i \text{ of } (w * x_i + b) - y_i)^2 \\ n &= 5 \\ &= (1/5) * (0.64 + 0.36 + 1.00 + 0.36 + 0.04) \\ &= (1/5) * 2.40 \\ &= 0.48 \end{aligned}$$

7. Now mean absolute error on the same fit. Drop the signs: 0.8, 0.6, 1.0, 0.6, 0.2. Sum is 3.2. Divide by 5: 0.64.
8. Note that the errors were not squared, so the answer is in the same units as
 - y. An MAE of 0.64 means “typically wrong by about 0.64 marks”. An MSE of 0.48 is in squared marks, which is not a unit anyone can feel.
9. Taking the square root of MSE gives 0.693, the root mean squared error, back in the original units. That is why people quote RMSE in reports.
10. Now the demonstration that the choice matters. Add a sixth point that is wildly wrong: $x = 6, y = 20$. Perhaps a data entry error.
11. Refit with squared error. The best line becomes $y = 2.629x - 2.533$.
12. The slope went from 0.6 to 2.63. One bad point out of six dragged the whole line more than four times steeper.
13. Refit with absolute error. The best line becomes $y = 1.5x + 0.5$.
14. Still moved, but far less. The absolute-error fit largely ignored the outlier while the squared-error fit chased it.
15. That is not a small technical detail. That is the loss deciding what the model believes.

■ 46.4.4 PLAIN – what is really happening inside

1. Now the idea that makes optimization possible: the **loss surface**.
2. The loss depends on the parameters. Change w , the loss changes. Change b , the loss changes.
3. So imagine a flat map. East-west is the value of w . North-south is the value of b . Every point on that map is one possible model.
4. At every point, compute the loss and use it as a height. Now you have a landscape.
5. High ground means bad parameters. Low ground means good ones.
6. Training is walking downhill on this landscape. That is the entire idea.

7. For our five points, here is the real landscape. Every number is a genuine mean squared error, computed for that pair of w and b .

	$w=0.0$	$w=0.3$	$w=0.6$	$w=0.9$	$w=1.2$
$b=4.4$	1.36	2.35	5.32	10.27	17.20
$b=3.3$	1.69	0.70	1.69	4.66	9.61
$b=2.2$	4.44	1.47	0.48*	1.47	4.44
$b=1.1$	9.61	4.66	1.69	0.70	1.69
$b=0.0$	17.20	10.27	5.32	2.35	1.36

* lowest point: $w=0.6$, $b=2.2$

8. Read it like a contour map. The 0.48 in the middle is the valley floor. That is exactly the answer we computed by formula in section 46.3.
9. Look at the diagonal. Going from top-left to bottom-right, the values stay small: 1.36, 0.70, 0.48, 0.70, 1.36. That is a long shallow trough.
10. Going the other diagonal, values shoot up fast: 1.36 to 17.20.
11. So this valley is not a round bowl. It is a long narrow ravine, steep across and shallow along. Remember that shape. It causes most of the pain in section 46.5.
12. The reason the ravine is tilted is that w and b are not independent here. Making the line steeper and lowering it can partly cancel out.

■ 46.4.5 TECHNICAL - the engineer's version

1. The standard losses, with the situation each belongs to.

Loss	Formula core	Used for
MSE / L2	mean of $(\text{pred} - y)^2$	regression, default
MAE / L1	mean of $\text{abs}(\text{pred} - y)$	regression, outliers
Huber	L2 near 0, L1 far out	robust regression
Binary cross-entropy	$-y \log p - (1-y) \log(1-p)$	two-class
Cross-entropy	$-\sum y_k \log p_k$	many-class

2. Mean squared error is the maximum likelihood estimator under the assumption that the errors are Gaussian with constant variance. That is where it comes from. It is not an arbitrary choice.
3. Mean absolute error is the maximum likelihood estimator under Laplace- distributed errors, and its minimizer is the conditional median rather than the conditional mean. That is exactly why it resists outliers.
4. MSE has a continuous derivative everywhere. MAE has a kink at zero where the derivative jumps from -1 to +1 and is undefined at the point itself. Implementations return 0 or a subgradient there. This is a real, if minor, optimization nuisance.
5. Huber loss, from Peter Huber's 1964 paper, is quadratic within a threshold δ of zero and linear outside it. It gets MSE's smooth gradients near the optimum and MAE's outlier resistance far away. In object detection the same idea appears as smooth L1 loss.
6. For classification you almost never use MSE. Cross-entropy is standard, because its gradient with respect to the pre-activation of a softmax output is simply predicted probability minus target, which is clean and does not saturate. MSE combined with a sigmoid output produces vanishing gradients when the model is confidently wrong, which is the worst possible time.
7. The loss surface of linear regression under MSE is a convex quadratic, so it has exactly one minimum and no local traps. Its Hessian for our five points is the matrix with rows $[22, 6]$ and $[6, 2]$.

8. That matrix has eigenvalues 23.66 and 0.338. The ratio, 70.0, is the condition number. It is the numerical measure of the ravine we saw above, and it directly bounds how fast gradient descent can converge.
9. Observation tools: log the loss every step and plot it. In PyTorch a scalar `loss.item()` per step written to TensorBoard, Weights and Biases, or a plain CSV is enough. Anyone who trains without a loss curve is guessing.

■ 46.4.6 WORDS – remember these

1. Loss — one number saying how wrong you are — a scalar objective evaluated on predictions and targets, to be minimized.
2. Mean squared error — average of the squared misses — the L2 loss, maximum likelihood under Gaussian noise, minimized by the conditional mean.
3. Mean absolute error — average of the misses, signs dropped — the L1 loss, minimized by the conditional median, robust to outliers.
4. Loss surface — the landscape of wrongness over parameter settings — the graph of the objective as a function of theta, in as many dimensions as there are parameters.
5. Outlier — a data point far from the rest — an observation with low probability under the assumed noise model, which dominates squared loss.
6. Cross-entropy — the standard score for choosing between categories — the negative log-likelihood of the correct class under the predicted distribution.

46.5 Gradient descent: walking downhill in fog

■ 46.5.1 PLAIN – in simple words

1. You are standing somewhere on a hillside. Thick fog. You can see nothing beyond your own boots.
2. You want to reach the lowest point in the valley.
3. You cannot see the valley. But you can feel the ground under your feet, and feel which way is downhill.
4. So you do the only sensible thing. You take a small step in the steepest downhill direction. Then you stop and feel again. Then step again.
5. Repeat a few thousand times and you will end up at the bottom of something.
6. That is gradient descent. That is the entire algorithm. Everything else in this section is detail.
7. Translate it. The hillside is the loss surface from section 46.4. Your position is the current values of the parameters.
8. “Feeling which way is downhill” means computing, for each parameter, whether increasing it makes the loss go up or down, and by how much.
9. That per-parameter answer is called the **gradient**. It is a list with one number per parameter.
10. The size of your step is called the **learning rate**. You choose it. It is a hyperparameter, and it matters enormously.
11. Every neural network you have heard of was trained this way. There is no more sophisticated principle hiding behind it.

■ 46.5.2 PLAIN – a picture in your head

1. Stay with the fog, but make the ground more interesting.
2. Suppose you are in a long narrow ravine, steep on the sides, almost flat along the bottom, running away into the fog.
3. You feel the ground. The steepest direction is across the ravine, not along it, because the sides are steep and the floor is nearly level.

4. So you step across. Now you are on the other side, slightly lower. You feel again. Steepest is back across.
5. You zig-zag from wall to wall, making painfully slow progress along the valley floor, which is where you actually needed to go.
6. That is the single most common failure of plain gradient descent, and our five-point problem has exactly this shape. Its condition number is 70, which means the ravine is seventy times steeper across than along.
7. The fixes all amount to remembering which way you were already going, so the zig-zags cancel and the along-valley movement adds up. That is momentum.

Where this comparison breaks: a real hiker can see their boots, feel the whole ground under them, and take a step of any size. A model computes the slope at exactly one point and then assumes the ground keeps that slope for the whole step. If the step is long and the ground curves, that assumption is wrong and you can land higher than you started. That is not a metaphor for divergence. It is literally what divergence is.

■ 46.5.3 PLAIN – a worked example

1. First, what “the slope” means for one parameter. Move w a tiny bit and see how much the loss moves.
2. At $w = 0.6$, $b = 2.2$, the loss is 0.48. At $w = 0.7$, $b = 2.2$, it is 0.58. The loss rose by 0.10 for a rise of 0.1 in w . So the slope is about +1.0.
3. Positive slope means increasing w increases the loss. So to reduce the loss, decrease w . Always move opposite to the slope. That is the rule.
4. The update rule, for every parameter:

$$\text{new value} = \text{old value} - (\text{learning rate}) * (\text{slope for it})$$

5. For mean squared error on a line, the two slopes have exact formulas:

$$\begin{aligned} \text{slope for } w &= (2/n) * \text{sum of } (w*x + b - y) * x \\ \text{slope for } b &= (2/n) * \text{sum of } (w*x + b - y) \end{aligned}$$

6. Now run it. Start at $w = 0$, $b = 0$, which is the flat line $y = 0$. Learning rate 0.02. Four full steps, all arithmetic shown.
7. Step 1. Predictions are all 0. Errors, predicted minus true, are -2, -4, -5, -4, -5.
8. Slope for w : $(2/5) \text{ times } [(-2)(1) + (-4)(2) + (-5)(3) + (-4)(4) + (-5)(5)] = (0.4) \text{ times } [-2 - 8 - 15 - 16 - 25] = 0.4 \text{ times } -66 = -26.4$.
9. Slope for b : $(2/5) \text{ times } [-2 - 4 - 5 - 4 - 5] = 0.4 \text{ times } -20 = -8.0$.
10. Update: $w = 0 - 0.02 \text{ times } (-26.4) = +0.528$. $b = 0 - 0.02 \text{ times } (-8.0) = +0.16$.
11. Both went up, which is right, because the line started far too low and far too flat.
12. Now the whole run in a table. Every number is exact.

Step	w	b	Loss
0	0.0000	0.0000	17.2000
1	0.5280	0.1600	5.5799
2	0.8045	0.2502	2.3494
3	0.9485	0.3037	1.4469
4	1.0227	0.3377	1.1904

13. The slopes computed at each of those positions were:

Step	Slope for w	Slope for b
0	-26.400	-8.000
1	-13.824	-4.512
2	-7.200	-2.673
3	-3.711	-1.702
4	-1.874	-1.188

14. Read the loss column. 17.20, then 5.58, then 2.35, then 1.45, then 1.19. It falls fast at first and then slows. That shape is normal and you will see it in every real training run.
15. Read the slope table. The slopes shrink towards zero. That is the ground flattening as you approach the valley floor.
16. But look at w. It has overshoot. It is 1.02 and the true answer is 0.6. And b is only 0.34 when the answer is 2.2.
17. That is the ravine. The optimizer raced along the steep w direction and is crawling along the shallow b direction.
18. Continue the run and it does get there, slowly:

Steps run	w	b	Loss
10	1.0870	0.4345	1.0471
100	0.8655	1.2416	0.6472
500	0.6176	2.1365	0.4807
2000	0.6000	2.2000	0.4800

19. Two thousand steps to find two numbers that a formula gave us instantly in section 46.3. That is the price of not having a formula. For any model worth training, there is no formula, so you pay it.
20. Notice also that the loss stops at 0.48 and does not reach zero. 0.48 is the best a straight line can do on this data. The remaining error is not a failure of the optimizer. The model simply is not a perfect description of the world, and no amount of training fixes that.

■ 46.5.4 PLAIN - what is really happening inside

1. Now the learning rate problem, with real numbers from the same dataset.
2. Too small: with a learning rate of 0.0001 instead of 0.02, every step is two hundred times shorter. You would need roughly 400,000 steps instead of 2,000. It works. It just wastes your week.
3. Too large: here is what happens at 0.09.

Step	w	b	Loss
0	0.000	0.000	17.20
1	2.376	0.720	21.60
2	-0.341	0.027	27.22
3	2.696	0.927	34.40
4	-0.766	0.024	43.58
5	3.114	1.154	55.30

4. The loss is going up, and it is going up faster each step. The parameters are flinging from one side of the ravine to the other, further each time.

5. This is **divergence**. Left alone it ends in numbers too large to represent, and the loss prints as nan, meaning not a number.
6. Seeing nan in a loss log means, in the overwhelming majority of cases, the learning rate was too high. Halve it and restart.
7. For this problem the exact boundary is computable: any learning rate below 0.0845 converges, and anything above diverges. At 0.02 we were safe. At 0.09 we were not.
8. Now a fact worth knowing. That boundary depends on how the input numbers are scaled. Our x values run 1 to 5, so their average is 3, far from zero.
9. Subtract 3 from every x, so they run -2 to +2 with an average of zero. Fit the same model. The condition number drops from 70 to 2, and the safe learning rate rises from 0.0845 to 0.5.
10. Nothing about the problem changed. We only recentred the inputs, and training became about thirty-five times easier.
11. That is why every practical guide tells you to normalize your inputs. It is not superstition. It reshapes the ravine into a bowl.
12. Finally, local minima. In a landscape with hills and dips, you might walk into a small dip that is not the true bottom, and get stuck.
13. In two dimensions this is a serious worry. In a million dimensions it is much less of one, and the reason is worth understanding.
14. To be a local minimum, the ground must curve upward in every single direction at once. With a million directions, that is like flipping a million coins and getting heads every time.
15. What you meet instead are saddle points: up in some directions, down in others. A saddle is not a trap. There is always a way out, though progress near one is slow.

■ 46.5.5 TECHNICAL - the engineer's version

1. The gradient of a scalar loss L with respect to a parameter vector θ is the vector of partial derivatives, written $\text{grad } L$. It points in the direction of steepest increase, so we step along its negative.
2. The update is θ at time $t+1$ equals θ at time t minus η times $\text{grad } L$, where η is the learning rate.
3. Convergence bound for a quadratic objective: plain gradient descent is stable if and only if η is less than 2 divided by the largest eigenvalue of the Hessian. For our five-point problem that is $2 / 23.66 = 0.0845$, which is exactly the boundary observed above.
4. The number of iterations to a fixed accuracy scales with the condition number, the ratio of largest to smallest eigenvalue. Ours was 70 raw and 2 after centring the inputs.
5. The three ways to choose how much data to use per step:

Variant	Data per step	Character
Batch (full)	all n samples	smooth, slow, exact
Stochastic (SGD)	1 sample	noisy, fast, jumpy
Mini-batch	32 to 8192	the practical default

6. Mini-batch is universal in practice. Batch sizes are usually powers of two, which is a convention rather than a requirement, chosen because it maps cleanly onto hardware. Common values are 32, 64, 128, 256.
7. The gradient noise from small batches is not purely harmful. It acts as a mild regularizer and helps escape sharp regions. Very large batches often generalize slightly worse unless the learning rate

is scaled up, an effect documented by Keskar and colleagues in 2017 and by Goyal and colleagues in the same year, whose linear scaling rule trained ImageNet in one hour.

8. The optimizer family, in order of invention:

Optimizer	Year	Core idea
Momentum	1964, Polyak	keep a velocity
Nesterov	1983	look ahead, then step
AdaGrad	2011, Duchi	per-parameter scaling
RMSProp	2012, Hinton	decaying squared average
Adam	2014, Kingma and Ba	momentum plus RMSProp
AdamW	2017, Loshchilov	Adam, decoupled decay

9. Momentum keeps a running velocity v : v becomes β times v plus the gradient, then θ moves by $-\eta$ times v . Typical β is 0.9, which means the step is roughly an average of the last ten gradients. In a ravine the across-valley components alternate in sign and cancel, while the along-valley components agree and accumulate.
10. RMSProp divides each parameter's step by the square root of a running average of that parameter's recent squared gradients. Parameters with consistently large gradients get shorter steps. Hinton introduced it in a 2012 Coursera lecture and never published it. It is cited as such.
11. Adam combines both: a running mean of the gradient, a running mean of the squared gradient, and a bias correction for the fact that both averages start at zero. Defaults from the paper: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e-8$. These defaults are used almost unchanged a decade later.
12. AdamW, from Ilya Loshchilov and Frank Hutter, corrects how Adam applies weight decay. It is the standard for transformer training. Typical settings for large language model pretraining are $\beta_2 = 0.95$ and weight decay 0.1.
13. Practical starting learning rates, as of 2026, as rough guidance rather than law:

Setting	Optimizer	Typical rate
Small MLP or CNN	SGD + momentum	0.1 to 0.01
CNN fine-tuning	SGD + momentum	0.001
Transformer pretraining	AdamW	$1e-4$ to $3e-4$
Transformer fine-tuning	AdamW	$1e-5$ to $5e-5$

14. On saddle points: Dauphin and colleagues argued in 2014, in "Identifying and attacking the saddle point problem", that saddles rather than local minima are the dominant obstacle in high-dimensional non-convex optimization. This is the standard modern view, though the loss landscape of deep networks remains an area of active research rather than settled theory.

■ 46.5.6 WORDS – remember these

- Gradient — which way is uphill, and how steeply, for every parameter at once — the vector of partial derivatives of the loss with respect to θ .
- Gradient descent — repeatedly stepping downhill — iterative first-order minimization by moving against the gradient.
- Learning rate — how big a step you take — the scalar η multiplying the gradient in the update rule; a hyperparameter.
- Divergence — the loss climbing instead of falling — instability from a step size above 2 over the largest Hessian eigenvalue, usually ending in NaN.

5. Momentum — remembering the direction you were already going — an exponentially weighted moving average of past gradients, coefficient beta.
6. Adam — the optimizer most people use by default — adaptive moment estimation, combining momentum with per-parameter gradient scaling.
7. Saddle point — flat ground that is up one way and down another — a stationary point whose Hessian has both positive and negative eigenvalues.

46.6 The vocabulary of training

■ 46.6.1 PLAIN – in simple words

1. This field reuses ordinary words with narrow meanings, and half of all confusion comes from that. Fix the words now.
2. A **sample** is one example. One email. One photograph. One patient record. Also called an instance, an observation, or a row.
3. A **feature** is one measured property of a sample. The number of links in the email. The brightness of pixel 47. The patient's age.
4. A **label** is the right answer for that sample, provided by whoever made the data. "Spam". "Cat". "Recovered".
5. A **dataset** is all the samples together, with their labels.
6. Now the split, which is the most important habit in the field. You cut the dataset into three parts and treat them differently.
7. The **training set** is what the optimizer sees. The parameters are fitted to it. Usually 70 to 80 percent of the data.
8. The **validation set** is held back. You never train on it. You use it to choose between models and settings. Usually 10 to 15 percent.
9. The **test set** is held back even harder. You look at it once, at the very end, to report an honest number. Usually 10 to 15 percent.
10. Why three and not two? Because if you tune your settings by watching the validation score, you are indirectly fitting to it, so it stops being honest. The test set stays clean because you never used it to decide anything.
11. Now the counting words. An **epoch** is one complete pass through the whole training set.
12. A **batch** is a small group of samples processed together, typically 32 to 512 of them.
13. An **iteration**, also called a **step**, is one batch going through the model and producing one parameter update.
14. So: one step handles one batch, and one epoch is many steps.

■ 46.6.2 PLAIN – a picture in your head

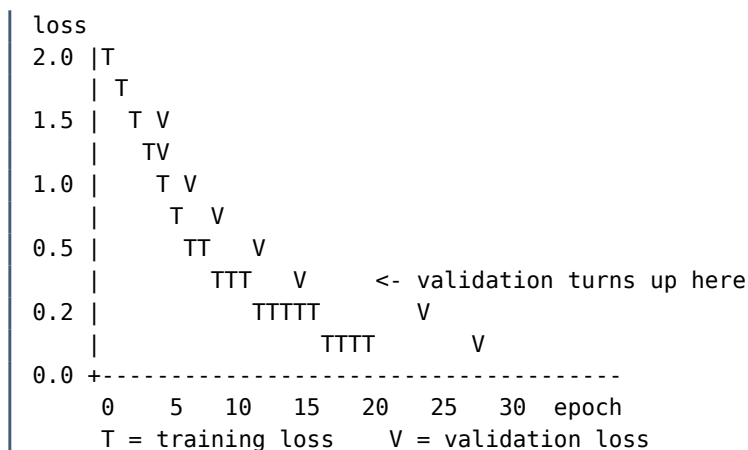
1. A student is preparing for an exam and has a book of 500 practice questions with answers at the back.
2. Studying the questions is training. The 500 questions are the training set.
3. The real exam has different questions on the same subject. That is the test set. The student has never seen it.
4. Student A learns the ideas. On the exam, new questions, good marks. That is a model that generalizes.
5. Student B memorizes all 500 questions and their answers word for word. Perfect on practice, lost in the exam. That is **overfitting**.

6. Student C did not study enough and gets both the practice and the exam wrong. That is **under-fitting**.
7. The tell is the gap. A big gap between practice performance and exam performance means memorization, not learning.

Where this comparison breaks: a student who memorizes knows they are memorizing. A model cannot tell the difference between a real pattern and a coincidence in the data, and does not experience one as different from the other. Worse, the analogy suggests memorization is always bad. It is not that simple. Very large modern models memorize a great deal of their training data and still generalize well, which classical theory did not predict. That gap between theory and observation is real, and it is active research, not settled.

■ 46.6.3 PLAIN – a worked example

1. Take 50,000 labelled photographs. Split 80 / 10 / 10.
2. Training set: 40,000. Validation set: 5,000. Test set: 5,000.
3. Choose a batch size of 100.
4. Steps in one epoch: $40,000 / 100 = 400$.
5. Train for 20 epochs. Total steps: $20 \times 400 = 8,000$ parameter updates.
6. The model has seen each photograph 20 times, and the parameters have been changed 8,000 times.
7. Now read a real training curve. Both numbers are losses. Lower is better.



8. Epochs 0 to 10: both fall together. This is real learning. Nothing to do.
9. Epochs 10 to 18: training keeps falling, validation flattens. The model is starting to fit noise. Watch it.
10. Epoch 18 onwards: training still falls, validation rises. This is overfitting, and it is unambiguous.
11. The correct action is to stop at around epoch 18 and keep those parameters. That is **early stopping**, and it is the cheapest regularizer there is.
12. Here is the opposite picture, underfitting: both curves flatten early and high, at say 1.4 and 1.45, and stay there. Small gap, bad numbers.
13. That is not a stopping problem. That means the model is too simple, or the features carry no signal, or the learning rate is wrong.
14. Read the two symptoms as a pair. Big gap, low training loss: too much capacity, add regularization. Small gap, high loss on both: too little capacity, make the model bigger.

■ 46.6.4 PLAIN – what is really happening inside

1. Underneath the two symptoms sits one trade-off, and it has a name: **bias and variance**.
2. Bias, here, does not mean unfairness. It means systematic error from a model too simple to represent the truth.
3. A straight line fitted to data shaped like a curve has high bias. It is wrong in the same way every time, no matter how much data you give it.
4. Variance means sensitivity to the particular data you happened to get.
5. A very flexible model has low bias and high variance. Show it a slightly different sample of data and it produces a noticeably different answer.
6. Total error is roughly bias squared, plus variance, plus noise that nothing can remove.
7. Simple models sit at the high-bias end. Complex models sit at the high-variance end. Classically, the best model is somewhere in between.
8. **Regularization** is anything you do to push a flexible model back towards simplicity, so you get low bias without paying full variance.
9. The four you will actually use:
10. Weight decay, also called L2: add a penalty proportional to the sum of the squared weights. Big weights become expensive, so the model prefers small ones, and small weights make gentler, smoother functions.
11. L1: penalize the sum of the absolute weights instead. This drives many weights to exactly zero, which selects features and gives a sparse model you can read.
12. Dropout: during training, at every step, randomly switch off a fraction of the units, typically half. No unit can rely on any other being present, so the network cannot build fragile chains.
13. Early stopping: stop when validation loss stops improving, as in the curve above.
14. Separately from all of those, **data augmentation**: make more training data by altering what you have in ways that do not change the label. Flip the photograph left to right, crop it, rotate it slightly, change its brightness. A flipped cat is still a cat.
15. Augmentation is often the most effective of the lot, because the real problem is usually not enough data, and this is the cheapest way to get more.

■ 46.6.5 TECHNICAL – the engineer’s version

1. Precise definitions, so the words never slip again:

Term	Exact meaning
Sample	one row of the dataset
Feature	one input dimension
Label	the target value for a sample
Batch	samples in one forward pass
Iteration / step	one parameter update
Epoch	$n_{\text{train}} / \text{batch\_size}$ steps

2. Real scale: ImageNet’s ILSVRC training set is 1,281,167 images. At batch size 256 that is 5,004 iterations per epoch. The classic ResNet recipe runs 90 epochs, so about 450,000 parameter updates.
3. Splitting must respect structure. If several rows come from the same patient, the same user or the same day, they must not be split across train and test, or information leaks and your test number is a fantasy. Use grouped or time-based splits when the data has groups or an ordering.

4. K-fold cross-validation splits the data into k parts, trains k times with a different part held out each time, and averages. It gives a far more stable estimate on small datasets, at k times the cost. Common k is 5 or 10. It is rarely used for large deep networks purely because of cost.
5. L2 regularization adds λ times the sum of squared weights to the loss. Its gradient contribution is $2\lambda w$, so each step multiplies every weight by a factor slightly under one. With λ 0.01 and learning rate 0.02, that factor is 0.99960, and 1,000 steps of pure decay would shrink a weight of 0.6 to 0.402.
6. In Adam, adding L2 to the loss and applying weight decay directly are not equivalent, because Adam rescales the gradient per parameter. AdamW, from Loshchilov and Hutter in 2017, decouples them. This is why AdamW replaced Adam for transformer training, and it is a real bug fix, not a preference.
7. Dropout, from Srivastava, Hinton and colleagues, published in JMLR in 2014, with an earlier 2012 preprint. At training time each unit is kept with probability $1-p$. At inference nothing is dropped. Standard implementations use inverted dropout: divide activations by $1-p$ during training so the expected value matches, and inference needs no adjustment. Typical p is 0.5 for fully connected layers and 0.1 for transformer residual paths.
8. Batch normalization interacts with dropout badly, and modern convolutional networks often use batch normalization with little or no dropout.
9. Standard augmentations by field:

Field	Common augmentations
Images	flip, crop, colour jitter
Images, stronger	mixup, cutmix, RandAugment
Audio	time and frequency masking
Text	back-translation, dropout

10. The classical bias-variance curve is U-shaped: error falls, then rises with model size. Established fact for classical models.
11. Active research, and important: Belkin and colleagues showed in 2019 that for very over-parameterized models the test error falls again past the interpolation point. This is called **double descent**, and it means the classical U-shape is only the left half of the real picture. The theory is incomplete and the practical implications are still debated.
12. Practically, as of 2026, the standard recipe for very large models is not “find the sweet spot on the U”. It is “make the model large, train on far more data, regularize lightly, and stop before the data runs out”. That recipe is empirical, and it works.

■ 46.6.6 WORDS – remember these

1. Sample — one example — one row of the dataset, one point in feature space.
2. Feature — one measured property — one input dimension of the model.
3. Label — the right answer for a sample — the supervision target y .
4. Epoch — one full pass over the training data — $\text{ceil}(n / \text{batch size})$ optimizer steps.
5. Batch — the group processed together — the set of samples whose gradients are averaged into one update.
6. Overfitting — memorizing the practice questions — low training error with a large gap to validation error; fitting sample noise.
7. Underfitting — not learning enough — high error on both sets; insufficient model capacity or bad optimization.

8. Regularization — anything that pushes towards simpler solutions — a penalty or constraint that reduces effective capacity, trading bias for variance.
9. Data augmentation — making more examples by changing existing ones — label- preserving transformations that expand the effective training distribution.

46.7 Classical machine learning before neural networks

■ 46.7.1 PLAIN - in simple words

1. Neural networks are not the field. They are one family in it, and for a large amount of real work they are the wrong choice.
2. **Linear regression** predicts a number by adding up weighted inputs. That is section 46.3, with more than one input.
3. **Logistic regression** predicts a yes-or-no answer. It adds up weighted inputs the same way, then squashes the total into a probability between 0 and 1. Despite the name, it does classification.
4. A **decision tree** asks a series of yes-or-no questions and follows the answers to a leaf. “Is the income above 40,000? Then, is the age under 25?” You can read it out loud and a person will understand it.
5. One tree is easy to read and usually not very accurate. It clings to accidents in the data.
6. A **random forest** grows hundreds of different trees on different random parts of the data, and lets them vote. The mistakes are different, so they partly cancel.
7. **Gradient boosting** grows trees one after another, where each new tree is trained to fix the errors the current set makes. This is usually the most accurate method available for table-shaped data.
8. **k-nearest neighbours** does not train at all. To classify a new point, it finds the k most similar stored points and takes the majority answer.
9. **k-means clustering** takes unlabelled data and sorts it into k groups by similarity. No right answers are provided.
10. A **support vector machine** draws the dividing line that leaves the widest possible empty corridor between the two classes.
11. Every one of these is still in production somewhere, today, doing useful work.

■ 46.7.2 PLAIN - a picture in your head

1. Think of a toolbox with a hammer, a screwdriver, a spanner and a saw.
2. Linear regression is the tape measure. Cheap, fast, always the first thing you reach for, and it tells you a lot before you do anything clever.
3. A decision tree is a flowchart pinned to the wall. Anybody can follow it and anybody can argue with it.
4. A random forest is asking three hundred people the same question and taking the majority. No individual is reliable. The crowd is.
5. Gradient boosting is one person answering, then a second person who only corrects the first one’s mistakes, then a third who corrects what is left.
6. k-nearest neighbours is asking your five closest neighbours what they did.
7. A neural network is a machine tool. Enormously capable, expensive to set up, and absurd for hanging a picture.

Where this comparison breaks: real tools have obvious jobs and you can see when you are using the wrong one. Here you cannot, because every method returns a number for every input, and a badly chosen method still returns confident answers. The only way to know is to try several and measure

honestly on held- out data. The honest version: the choice is empirical, and anyone who tells you which method wins before seeing your data is guessing.

■ 46.7.3 PLAIN – a worked example

1. Logistic regression, worked fully. Predict whether a student passes, from hours studied.
2. The model is: $z = 0.8 \text{ times hours, minus } 1.2$. Then squash z with the sigmoid function, which is 1 divided by (1 plus e to the minus z).
3. Student studied 3 hours. $z = (0.8 \times 3) - 1.2 = 2.4 - 1.2 = 1.2$.
4. Sigmoid of 1.2 = $1 / (1 + e^{-1.2}) = 1 / (1 + 0.3012) = 0.7685$.
5. So the model says 76.85 percent chance of passing.
6. If the student did pass, the loss is minus the natural log of 0.7685, which is 0.263. Small loss, because the model was mostly right.
7. If the student failed, the loss is minus the log of $(1 - 0.7685) = \text{minus log of } 0.2315 = 1.463$. Much bigger loss. Correctly punished.
8. Now k-nearest neighbours, worked fully. New point at (3, 4). Five stored points with known classes:

Point	Class	Distance from (3,4)
(4, 4)	B	1.00
(2, 5)	B	1.41
(3, 6)	B	2.00
(1, 2)	A	2.83
(6, 1)	A	4.24

9. With $k = 3$, the three closest are all B, so the answer is B.
10. With $k = 5$, the vote is three B against two A, so still B, but less confidently.
11. Notice there was no training. The distances were computed at question time, against every stored point. That is why it is slow to use and instant to “train”.
12. Notice also that the distances mix the two axes as if they were comparable. If the first axis were rupees and the second were years, this would be nonsense. k-nearest neighbours demands that you scale your features first.

■ 46.7.4 PLAIN – what is really happening inside

1. Now the four kinds of learning, distinguished by what you give the system.
2. **Supervised learning:** every sample comes with the right answer. The system learns to map input to answer. Spam or not. Cat or dog. House price. This is most of the industry.
3. **Unsupervised learning:** no answers at all. The system finds structure in the data itself. Grouping customers. Compressing images. Spotting anomalies.
4. **Self-supervised learning:** the answers are generated automatically from the data by hiding part of it. Take a sentence, hide a word, and the label is the hidden word.
5. That last one deserves attention, because it is how every large language model is trained. Nobody labelled the internet. The data labels itself by the trick of prediction.
6. Self-supervised learning is why scale became possible. Human labelling costs money per example. Hiding a word costs nothing, so the amount of training data is limited only by how much text exists.
7. **Reinforcement learning:** no labels, only rewards. The system acts, receives a score, and must work out which of its actions caused the score.

8. The hard part of reinforcement learning is that the reward arrives late. You lose a game of chess at move 60. Which of the 60 moves was the bad one? That is the credit assignment problem, and it makes reinforcement learning far harder and far less stable than the other three.
9. One clean way to keep them straight: supervised has answers, unsupervised has none, self-supervised manufactures its own, reinforcement has only scores.

■ 46.7.5 TECHNICAL - the engineer's version

1. When each method is right:

Method	Best when	Weak when
Linear regression	few features, linear	curved relations
Logistic regression	need calibrated odds	complex boundaries
Decision tree	must explain decision	needs accuracy
Random forest	tabular, low tuning	wide sparse data
Gradient boosting	tabular, top accuracy	needs tuning
k-nearest neighbours	tiny data, odd shapes	many features
k-means	quick grouping	non-round clusters
SVM	small data, clear margin	over 100k samples
Neural network	images, audio, text	small tabular data

2. Origins, with dates: decision trees as CART, from Breiman, Friedman, Olshen and Stone in 1984, and ID3 from Ross Quinlan in 1986. Random forests from Leo Breiman in 2001. Gradient boosting from Jerome Friedman in 2001. k-means from Stuart Lloyd in 1957, published 1982, named by James MacQueen in 1967. k-nearest neighbours from Evelyn Fix and Joseph Hodges in 1951, with the classic analysis by Cover and Hart in 1967.
3. The practical boosting libraries: XGBoost, from Tianqi Chen and Carlos Guestrin in 2016; LightGBM from Microsoft in 2017; CatBoost from Yandex in 2017. Between them they won a very large share of Kaggle competitions on tabular data from 2016 onwards.
4. The honest note on tabular data, which is what most companies actually have. Established fact: for table-shaped data with mixed numeric and categorical columns, gradient-boosted trees have been the strongest default for a decade, and remain a leading choice in 2026. The 2022 paper by Grinsztajn, Oyallon and Varoquaux, "Why do tree-based models still outperform deep learning on typical tabular data", is the standard citation, and gives the reasons: trees handle uninformative features, irregular target functions and unnormalized columns well, and neural networks handle none of them well.
5. Active research, and genuinely moving: tabular foundation models, starting with TabPFN in 2022 and TabPFN v2 published in Nature in January 2025, are pretrained transformers that make predictions on a new small table without fitting it in the usual sense. On benchmarks of small datasets, roughly under a few tens of thousands of rows, they now often beat tuned gradient boosting. On larger tables and heavily categorical tables, boosted trees still generally lead. This boundary is moving, so check the current benchmarks rather than trusting a book on this point.
6. SVMs use the hinge loss and maximize the margin, the distance from the decision boundary to the closest points, which are the support vectors. The kernel trick, from Boser, Guyon and Vapnik in 1992, computes inner products in a high-dimensional space without ever building the coordinates. The radial basis function kernel is the usual default.

7. Cost matters: the standard SVM solver is between quadratic and cubic in the number of samples, which is why SVMs are rare above roughly 100,000 samples and were displaced partly for that reason.
8. k-means minimizes within-cluster sum of squares. Lloyd's algorithm is not guaranteed to find the global optimum, so you restart it several times. The k-means++ initialization of Arthur and Vassilvitskii, 2007, is the standard default and is what `sklearn.cluster.KMeans` uses.
9. Reinforcement learning landmarks: Q-learning from Chris Watkins in 1989, temporal difference learning from Richard Sutton in 1988, Deep Q-Networks from DeepMind in Nature in 2015, AlphaGo defeating Lee Sedol 4-1 in March 2016, and reinforcement learning from human feedback, which is how chat models are aligned after pretraining and is covered later in this book.

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split

Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.2,
                                     random_state=0)
clf = GradientBoostingClassifier(n_estimators=200,
                                learning_rate=0.05,
                                max_depth=3).fit(Xtr, ytr)

print(clf.score(Xte, yte))
```

10. Note the hyperparameters in that snippet: number of trees, learning rate and tree depth. Boosting has a learning rate too, for the same reason gradient descent does, and 0.05 with 200 trees is a reasonable default.

■ 46.7.6 WORDS - remember these

1. Logistic regression — a linear model that outputs a probability — a generalized linear model with a sigmoid link, fitted by minimizing binary cross-entropy.
2. Decision tree — a readable flowchart of yes-or-no questions — a recursive partition of feature space, split by information gain or Gini impurity.
3. Random forest — many different trees voting — bagged trees with random feature subsets at each split, reducing variance.
4. Gradient boosting — each new tree fixes the last one's mistakes — stage-wise additive modelling fitting each learner to the current negative gradient.
5. Support vector machine — the widest empty corridor between two classes — a maximum-margin classifier with hinge loss, optionally kernelized.
6. Supervised learning — learning from labelled answers — estimating a mapping from inputs to targets given paired examples.
7. Self-supervised learning — the data provides its own answers — training on a pretext task built by hiding part of the input, such as next-token prediction.
8. Reinforcement learning — learning from scores, not answers — optimizing a policy to maximize expected cumulative reward from delayed feedback.

46.8 The neuron

■ 46.8.1 PLAIN – in simple words

1. An artificial neuron is a tiny arithmetic unit. It does exactly three things and then it stops.
2. One: it takes several numbers in, and multiplies each by its own weight.
3. Two: it adds all those products together, then adds one more number called the bias.
4. Three: it passes that single total through a fixed non-straight function, and that answer is its output.
5. That is all. Multiply, add, bend. There is no memory, no decision, no state.
6. Steps one and two are just the line from section 46.3 with more inputs: $\text{output} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$.
7. Step three, the bend, is the only genuinely new part, and section 46.9 explains why it is essential.
8. A weight says how much that input matters, and in which direction. Large positive means “this pushes the answer up hard”. Negative means it pushes down. Near zero means the neuron ignores it.
9. The bias is the neuron’s own starting opinion, applied before any input is considered.
10. A network is thousands to billions of these, arranged in rows, each row feeding the next.

■ 46.8.2 PLAIN – a picture in your head

1. Picture a small committee vote where the members have unequal power.
2. Each member reports a number. Each member’s report is multiplied by their voting power, which is the weight.
3. The chair has a fixed opinion held before anyone speaks. That is the bias.
4. Everything is added into one total, and a rule converts the total into the committee’s answer. That rule is the activation function.
5. A member with a negative weight is one whose enthusiasm counts against the proposal. That is normal and useful.
6. A member with weight zero has been muted. Their report changes nothing.

Where this comparison breaks: committee members reason about the proposal and about each other. Nothing here does. The weights are fixed numbers that were tuned by an optimizer, not opinions held by anyone. And a real committee can deadlock or refuse to answer. A neuron always produces exactly one number, for every input, instantly, forever.

■ 46.8.3 PLAIN – a worked example

1. One neuron, three inputs. Weights 0.5, -0.2 and 0.1. Bias 0.3.
2. Inputs arrive: 2.0, 3.0 and -1.0.

$$\begin{array}{rcl}
 x_1 = 2.0 & \text{---} [w_1 = 0.5] & \text{---} \backslash \\
 x_2 = 3.0 & \text{---} [w_2 = -0.2] & \text{---} \rightarrow (+) \rightarrow z \rightarrow f(z) \rightarrow \text{output} \\
 x_3 = -1.0 & \text{---} [w_3 = 0.1] & \text{---} / \quad \quad \quad \wedge \\
 & & \quad \quad \quad | \\
 & & b = 0.3
 \end{array}$$

3. Multiply each input by its weight:
4. $2.0 \times 0.5 = 1.0$
5. $3.0 \times -0.2 = -0.6$
6. $-1.0 \times 0.1 = -0.1$
7. Add them: $1.0 - 0.6 - 0.1 = 0.3$
8. Add the bias: $0.3 + 0.3 = 0.6$. Call this z , the pre-activation.
9. Now apply the bend. Three common choices, all on the same $z = 0.6$:

Function	Output at $z = 0.6$
ReLU	0.600
Sigmoid	0.646
tanh	0.537

10. That is a complete neuron evaluation. Six multiplications and additions, then one function call.
11. Read the weights. Input one matters most and pushes up. Input two pushes down, less strongly. Input three barely matters at all.
12. And notice: input two had the largest raw value, 3.0, but contributed less than input one, because its weight was smaller. Size of input and importance of input are different things.

■ 46.8.4 PLAIN - what is really happening inside

1. Geometrically, the weighted sum plus bias defines a plane cutting through the input space.
2. The neuron is measuring which side of that plane the input lies on, and how far from it.
3. The weights set the direction the plane faces. The bias sets how far the plane sits from the origin.
4. So a single neuron with a threshold is a straight-line separator, and nothing more. It is exactly the 1958 perceptron.
5. This is why one neuron cannot do XOR: one plane, two regions, and XOR needs the two “1” corners on the same side, which no plane achieves.
6. Layers fix it, because the second layer separates a space that the first layer has already bent.
7. Now the honest part about the brain. A neuron in your head is a living cell. It fires discrete spikes whose timing carries information. It has thousands of synapses with their own chemistry, it changes its own structure, it is regulated by hormones, and it is affected by nearby cells that are not even neurons.
8. The artificial version is a weighted sum and a bend. It does not spike, has no timing, no chemistry and no internal state.
9. The name came from the 1943 McCulloch and Pitts paper, which was genuinely an attempt to model nerve cells. The name stuck long after the modelling ambition was dropped.
10. The honest version: the brain analogy is historical and, today, mostly marketing. Calling it a neuron tells you where the idea came from. It tells you nothing about how brains work, and nothing about how networks work. Real computational neuroscience uses far more detailed models and does not use these.

■ 46.8.5 TECHNICAL - the engineer's version

1. Formally, for input vector x in $\mathbb{R}^n$, weight vector w in $\mathbb{R}^n$ and bias b in $\mathbb{R}$: $z = w\text{-transpose } x + b$, and $a = f(z)$, where f is the activation.
2. z is the pre-activation or logit. a is the activation or unit output.
3. Cost per neuron: n multiply-accumulate operations, counted as $2n$ floating-point operations, plus one activation evaluation.
4. Storage per neuron: $n + 1$ parameters. In float32 that is $4(n+1)$ bytes; in bfloat16, $2(n+1)$ bytes.
5. The set of points where z equals zero is a hyperplane of dimension $n-1$. The vector w is its normal, and the perpendicular distance from the origin is the absolute value of b divided by the Euclidean norm of w .
6. The original 1958 perceptron used the Heaviside step as f , which has a derivative of zero everywhere it is defined and is undefined at zero. That is precisely why perceptrons could not be trained by gradient descent, and why the switch to smooth activations mattered so much.
7. Biological comparison, with real figures:

Property	Biological neuron	Artificial unit
Signal	timed voltage spikes	one real number
Inputs	~1,000 to 10,000	as designed
Firing rate	up to ~200 Hz	not applicable
State	membrane potential	none

- The human brain has roughly 86 billion neurons, a figure from Suzana Herculano-Houzel's 2009 work, with of the order of 100 trillion synapses. Parameter counts of large models are sometimes compared to synapse counts. That comparison is not meaningful, because the units are not comparable, and you should treat it as a marketing claim.
- Spiking neural networks and neuromorphic hardware, such as Intel's Loihi 2 and IBM's TrueNorth, do model timing and spikes. They are active research and are not what runs any deployed large model today.

■ 46.8.6 WORDS - remember these

- Neuron / unit — a weighted sum followed by a bend — a parameterized affine map composed with a fixed non-linearity.
- Weight — how much an input matters and in which direction — a learned coefficient in the affine map.
- Bias — the unit's fixed starting offset — the additive constant, shifting the hyperplane off the origin.
- Pre-activation — the total before the bend — z , the logit; the affine output before f is applied.
- Activation — the neuron's final output — $a = f(z)$, the value passed to the next layer.
- Hyperplane — the flat surface a neuron measures distance from — the set where w -transpose $x + b$ equals zero, of dimension $n-1$.

46.9 Activation functions

■ 46.9.1 PLAIN - in simple words

- Every neuron ends with a bend, and without the bend the whole network is worthless. Here is the proof, in two lines of algebra.
- Take two layers with no bend. Layer one computes $W_1 x + b_1$. Layer two takes that and computes $W_2 (W_1 x + b_1) + b_2$.
- Multiply out: $W_2 W_1 x + W_2 b_1 + b_2$. Now let W' be $W_2 W_1$ and let b' be $W_2 b_1 + b_2$. Both are just fixed matrices of numbers.
- The result is $W' x + b'$. That is one single layer.
- So a hundred layers with no bend collapse into one layer. Depth buys nothing at all. Not less than expected. Exactly nothing.
- The bend is what stops the collapse. Once you put a non-straight function between the layers, the multiplication cannot be folded up, and the second layer sees a genuinely reshaped space.
- The function that provides the bend is called the **activation function**.
- There are only a handful in common use, and almost all of modern practice is one of them: ReLU, which is "if the number is negative, make it zero; otherwise leave it alone".
- That is genuinely the whole of ReLU. One comparison and one selection.

■ 46.9.2 PLAIN – a picture in your head

1. Picture folding a flat sheet of paper.
2. A linear layer stretches, rotates and slides the sheet. However many times you do that, the sheet stays flat, and you could have done it in one move.
3. ReLU folds it. Everything below the crease is flattened onto the crease line, and everything above is untouched.
4. Fold, stretch, fold, stretch, twenty times, and the sheet is a complicated crumpled shape that no single stretch could produce.
5. Now a straight cut through the crumpled sheet is, when you unfold it, a complicated curved boundary in the original space.
6. That is exactly what a deep network with ReLU is: many folds, then one straight cut at the end.

Where this comparison breaks: paper folds along a line you choose by hand. Here the crease positions are the weights and biases, and they are learned. Also, paper is two-dimensional, so you can see the folds. In 512 dimensions with thousands of folds there is nothing to see, and the intuition stops helping long before the mathematics does.

■ 46.9.3 PLAIN – a worked example

1. The same input z , through four different activations.

z	sigmoid	tanh
-2.0	0.1192	-0.9640
-1.0	0.2689	-0.7616
0.0	0.5000	0.0000
1.0	0.7311	0.7616
2.0	0.8808	0.9640

z	ReLU	GELU
-2.0	0.00	-0.0455
-1.0	0.00	-0.1587
0.0	0.00	0.0000
1.0	1.00	0.8413
2.0	2.00	1.9545

2. Notice sigmoid never leaves the range 0 to 1, and tanh never leaves -1 to 1. Both are squashers.
3. Notice ReLU has no upper limit. Feed it 1000 and it returns 1000.
4. Notice GELU is almost ReLU for large inputs but is smooth near zero and allows a small negative output. That smoothness is why it is preferred in transformers.
5. Now the vanishing gradient, worked with numbers. The steepness of sigmoid at any point is s times $(1 - s)$, where s is its output.
6. At $z = 0$, $s = 0.5$, so the steepness is $0.5 \times 0.5 = 0.25$. That is the maximum it ever reaches.
7. At $z = 6$, $s = 0.9975$, so the steepness is $0.9975 \times 0.0025 = 0.0025$. A hundred times flatter.
8. Now recall that backpropagation multiplies these steepness values together, once per layer. Best case, every layer contributes 0.25.

Layers of sigmoid	Best-case gradient factor
1	0.25
3	0.0156
5	0.00098
10	0.00000095

- After ten sigmoid layers, the gradient reaching the first layer is at best about one millionth of the gradient at the output, and in practice far less.
- So the early layers barely move. They are effectively frozen at their random starting values. The network trains its last two layers and nothing else.
- That is the vanishing gradient problem, and it is the single technical reason deep networks did not work before roughly 2010.
- ReLU's steepness is exactly 1 for every positive input. Multiply 1 by itself fifty times and you still have 1. The problem disappears.

■ 46.9.4 PLAIN – what is really happening inside

- So why did ReLU win, when it looks crude next to a smooth curve?
- Reason one, the gradient. Slope of exactly 1 on the positive side, so gradients pass through many layers undamaged.
- Reason two, speed. ReLU is one comparison. Sigmoid needs an exponential, which is many times more expensive, and it is evaluated billions of times.
- Reason three, sparsity. Roughly half the units output exactly zero for a given input. Zeros propagate nothing and cost nothing, and the resulting sparse representations seem to help.
- ReLU has one real failure, called the dying ReLU. If a unit's weights land such that its pre-activation is negative for every input in the data, its output is always zero, its gradient is always zero, and it can never recover. It is dead for the rest of training.
- Leaky ReLU fixes this by giving the negative side a small slope, typically 0.01, instead of zero. A dead unit still receives a trickle of gradient.
- In practice, plain ReLU with sensible initialization dies rarely enough that leaky ReLU is optional rather than standard, which is a somewhat surprising empirical result.
- GELU and SiLU are the modern smooth alternatives. Both look like ReLU from a distance, but curve gently through zero and dip slightly negative just below it.
- There is no strong theory for why that small dip helps. The honest answer is that it measured better on transformer benchmarks and the field adopted it.

■ 46.9.5 TECHNICAL – the engineer's version

- The reference table:

Name	Definition	Range
Sigmoid	$1/(1+e^{-z})$	(0, 1)
Tanh	$(e^{z-e}-z)/(e^{z+e}-z)$	(-1, 1)
ReLU	$\max(0, z)$	[0, inf)
Leaky ReLU	$\max(0.01z, z)$	(-inf, inf)
GELU	$z * \Phi(z)$	(-0.17, inf)
SiLU / Swish	$z * \text{sigmoid}(z)$	(-0.28, inf)
Softmax	$e^{z_i} / \sum e^{z_j}$	(0, 1), sums to 1

2. Where each is used today, as of 2026:

Activation	Typical use
ReLU	CNNs, most MLPs
GELU	BERT, GPT-2, GPT-3
SiLU / SwiGLU	Llama, Mistral, Qwen
Sigmoid	binary output layer only
Tanh	LSTM gates, some RL heads
Softmax	multi-class output layer

- Derivatives, which is what backpropagation actually needs. Sigmoid: $s(1-s)$, maximum 0.25 at $z = 0$. Tanh: $1 - \tanh^2$, maximum 1.0 at $z = 0$. ReLU: 1 for z above 0, 0 below, undefined at exactly 0, where implementations return 0 by convention.
- Tanh's maximum derivative of 1.0 versus sigmoid's 0.25 is why tanh was preferred over sigmoid through the 1990s and 2000s. It is zero-centred, which also keeps the mean activation near zero. It still saturates at both ends, so it only postpones the problem.
- GELU, from Dan Hendrycks and Kevin Gimpel in 2016, is z times the standard normal cumulative distribution function. The common tanh approximation is $0.5z(1 + \tanh(\sqrt{2/\pi})(z + 0.044715z^3))$. Both forms are shipped in PyTorch, selected by the `approximate` argument.
- SiLU, also called Swish, appeared in Elfwing, Uchibe and Doya in 2017 and in Ramachandran, Zoph and Le's neural architecture search paper the same year.
- SwiGLU, from Noam Shazeer in 2020, is a gated variant using SiLU. It is the feedforward activation in the Llama family and much of the current open model ecosystem. It uses three weight matrices rather than two, so implementations shrink the hidden dimension to keep the parameter count equal.
- Softmax is not a per-neuron activation. It operates on a whole output vector, turning logits into a probability distribution. Implementations always subtract the maximum logit before exponentiating, to avoid overflow. This is called the log-sum-exp trick and it is an implementation detail every library handles for you.
- The vanishing gradient problem was identified in Sepp Hochreiter's 1991 diploma thesis and analysed by Bengio, Simard and Frasconi in 1994. The move to ReLU came through Nair and Hinton in 2010 and Glorot, Bordes and Bengio in 2011, and reached everyone through AlexNet in 2012.

■ 46.9.6 WORDS – remember these

- Activation function — the bend after the weighted sum — the elementwise non-linearity f applied to the pre-activation.
- ReLU — negative becomes zero, positive passes through — the rectified linear unit, $\max(0, z)$, with derivative 1 on the positive side.
- Vanishing gradient — the learning signal fading out before it reaches the early layers — the product of many derivatives below 1 decaying towards zero exponentially in depth.
- Saturation — the region where a squashing function stops responding — where the derivative is near zero because the input is far from zero.
- Dying ReLU — a unit stuck outputting zero forever — a unit whose pre-activation is negative across the entire data distribution, so its gradient is permanently zero.
- Softmax — turning scores into probabilities that add to one — the normalized exponential over a logit vector, used with cross-entropy.

46.10 Layers and depth

■ 46.10.1 PLAIN – in simple words

1. A **layer** is a row of neurons that all look at the same inputs and all produce outputs at the same time.
2. Neurons in one layer do not talk to each other. They work in parallel, which is exactly why this runs so well on the hardware from Chapter 22.
3. Stack layers so each one's outputs are the next one's inputs, and you have a network.
4. The **input layer** is not really a layer of neurons. It is just the numbers you feed in. Counting it is a convention some books follow and some do not.
5. **Hidden layers** are the middle ones. They are called hidden because nothing outside the network ever sees their values.
6. The **output layer** is the last one, and its size is set by the task. One unit for a single number. Ten units for ten categories.
7. **Width** is how many neurons are in a layer. **Depth** is how many layers there are.
8. "Deep learning" means, roughly, more than two or three hidden layers. There is no official threshold. It is a marketing term that stuck.
9. What depth actually buys you is composition. Early layers find simple patterns, later layers combine those into complicated ones.
10. In a picture network, the first layer finds edges, the third finds corners and textures, the tenth finds eyes and wheels, the last finds faces and cars. Nobody designed that. It emerges from training.

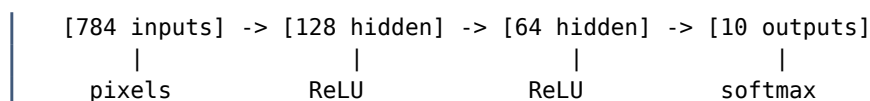
■ 46.10.2 PLAIN – a picture in your head

1. Think of a factory production line rather than one craftsman.
2. Station one takes raw metal and cuts flat pieces. That is all it does, and it does it to everything.
3. Station two takes flat pieces and bends them into brackets.
4. Station three joins brackets into frames. Station four fits frames into cabinets.
5. No station understands the cabinet. Each does one narrow transformation of whatever arrives.
6. Depth is the number of stations. Width is how many workers stand at each station doing that station's job in parallel.
7. Adding stations lets you build more complicated products. Adding workers per station lets you build more different things at each stage.

Where this comparison breaks: a factory line is designed, and someone knows what each station is for. Here nobody assigns jobs. The stations arrive at their own division of labour during training, and often it does not decompose neatly at all. Some units respond to combinations of things that have no name. The tidy edges-to-faces story is real for convolutional image networks and was demonstrated by visualization work in the 2010s, but it is a summary of typical behaviour, not a law.

■ 46.10.3 PLAIN – a worked example

1. A network for handwritten digits. Input is a 28 by 28 greyscale image, which is 784 numbers. Output is 10 scores, one per digit.
2. Shape: 784 inputs, then a hidden layer of 128, then a hidden layer of 64, then 10 outputs.



3. Parameter count, layer by layer. Each layer has (inputs x outputs) weights plus (outputs) biases.

4. Layer 1: $784 \times 128 = 100,352$ weights, plus 128 biases = 100,480.
5. Layer 2: $128 \times 64 = 8,192$ weights, plus 64 biases = 8,256.
6. Layer 3: $64 \times 10 = 640$ weights, plus 10 biases = 650.
7. Total: $100,480 + 8,256 + 650 = 109,386$ parameters.
8. Note how lopsided that is. The first layer holds 92 percent of the parameters, purely because 784 inputs is a lot to connect to.
9. This is a real network. Trained for a few minutes it reaches roughly 97 to 98 percent accuracy on the MNIST test set.
10. Now change the shape and see the trade. Width 512 in one hidden layer: $784 \times 512 + 512$, then $512 \times 10 + 10 = 401,920 + 5,130 = 407,050$ parameters.
11. Four times the parameters, one layer fewer, and typically slightly worse accuracy on this task. Depth is usually the more efficient way to spend parameters, though not without limit.

■ 46.10.4 PLAIN – what is really happening inside

1. The **universal approximation theorem** is the famous result here, and it is almost always quoted wrongly. Here it is stated correctly.
2. A feedforward network with a single hidden layer, containing enough neurons, and a suitable non-polynomial activation, can approximate any continuous function on a closed bounded region, to any accuracy you like.
3. Read the conditions again. Continuous. Closed and bounded. Any accuracy you like, but “enough neurons” is not bounded.
4. Now the three things it does not say, which is where the misuse happens.
5. It does not say how many neurons. “Enough” can mean exponentially many in the number of inputs. The theorem is silent on the count.
6. It does not say you can find those weights. It proves such weights exist. It gives no procedure, and gradient descent may never reach them.
7. It does not say anything about new data. Approximating a function on the region you sampled says nothing about behaviour outside it.
8. So the theorem is an existence result. It tells you the model family is rich enough. It does not tell you the model is learnable, efficient or useful.
9. The practical answer to why we use depth at all: for many functions, a deep network needs exponentially fewer units than a shallow one to reach the same accuracy. Depth is about efficiency, not possibility.
10. That is not folklore. There are proven separation results, for example Telgarsky in 2016, exhibiting functions a deep network represents with few units that any shallow network needs exponentially many units to match.

■ 46.10.5 TECHNICAL – the engineer’s version

1. A fully connected layer with n inputs and m outputs has $n \times m$ weights and m biases, and costs $2nm$ floating-point operations per sample.
2. Layer terminology in code: `nn.Linear(in, out)` in PyTorch, `Dense(units)` in Keras, `nn.Dense(features)` in Flax. All the same object.
3. Depth in real architectures, with dates:

Model	Year	Depth
LeNet-5	1998	7 layers
AlexNet	2012	8 layers

Model	Year	Depth
VGG-19	2014	19 layers
GoogLeNet	2014	22 layers
ResNet-152	2015	152 layers
GPT-3	2020	96 blocks
Llama 3 70B	2024	80 blocks

- Depth beyond about 20 plain layers did not train at all before residual connections. ResNet, from Kaiming He and colleagues at Microsoft Research in 2015, adds the input of a block to its output, so the block only has to learn a correction. That single change took usable depth from about 20 to over 1,000 and is the most important architectural idea of the decade.
- Approximation theory: George Cybenko proved the sigmoid case in 1989. Kurt Hornik generalized it in 1991, showing that the multilayer structure, not the choice of activation, is what gives universality. Leshno and colleagues showed in 1993 that a non-polynomial activation is the precise condition.
- There are also dual results for width. Lu and colleagues showed in 2017 that networks of bounded width and unbounded depth are universal, with a width threshold around the input dimension plus four.
- Established fact: depth gives exponential representational efficiency for some function classes, and residual connections make great depth trainable.
- Active research: exactly why depth helps optimization, not just representation, and why over-parameterized networks generalize at all. There is no complete theory. Anyone claiming otherwise is overselling.

■ 46.10.6 WORDS - remember these

- Layer — a row of neurons computed together — an affine transform followed by an elementwise non-linearity.
- Hidden layer — a middle layer nothing outside ever sees — any layer that is neither the input nor the output.
- Width — how many neurons in one layer — the output dimension of that layer.
- Depth — how many layers deep the network is — the number of composed non-linear transformations.
- Universal approximation theorem — one wide layer can copy any smooth function — for any continuous f on a compact set and any epsilon, a sufficiently wide single-hidden-layer network exists within epsilon. An existence result only.
- Residual connection — adding a block's input to its output — a skip connection giving gradients a direct path, from ResNet in 2015.

46.11 The forward pass as matrix multiplication

■ 46.11.1 PLAIN - in simple words

- Running a network on an input is called the **forward pass**. Numbers go in at one end and come out the other.
- Doing it one neuron at a time, as in section 46.8, is correct but slow to describe and slower to compute.
- Every neuron in a layer does the same thing to the same inputs, with different weights. That is exactly what a matrix multiplication is.

4. So stack the weights of all the neurons in a layer as rows of a grid. Then one grid-times-list operation computes the whole layer at once.
5. That grid is the **weight matrix**. Its height is the number of neurons and its width is the number of inputs.
6. The whole forward pass becomes: multiply by a matrix, add a bias list, apply the bend. Repeat once per layer. That is all a network is.
7. Chapter 22 explained the matrix multiplication and the chip that does it. This is the same operation, unchanged, on bigger grids.
8. That is not a coincidence or an analogy. It is the same arithmetic, and it is the entire reason graphics hardware runs this field.

■ 46.11.2 PLAIN – a picture in your head

1. Imagine a spreadsheet where a whole column of results is computed by one formula dragged down.
2. You do not calculate each cell separately. You describe the pattern once and the machine fills the column.
3. A matrix multiplication is that idea in two directions at once, and on hardware built to do thousands of the multiplications simultaneously.

Where this comparison breaks: a spreadsheet computes cells one after another and just hides it from you. A graphics chip genuinely computes thousands at the same instant, in separate arithmetic units. The parallelism is real, not presentational, and that difference is worth roughly a thousandfold in speed.

■ 46.11.3 PLAIN – a worked example

1. A two-layer network. Two inputs, three hidden units with ReLU, one output.
2. Input $x = [1.0, 2.0]$.
3. Layer 1 weight matrix $W1$, three rows of two, and bias $b1$:

$$\begin{array}{|l} W1 = \begin{bmatrix} 0.5 & -0.2 \\ 0.3 & 0.8 \\ -0.4 & 0.1 \end{bmatrix} \quad b1 = \begin{bmatrix} 0.1 \\ -0.1 \\ 0.2 \end{bmatrix} \end{array}$$

4. Multiply row by row. Row 1: $(0.5 \times 1.0) + (-0.2 \times 2.0) = 0.5 - 0.4 = 0.1$. Add bias 0.1, giving 0.2.
5. Row 2: $(0.3 \times 1.0) + (0.8 \times 2.0) = 0.3 + 1.6 = 1.9$. Add bias -0.1, giving 1.8.
6. Row 3: $(-0.4 \times 1.0) + (0.1 \times 2.0) = -0.4 + 0.2 = -0.2$. Add bias 0.2, giving 0.0.
7. So $z1 = [0.2, 1.8, 0.0]$. Apply ReLU: nothing is negative, so $a1$ is unchanged at $[0.2, 1.8, 0.0]$.
8. Layer 2 is one row of three: $W2 = [1.0, -0.5, 0.25]$, and $b2 = [0.05]$.
9. Multiply: $(1.0 \times 0.2) + (-0.5 \times 1.8) + (0.25 \times 0.0) = 0.2 - 0.9 + 0 = -0.7$. Add bias: $-0.7 + 0.05 = -0.65$.
10. Output is -0.65. If this were a two-class problem, apply sigmoid: 0.343, so 34.3 percent.
11. Total parameters: $(3 \times 2 + 3) + (1 \times 3 + 1) = 9 + 4 = 13$. Total arithmetic: 9 multiply-adds. That is the whole network.
12. Written compactly, the entire forward pass is two lines:

$$\begin{array}{|l} a1 = \text{ReLU}(W1 @ x + b1) \\ out = \text{sigmoid}(W2 @ a1 + b2) \end{array}$$

■ 46.11.4 PLAIN – what is really happening inside

1. Now count the work for a real network: the 784 to 128 to 64 to 10 digit classifier from section 46.10.
2. Each weight is used exactly once per input, in one multiply and one add. So count multiply-accumulate operations, then double for floating-point operations.

Layer	Multiply-adds	FLOPs
784 → 128	100,352	200,704
128 → 64	8,192	16,384
64 → 10	640	1,280
Total	109,184	218,368

3. About 218,000 floating-point operations to classify one digit. A modern laptop core does billions per second, so this takes microseconds.
4. Useful rule to memorize: a forward pass costs roughly two floating-point operations per parameter. Here, 218,368 divided by 109,386 is 1.996.
5. That rule holds for dense layers and for transformers, so it lets you estimate the cost of any model you know the size of.
6. Now batching. Instead of one input vector, stack 64 inputs as columns of a matrix. One matrix-times-matrix computes all 64 forward passes.
7. Same total arithmetic, 64 times 218,368 which is about 14 million operations, but each weight is loaded from memory once and used 64 times.
8. Chapter 22 made the point that memory bandwidth, not arithmetic, is usually the limit. Batching is how you escape it, and it is why batch size is a performance knob and not only a statistical one.

■ 46.11.5 TECHNICAL – the engineer’s version

1. For a layer with weight matrix W of shape (m, n) , input X of shape (n, B) where B is the batch size, and bias b of shape $(m, 1)$: $Z = W X + b$, broadcast over the batch dimension, then $A = f(Z)$.
2. This is a GEMM, a general matrix-matrix multiply, the operation every BLAS library optimizes above all others. On NVIDIA hardware it runs through cuBLAS or CUTLASS, and on tensor cores where precision allows.
3. Cost: $2 m n B$ FLOPs. Memory traffic if unbatched and uncached: $m n$ reads for the weights per sample. The arithmetic intensity, operations per byte, rises linearly with B until the compute units saturate.
4. Convention warning: mathematics texts write $W x$ with x as a column vector. PyTorch’s `nn.Linear` stores weight of shape $(\text{out\_features}, \text{in\_features})$ but computes $x @ W.T + b$ with x of shape $(\text{batch}, \text{in\_features})$, so the batch dimension is first. Both conventions are in daily use and mixing them up is a common source of shape errors.
5. Real forward-pass costs, per single sample:

Model	Parameters	FLOPs per forward
MNIST MLP above	109 thousand	0.22 million
ResNet-50, 224px	25.6 million	about 8.2 billion
GPT-3, per token	175 billion	about 350 billion

6. Note ResNet-50 breaks the two-FLOPs-per-parameter rule badly, at about 320 FLOPs per parameter. That is because convolution reuses each weight at every spatial position. The rule applies to dense layers, and transformers are mostly dense layers, which is why it works there.

7. The training-cost rule of thumb, from Kaplan and colleagues in 2020, is about $6ND$ floating-point operations to train a model of N parameters on D tokens: roughly $2N$ for the forward pass and $4N$ for the backward pass.
8. Precision matters enormously here, and Chapter 22 gave the per-chip figures. The short summary without repeating them: halving the bits per number roughly doubles the achievable rate on the same silicon, which is why training moved from FP32 to mixed BF16 and inference is moving to FP8 and 4-bit integers.

■ 46.11.6 WORDS – remember these

1. Forward pass — running the input through to the output — evaluating the composed function, layer by layer, with no gradient computation.
2. Weight matrix — the grid holding a layer's weights — a matrix of shape (out_features, in_features).
3. GEMM — the matrix multiply that dominates the work — general matrix-matrix multiply, the core BLAS level-3 routine.
4. Batch dimension — many inputs processed at once — the axis over which samples are stacked to raise arithmetic intensity.
5. FLOP — one floating-point operation — one multiply or one add; a forward pass through dense layers costs about 2 per parameter.

46.12 Backpropagation

■ 46.12.1 PLAIN – in simple words

1. The network produced an answer. The answer was wrong by some amount. Now what?
2. Every weight in the network contributed something to that wrong answer. Some contributed a lot. Some contributed almost nothing. Some pushed in the right direction and were outvoted.
3. What we need is, for each weight, its share of the blame: if this one weight were slightly larger, would the error get better or worse, and by how much?
4. **Backpropagation** is the procedure that works out every weight's share.
5. It works backwards. It starts at the output, where the error is directly measurable, and moves towards the input, one layer at a time.
6. At the output layer, the blame is obvious. Each output weight fed directly into the answer, so its share is easy to compute.
7. Then the trick. Having computed how much blame belongs to each unit in the last hidden layer, that blame is passed further back, split among the weights that fed those units, in proportion to how strongly each fed them.
8. Repeat until you reach the input. Now every weight in the network has a number attached: its share of the error.
9. Then every weight is nudged a little, against its share. Weights that made things worse move down. Weights that helped move up.
10. That is it. Compute forwards, assign blame backwards, nudge everything. Millions of times.
11. The reason this matters so much is efficiency. It computes the share for every weight in one backward sweep, costing about twice one forward pass, regardless of how many weights there are.

■ 46.12.2 PLAIN – a picture in your head

1. A large company delivers a project three weeks late. The chief executive wants to know who is responsible.
2. She does not interview every one of ten thousand employees. That would take longer than the project.
3. Instead she asks four department heads: how much of the delay came through your department? They can answer, because they sit right next to the outcome.
4. Manufacturing says 60 percent, logistics 30 percent, design 10 percent.
5. Now each head asks their own team leads the same question, splitting only their own share. The manufacturing head splits her 60 percent among five teams.
6. Each team lead splits their portion among their engineers. In four rounds, every one of ten thousand employees has a number.
7. Four conversations per level, not ten thousand interviews. The saving is the whole point.
8. And the blame is proportional to influence. Someone whose work barely touched the delivery date receives a share near zero automatically.

Where this comparison breaks, and it matters. First, blame in a company is a judgement, and people argue. Here it is an exact derivative with a single correct value. Second, real blame implies fault. Here a “share” can be negative, which simply means that increasing that weight would reduce the error, so the weight should grow. Third, and most important, the shares are only valid for a tiny nudge. They tell you the effect of changing one weight by a hair while everything else stays fixed. Change all the weights at once by a large amount and the shares no longer describe reality. That is exactly why the learning rate must be small.

■ 46.12.3 PLAIN – a worked example

1. Now the chain rule, which is the exact rule behind the blame splitting.
2. If a affects b, and b affects c, then the effect of a on c is the effect of a on b multiplied by the effect of b on c.
3. If turning a dial doubles a pressure, and doubling the pressure triples a flow, then turning the dial multiplies the flow by six. Effects along a chain multiply.
4. A network is a long chain, so a weight’s effect on the loss is the product of the effects at every step between that weight and the loss.
5. Here is the network we will do completely. Two inputs, two hidden units with sigmoid, one output unit with sigmoid, and the loss is half the squared error.

```
x1=0.5 --w11=0.8--> h1 --v1=0.5--\
      \--w21=0.2-> h2 --v2=-0.3--> out -> loss
x2=-1.0-/w12=-0.4, w22=0.6/
biases: b1=0.1, b2=-0.2, c=0.2      target y = 1.0
```

6. Forward pass. Hidden unit 1: $z = (0.8 \times 0.5) + (-0.4 \times -1.0) + 0.1 = 0.4 + 0.4 + 0.1 = 0.9$. Sigmoid of 0.9 = 0.710950.
7. Hidden unit 2: $z = (0.2 \times 0.5) + (0.6 \times -1.0) + (-0.2) = 0.1 - 0.6 - 0.2 = -0.7$. Sigmoid of -0.7 = 0.331812.
8. Output: $z = (0.5 \times 0.710950) + (-0.3 \times 0.331812) + 0.2 = 0.355475 - 0.099544 + 0.2 = 0.455931$. Sigmoid = 0.612048.
9. Loss = $0.5 \times (0.612048 - 1.0)^2 = 0.5 \times 0.150506 = 0.075253$.
10. Backward pass, output layer first.
11. Effect of the output on the loss: predicted minus target = $0.612048 - 1.0 = -0.387952$.

12. Effect of the pre-activation on the output, which is the sigmoid derivative: $0.612048 \times (1 - 0.612048) = 0.237445$.
13. Multiply them. This product is the output unit's blame share, called delta: $-0.387952 \times 0.237445 = -0.092117$.
14. Now each output weight's share is that delta times the activation it multiplied:
15. Share for $v1 = -0.092117 \times 0.710950 = -0.065491$.
16. Share for $v2 = -0.092117 \times 0.331812 = -0.030566$.
17. Share for the output bias $c = -0.092117$, because a bias multiplies 1.
18. Now push the blame back to the hidden units. Each hidden unit's blame is the output delta times the weight connecting them, times that unit's own sigmoid derivative.
19. Hidden unit 1: $-0.092117 \times 0.5 = -0.046059$. Its sigmoid derivative is $0.710950 \times 0.289050 = 0.205500$. Product: -0.009465 .
20. Hidden unit 2: $-0.092117 \times -0.3 = +0.027635$. Its sigmoid derivative is $0.331812 \times 0.668188 = 0.221713$. Product: $+0.006127$.
21. Note that hidden unit 2 got a positive share while unit 1 got a negative one, because its outgoing weight was negative. The sign carried through the multiplication automatically.
22. Now the input weights. Each is its unit's delta times the input it multiplied:
23. $w11 = -0.009465 \times 0.5 = -0.004733$. $w12 = -0.009465 \times -1.0 = +0.009465$.
24. $w21 = +0.006127 \times 0.5 = +0.003064$. $w22 = +0.006127 \times -1.0 = -0.006127$.
25. Biases $b1 = -0.009465$ and $b2 = +0.006127$.
26. Every parameter now has a share. Update with a learning rate of 0.5, using new value = old value minus rate times share:

Parameter	Old	Share	New
$v1$	0.5	-0.065491	0.532745
$v2$	-0.3	-0.030566	-0.284717
c	0.2	-0.092117	0.246059
$w11$	0.8	-0.004733	0.802366
$w12$	-0.4	+0.009465	-0.404733
$w21$	0.2	+0.003064	0.198468
$w22$	0.6	-0.006127	0.603064

27. Run the forward pass again with the new numbers. Output is now 0.629935, up from 0.612048, and the loss is 0.068474, down from 0.075253.
28. It moved towards the target of 1.0. One step, and it worked.
29. Continue: after 5 steps the output is 0.688, after 20 steps 0.801, after 50 steps 0.874. Loss falls from 0.0753 to 0.0079.
30. That is training. Everything else is this, at scale.

■ 46.12.4 PLAIN – what is really happening inside

1. Notice something about the arithmetic above. The value 0.710950 was used twice: once going forward, and once coming back.
2. That is the reason the forward pass stores its intermediate values. Every activation computed on the way in is kept, because the backward pass needs it.
3. This is why training uses far more memory than inference. Inference can discard each layer's output as soon as the next layer has consumed it. Training cannot discard anything until the backward pass has been.

4. For large models this stored activation memory frequently exceeds the memory taken by the weights themselves.
5. The other thing to notice: at no point did anyone write down a derivative formula for the whole network.
6. Each small step knew only its own local rule. The sigmoid knows its own derivative. Multiplication knows its own. Addition knows its own.
7. The overall answer came from chaining those local rules together, mechanically.
8. That mechanical chaining is called **automatic differentiation**, and it is what PyTorch, JAX and TensorFlow actually implement. Backpropagation is one specific use of it.
9. Here is how the frameworks do it. As the forward pass runs, every operation records itself and its inputs onto a graph. Multiply here, add there, ReLU there. That is the **computation graph**.
10. When you ask for gradients, the framework walks that graph backwards. At each node it applies that operation's known local derivative rule and multiplies it into what arrived from downstream.
11. The graph is built while the code runs, so an `if` statement or a loop in your model creates a different graph on different inputs, and it still works. That style is called define-by-run, and PyTorch popularized it.
12. So you never write a gradient. You write the forward computation in ordinary code, call `.backward()`, and every parameter's share appears in its `.grad` field.

■ 46.12.5 TECHNICAL – the engineer's version

1. Notation for a network of L layers, with z the pre-activation, a the activation, and δ the error signal at a layer.
2. Output layer: $\delta_L = \text{grad}_a L$, elementwise times $f'(z_L)$. For the common pairing of softmax with cross-entropy, this simplifies exactly to predicted probability minus one-hot target, with no activation derivative left over.
3. Recurrence for earlier layers: $\delta_l = (W_{l+1})^T \delta_{l+1}$, elementwise times $f'(z_l)$.
4. Parameter gradients: $dL/dW_l = \delta_l a_{l-1}^T$, and $dL/db_l = \delta_l$. Summed over the batch.
5. Read those four lines again. The forward pass multiplies by W . The backward pass multiplies by W^T . That is the whole structural symmetry, and it is why backward costs about the same as forward.
6. Cost: the backward pass is about twice the forward, because each layer computes both the gradient with respect to its input and the gradient with respect to its weights. Total training step is about three times a forward pass, giving the $2N$ forward plus $4N$ backward that produces the $6ND$ rule.
7. Memory: activations for all L layers must be retained. Gradient checkpointing, from Chen and colleagues in 2016, stores only some layers and recomputes the rest during the backward pass, trading roughly 30 percent more compute for a large memory saving. It is standard for large models.
8. Automatic differentiation has two modes. Reverse mode, used here, costs one sweep per output and is efficient when outputs are few and inputs are many. Forward mode costs one sweep per input and is efficient in the opposite case. Training has one scalar loss and billions of parameters, so reverse mode wins by an enormous margin.
9. Reverse mode is Linnainmaa's 1970 result. Applying it to networks is Werbos in 1974 and Rumelhart, Hinton and Williams in 1986.
10. Backpropagation is not finite differences. Nudging each weight and re-running would cost one

forward pass per parameter, so 175 billion forward passes for GPT-3 instead of one backward pass. It is also not symbolic differentiation, which would produce an expression that grows explosively with depth. Automatic differentiation is a third thing: numerically exact, and linear in the size of the computation.

```
import torch
x = torch.tensor([0.5, -1.0])
W1 = torch.tensor([[0.8, -0.4], [0.2, 0.6]], requires_grad=True)
b1 = torch.tensor([0.1, -0.2], requires_grad=True)
h = torch.sigmoid(W1 @ x + b1)
v = torch.tensor([0.5, -0.3], requires_grad=True)
c = torch.tensor(0.2, requires_grad=True)
out = torch.sigmoid(v @ h + c)
loss = 0.5 * (out - 1.0) ** 2
loss.backward()
print(v.grad)      # tensor([-0.0655, -0.0306])
print(W1.grad)     # tensor([[ -0.0047,  0.0095], [ 0.0031, -0.0061]])
```

11. Those printed gradients are exactly the shares computed by hand above. The hand calculation and the framework agree to the digits shown.
12. Practical detail: gradients accumulate by default in PyTorch, so `optimizer.zero_grad()` must be called each step. Forgetting it is one of the most common bugs in the field, and it shows up as training that starts fine and then degrades.

■ 46.12.6 WORDS - remember these

1. Backpropagation — passing the blame backwards through the layers — reverse- mode automatic differentiation applied to a layered network.
2. Chain rule — effects along a chain multiply — the derivative of a composition is the product of the derivatives of its parts.
3. Delta / error signal — one unit's share of the blame — the partial derivative of the loss with respect to that unit's pre-activation.
4. Computation graph — the record of every operation performed — a directed acyclic graph of operations and tensors, built during the forward pass.
5. Automatic differentiation — chaining known local derivative rules mechanically — exact derivative evaluation at machine precision, linear in the cost of the original computation.
6. Gradient checkpointing — recomputing instead of storing — trading extra forward compute for reduced activation memory during backpropagation.

46.13 Training in practice

■ 46.13.1 PLAIN - in simple words

1. The theory so far says: start somewhere, compute gradients, step downhill. In practice, five extra things decide whether it works at all.
2. **Initialization:** what the weights are before training starts. They cannot all be zero, because then every neuron in a layer computes the same thing, gets the same gradient, and stays identical forever. Symmetry must be broken by randomness.
3. But the random numbers cannot be any size. Too large and each layer amplifies its input, so after twenty layers the numbers explode. Too small and each layer shrinks its input, so after twenty layers the signal is gone.
4. The fix is to scale the random numbers by the size of the layer, so each layer roughly preserves the scale of what passes through it.

5. **Input normalization:** rescale your input data so each feature has an average near zero and a similar spread. Section 46.5 showed this making the safe learning rate thirty-five times larger on a two-parameter problem.
6. **Normalization inside the network:** do the same rescaling to the values flowing between layers, not just at the entrance.
7. **Gradient clipping:** if the gradient is enormous, shrink it before taking the step, so one bad batch cannot destroy the model.
8. **Learning rate schedules:** do not use one step size for the whole run. Start small, grow, then shrink towards the end.
9. None of these change what the model can represent. All of them change whether you can find the weights.

■ 46.13.2 PLAIN – a picture in your head

1. Think of a chain of loudspeakers, each feeding the next one's microphone.
2. If every stage amplifies by 1.5, then after twenty stages the signal is 3,300 times louder and everything clips into noise.
3. If every stage attenuates by 0.7, after twenty stages the signal is one thousandth of its original size and is lost in the hiss.
4. What you want is every stage having a gain of almost exactly 1, so the signal arrives at the end intact.
5. Careful initialization is setting each amplifier's gain to about 1 at the start. Normalization layers are automatic gain control, resetting the level at each stage while the system runs.

Where this comparison breaks: an audio chain carries one signal and the right gain is obviously 1. A network carries hundreds of dimensions, and the useful quantity is the variance across them, not the level of one. Also, the gradient flows in the opposite direction to the signal, and it has its own gain that can explode or vanish independently. Initialization schemes have to compromise between keeping the forward pass and the backward pass both well scaled, and in general you cannot have both exactly.

■ 46.13.3 PLAIN – a worked example

1. Xavier initialization, from Glorot and Bengio in 2010: draw weights from a distribution with standard deviation equal to the square root of 2 divided by (inputs plus outputs).
2. He initialization, from He and colleagues in 2015: standard deviation equal to the square root of 2 divided by inputs. It is larger, to compensate for ReLU throwing away half the signal.

Layer shape	Xavier std	He std
784 -> 128	0.0468	0.0505
128 -> 64	0.1021	0.1250
512 -> 512	0.0442	0.0625

3. Rule: Xavier with tanh or sigmoid, He with ReLU and its relatives. Both ship in every framework and are usually the default already.
4. Gradient clipping by norm, worked. Suppose three parameters have gradients 3.0, 4.0 and 12.0.
5. The norm is the square root of $(9 + 16 + 144) = \text{square root of } 169 = 13.0$.
6. Clip threshold is 1.0. Since 13.0 exceeds 1.0, multiply every component by $1.0 / 13.0 = 0.07692$.
7. New gradients: 0.2308, 0.3077, 0.9231. Their norm is exactly 1.0.

8. The direction is unchanged. Only the length was cut. That is the point: you still step the right way, just not off a cliff.
9. A warmup-plus-cosine schedule, with peak rate $3e-4$, 2,000 warmup steps and 100,000 total steps:

Step	Learning rate
0	0
1,000	$1.5e-4$
2,000	$3.0e-4$ (peak)
10,000	$2.95e-4$
50,000	$1.55e-4$
100,000	0

10. Warmup exists because at step zero the weights are random, the gradients are large and meaningless, and a full-size step would throw the model somewhere useless before it has learned anything. Starting from zero and ramping up avoids that.

■ 46.13.4 PLAIN – what is really happening inside

1. The two normalization layers, and exactly what each normalizes. People confuse these constantly.
2. **Batch normalization**, from Ioffe and Szegedy in 2015: for each feature, compute the average and spread across all the samples in the batch, then rescale that feature so it has mean 0 and variance 1.
3. It normalizes down the batch. Sample 5's value for feature 3 depends on what samples 1 to 64 happened to be.
4. That has a strange consequence: a sample's output depends on which other samples it travelled with. At inference you have no batch, so batch norm keeps a running average from training and uses that instead. This train versus inference difference is a genuine source of bugs.
5. **Layer normalization**, from Ba, Kiros and Hinton in 2016: for each sample, compute the average and spread across that sample's own features, then rescale.
6. It normalizes across the features. Each sample is handled entirely alone, so the batch size does not matter and training and inference behave identically.
7. That independence is why layer normalization won in transformers and sequence models, where sequences have different lengths and batch statistics are unreliable. Batch normalization remains standard in convolutional image networks.
8. One sentence to remember: batch norm normalizes each feature over the batch; layer norm normalizes each sample over its features.
9. Both add two learned parameters per feature, a scale and a shift, so the network can undo the normalization where it is unhelpful.

■ 46.13.5 TECHNICAL – the engineer's version

1. Troubleshooting, which is the most useful table in this chapter:

Symptom	Likely cause	Action
Loss is NaN	learning rate too high	halve it, add clipping
Loss flat from step 0	rate too low, or dead net	raise rate 10x, check init
Loss falls then explodes	rate too high late	add warmup and decay
Train falls, val rises	overfitting	augment, regularize, stop
Both flat and high	underfitting	bigger model, more steps

Symptom	Likely cause	Action
Loss spikes then recovers	bad batch or outlier	clip gradients, shuffle
Val loss very noisy	val set too small	enlarge it, use EMA
Works on 1 batch, not all	data pipeline bug	overfit 1 batch first
Slower each epoch	memory leak	check retained graphs

2. The single best debugging habit: take one batch of eight samples and train until the loss reaches near zero. If a model cannot memorize eight examples, the bug is in your code, not your hyperparameters. This takes two minutes and saves days.
3. Schedules in common use: step decay, which multiplies by 0.1 at fixed epochs, the classic ImageNet recipe; cosine annealing, from Loshchilov and Hutter in 2016, now the default for transformers; and inverse square root, used in the original 2017 transformer paper with 4,000 warmup steps.
4. Typical warmup lengths for large language models are 1 to 2 percent of total steps, or a fixed 2,000 steps. This is a convention, not a standard.
5. Gradient clipping: clip by global norm, not per-parameter, so the update direction is preserved. A threshold of 1.0 is the near-universal default for transformer training.
6. Mixed precision is standard practice: keep a master copy of weights in FP32, compute in BF16 or FP16, and with FP16 scale the loss up before the backward pass to stop small gradients underflowing to zero. BF16 has the same exponent range as FP32 so it usually needs no loss scaling, which is why it is preferred on hardware that supports it.
7. Observation tools: TensorBoard or Weights and Biases for curves, `torch.cuda.max_memory_allocated()` for memory, `nvidia-smi` for utilization, and `torch.profiler` or Nsight Systems for finding the real bottleneck. Low GPU utilization almost always means the data loader, not the model.
8. Reproducibility caveat: seeding with `torch.manual_seed` is not sufficient for bit-identical runs, because some GPU kernels are non-deterministic by design. `torch.use_deterministic_algorithms(True)` forces determinism where possible, at some cost in speed.

■ 46.13.6 WORDS - remember these

1. Initialization — the random starting values of the weights — a distribution chosen to keep activation and gradient variance stable across depth.
2. Xavier / Glorot initialization — scaling by inputs plus outputs — variance $2/(\text{fan\_in} + \text{fan\_out})$, suited to tanh and sigmoid.
3. He initialization — scaling by inputs only, larger — variance $2/\text{fan\_in}$, compensating for ReLU zeroing half its input.
4. Batch normalization — rescaling each feature across the batch — normalizing over the batch dimension, with running statistics used at inference.
5. Layer normalization — rescaling each sample across its own features — normalizing over the feature dimension, identical in training and inference.
6. Gradient clipping — cutting the step length without changing direction — rescaling the gradient vector when its global norm exceeds a threshold.
7. Warmup — starting with tiny steps and ramping up — a linear increase of the learning rate over the first steps, before the main decay schedule.

46.14 The families of neural network

■ 46.14.1 PLAIN – in simple words

1. Everything so far described the **feedforward** network: layers in a line, every unit connected to every unit in the next layer. It is the default and the fallback.
2. Its weakness is that it treats every input as unrelated to every other. Move a picture one pixel to the right and it is a completely different input.
3. A **convolutional network** fixes this for images. Instead of one weight per pixel, it learns a small patch of weights and slides that same patch over the whole image.
4. So a pattern learned in one corner is recognized everywhere, and the number of weights collapses.
5. A **recurrent network** handles sequences, such as words or sound. It reads one item at a time and keeps a running summary of what it has read.
6. An **autoencoder** learns to squeeze data into a small representation and rebuild it. Useful for compression, denoising and finding structure.
7. A **generative adversarial network** trains two networks against each other: one makes fakes, one detects fakes, and both improve.
8. A **diffusion model** learns to remove noise, and generates by starting from pure noise and removing it repeatedly until an image appears.
9. A **graph neural network** works on data with connections, such as molecules or social networks, by passing messages along the edges.
10. The **transformer** replaced recurrent networks for almost all sequence work after 2017. Chapter 48 covers it properly.

■ 46.14.2 PLAIN – a picture in your head

1. Think of a rubber stamp and an ink pad.
2. A convolutional filter is a stamp with a small pattern cut into it. You press it at every position on the page and record how well the pattern matched there.
3. One stamp finds one thing: a vertical edge, a spot of red, a diagonal line.
4. A layer has many stamps, so you get many maps of where each pattern occurred.
5. The next layer stamps patterns onto those maps, so it finds combinations of patterns. A corner is where a vertical edge map and a horizontal edge map both light up.
6. Layer after layer, the patterns become larger and more specific.

Where this comparison breaks: a stamp is a fixed pattern you carved. These are learned from data, and typically nobody can say in words what most of them detect. The early ones do look like edge detectors and colour blobs, reliably and across independently trained networks, which is a genuine and repeatable finding. The middle ones are much harder to describe, and the neat story that each unit detects one nameable thing is often false.

■ 46.14.3 PLAIN – a worked example

1. One convolution, worked completely. A 5 by 5 image with a vertical edge in it: two dark columns then three bright ones.

image	kernel (vertical edge)		
0 0 10 10 10	-1	0	+1
0 0 10 10 10	-2	0	+2
0 0 10 10 10	-1	0	+1
0 0 10 10 10			
0 0 10 10 10			

2. The kernel is 3 by 3, so it fits in the image at $3 \times 3 = 9$ positions.
3. Take the top-left position. The window covers rows 1 to 3, columns 1 to 3: all values are 0, 0, 10 on each row.
4. Multiply each window value by the kernel value in the same place and add:
5. Row 1: $(0 \times -1) + (0 \times 0) + (10 \times +1) = 10$.
6. Row 2: $(0 \times -2) + (0 \times 0) + (10 \times +2) = 20$.
7. Row 3: $(0 \times -1) + (0 \times 0) + (10 \times +1) = 10$.
8. Total: $10 + 20 + 10 = 40$. That is one output value.
9. Slide right by one and repeat. Do this for all nine positions:

```
output (3x3)
 40 40 0
 40 40 0
 40 40 0
```

10. Read it. The left columns are strongly positive, marking the edge. The right column is exactly 0, because that window was uniformly bright, and a uniform region produces no edge response.
11. Nine parameters found a vertical edge anywhere in the image. A fully connected layer would have needed 25 weights per output unit and would have had to learn the pattern separately at every position.
12. Now **pooling**, which shrinks the map. Take a 4 by 4 grid and keep the largest value in each 2 by 2 block:

```
 1  3 | 2  4          max-pool 2x2, stride 2
 5  6 | 1  2  ---->    6  4
-----+-----        7  9
 7  2 | 8  0
 1  4 | 3  9
```

13. The result is half the size in each direction, so a quarter of the values. It keeps the strongest response and discards where exactly it occurred, which makes the network tolerant of small shifts.

■ 46.14.4 PLAIN - what is really happening inside

1. Recurrent networks and their fundamental limit, which is worth understanding properly because it explains why transformers exist.
2. A recurrent network reads word one, updates a hidden summary, reads word two, updates it again, and so on. The same weights are used at every step.
3. So the summary after 50 words has been through 50 multiplications by the same matrix.
4. If that matrix shrinks things slightly, the influence of word one is multiplied by a number below 1 fifty times and vanishes. If it grows things slightly, the influence explodes.
5. That is the vanishing and exploding gradient problem again, this time through time rather than through layers, and it is why plain recurrent networks cannot remember much past about ten steps.
6. The **LSTM**, from Hochreiter and Schmidhuber in 1997, fixes it with a separate memory line that information can travel along unchanged, plus three learned gates that decide what to write, what to keep and what to read. That memory path is additive rather than multiplicative, so gradients survive along it. LSTMs handle hundreds of steps.
7. But there is a second limit that no gate fixes: a recurrent network must process step 1 before step 2. It is sequential by construction.
8. That means it cannot use the parallel hardware from Chapter 22 across the sequence. Doubling your chips does not halve your training time.

9. The transformer's central trick is to let every position look at every other position at once, in one matrix multiplication, with no sequential dependency. That is why it took over. Chapter 48 explains how.
10. Established fact: transformers replaced recurrent networks for nearly all language work from about 2018 onwards. Active research: state-space models such as Mamba, from 2023, revisit the recurrent idea with a formulation that can be parallelized during training. Whether they displace attention at scale is genuinely open as of 2026.

■ 46.14.5 TECHNICAL – the engineer's version

1. Summary of the families:

Family	Best for	Key idea
Feedforward MLP	tabular, final heads	dense layers
Convolutional	images, spectrograms	shared local filters
Recurrent / LSTM	short sequences, streams	carried hidden state
Transformer	text, and now most things	all-pairs attention
Autoencoder	compression, anomalies	bottleneck plus rebuild
GAN	fast image synthesis	generator versus critic
Diffusion	high-quality generation	learned denoising
Graph network	molecules, networks	message passing on edges

2. Convolution parameter count: a layer with C_{in} input channels, C_{out} output channels and a k by k kernel has $C_{in} \times C_{out} \times k \times k$ weights plus C_{out} biases, independent of image size. A 3×3 layer with 64 in and 128 out has 73,728 weights and is applied at every spatial position.
3. Convolution FLOPs: $2 \times H_{out} \times W_{out} \times C_{in} \times C_{out} \times k \times k$. This is why ResNet-50 costs about 8.2 billion operations per 224 by 224 image with only 25.6 million parameters. Weights are reused, not multiplied.
4. Convolutional milestones: Fukushima's Neocognitron in 1980, LeCun's backpropagation-trained digit network at Bell Labs in 1989, LeNet-5 in 1998 reading cheques in production, AlexNet in 2012, VGG and GoogLeNet in 2014, ResNet in 2015.
5. What is actually implemented: most convolutional layers are lowered to a matrix multiply, either by the im2col transformation or by an implicit GEMM in cuDNN, so the underlying arithmetic is the same GEMM from Chapter 22. Winograd algorithms are used for small kernels to cut the multiply count.
6. Autoencoders: an encoder maps to a bottleneck of lower dimension, a decoder rebuilds, and the loss is reconstruction error. The variational autoencoder, from Kingma and Welling in 2013, adds a probabilistic latent distribution and enables generation. Autoencoder latents are used inside modern image diffusion models to work in a compressed space rather than pixel space.
7. GANs, from Ian Goodfellow and colleagues in 2014, train a generator and a discriminator in a minimax game. They produce images in a single forward pass, which is fast, but training is notoriously unstable and can collapse to producing a few outputs repeatedly. Diffusion models displaced them for quality from about 2021.
8. Diffusion models come from Sohl-Dickstein and colleagues in 2015, made practical by Ho, Jain and Abbeel's denoising diffusion probabilistic models in 2020. A forward process adds Gaussian noise in steps; a network learns to predict and remove it; generation runs the process backwards from noise. Sampling originally took hundreds of steps; distilled and flow-matching variants now reach single-digit steps. This area is moving fast.

9. Graph neural networks aggregate each node's neighbours' features, transform the result, and repeat. Message passing was formalized by Gilmer and colleagues in 2017. They are used in molecular property prediction, and AlphaFold 2 in 2021 combined graph-like reasoning with attention.

■ 46.14.6 WORDS - remember these

1. Convolution — sliding one small pattern over the whole image — a shared, local, translation-equivariant linear operator, implemented as a GEMM.
2. Kernel / filter — the small patch of learned weights — the k by k by C_{in} tensor slid across the input.
3. Pooling — shrinking the map by keeping the strongest response — a downsampling operation, usually 2×2 max or average with stride 2.
4. Feature hierarchy — edges first, objects later — successive layers representing progressively larger and more abstract patterns.
5. Recurrent network — reading a sequence while keeping a running summary — a network applying shared weights over time steps with a carried hidden state.
6. LSTM — a recurrent unit with a protected memory line — long short-term memory, with input, forget and output gates over an additive cell state.
7. Autoencoder — squeeze it small, then rebuild it — an encoder-decoder trained on reconstruction loss through a bottleneck.
8. Diffusion model — generating by repeatedly removing noise — a model trained to reverse a gradual noising process.

46.15 What a trained model physically is

■ 46.15.1 PLAIN - in simple words

1. After all the training, what do you actually have? A file.
2. The file contains numbers. Nothing else. No code, no logic, no rules you could read.
3. It is a long list of the weights and biases, in a known order, with a small index at the front saying which block of numbers belongs to which layer.
4. To use the model you need two things: this file, and the code that knows the architecture, so it can put each block of numbers in the right place.
5. Neither is useful alone. The file without the architecture is noise. The architecture without the file is an untrained network that outputs rubbish.
6. **Training** is the expensive process of producing the file. **Inference** is the cheap process of using it to answer one question.
7. Training happens once, on a cluster, for weeks. Inference happens billions of times, on one machine, in milliseconds.
8. The asymmetry between those two is the single most important economic fact about this field. Chapter 49 goes into what is inside the file in detail.

■ 46.15.2 PLAIN - a picture in your head

1. Think of a printed dictionary.
2. Compiling it took a team of lexicographers many years, with arguments, revisions and enormous cost.
3. Looking up one word takes you four seconds and costs nothing.
4. The book is the model file. The compilation is training. The lookup is inference.
5. And note: you cannot recover the arguments, the drafts or the sources from the finished book. It contains the conclusions and nothing else.

Where this comparison breaks: a dictionary can be read by a person and checked entry by entry. A model file cannot. There is no entry for anything. The knowledge, if that word even applies, is spread across all the numbers at once, and no single number means anything on its own. That property is called distributed representation, and it is why interpreting these models is hard research rather than an afternoon of reading.

■ 46.15.3 PLAIN – a worked example

1. File sizes, computed directly. Multiply the parameter count by the bytes per number.

Model	Parameters	Precision	File size
MNIST MLP	109,386	FP32	0.44 MB
ResNet-50	25.6 million	FP32	102 MB
Llama-class 8B	8 billion	BF16	16 GB
Llama-class 70B	70 billion	BF16	140 GB
Same 70B, 4-bit	70 billion	INT4	35 GB

2. The arithmetic is simply: FP32 is 4 bytes per number, BF16 and FP16 are 2, INT8 is 1, INT4 is half a byte.
3. That last row is why quantization matters. The same model drops from 140 GB to 35 GB, which is the difference between needing several datacentre cards and fitting on one.
4. Now the compute asymmetry, with the standard estimates. Training cost is about 6 times parameters times training tokens.
5. For an 8-billion-parameter model trained on 15 trillion tokens: $6 \times 8e9 \times 15e12 = 7.2e23$ floating-point operations.
6. Inference cost for one token is about 2 times parameters: $2 \times 8e9 = 1.6e10$ operations.
7. The ratio is $4.5e13$. Training that model costs about as much arithmetic as generating 45 trillion tokens with it.
8. So if the model serves a billion tokens a day, the training cost is repaid in arithmetic terms after about 45,000 days of service.
9. That is why nobody trains their own foundation model unless they are selling access to it, and why fine-tuning an existing file is the normal path.

■ 46.15.4 PLAIN – what is really happening inside

1. Inside the file, the layout is boring by design.
2. A header holds the metadata: the name of each tensor, its shape, its data type, and the byte range where its numbers start and end.
3. Then a single long region of raw numbers, laid out so a program can memory-map the file and read a tensor without copying it.
4. The safetensors format, now the default for published models, is exactly that: a JSON header giving names, shapes and offsets, then raw bytes.
5. It replaced Python's pickle format for a security reason. A pickle file can execute arbitrary code when loaded, so downloading a model was equivalent to running a stranger's program. safetensors cannot execute anything.
6. What is not in the file: the training data, the optimizer state, the code, or any record of how the numbers were reached.
7. The optimizer state is worth noting. During training, Adam keeps two extra numbers per parameter, so the training checkpoint is roughly three times the size of the final model, or four if a master FP32 copy is kept. Those extra numbers are discarded before publication.

8. Inference needs the weights and one small activation buffer. Training needs the weights, the gradients, the optimizer state, and every stored activation. That is why a model you can run on one card may need eight cards to fine-tune.

■ 46.15.5 TECHNICAL – the engineer’s version

1. Memory required, per billion parameters, as a planning table:

Purpose	Precision	Memory per 1B
Inference weights	INT4	0.5 GB
Inference weights	BF16	2 GB
Training weights	FP32 master	4 GB
Adam state	FP32, 2 moments	8 GB
Gradients	BF16	2 GB

2. So full fine-tuning at 1 billion parameters needs roughly 16 GB before activations, against 2 GB to run it. A factor of eight, and this is exactly why parameter-efficient methods such as LoRA, from Hu and colleagues in 2021, exist.
3. Formats you will meet: safetensors for published weights, ONNX for framework-portable graphs, GGUF for quantized local inference in the llama.cpp ecosystem, and TensorRT engines for compiled NVIDIA deployment.
4. Established fact: a trained model is a fixed function. Given identical inputs, identical settings and deterministic kernels, it returns identical outputs. Chat models appear to vary because a random sampler is applied to their output distribution, not because the model changed.
5. Established fact: the model does not learn from your conversation. Its weights are read-only at inference. Anything that looks like learning within a session is context, not weight change.
6. Marketing claim: that a deployed model “improves as you use it”. In almost all cases the file is static and improvements arrive as a new file, trained separately, on data that may include logged interactions. That is a different and much slower loop, and it is worth insisting on the distinction.

■ 46.15.6 WORDS – remember these

1. Model file / checkpoint — the file of trained numbers — a serialized tensor dictionary mapping parameter names to arrays.
2. Inference — using the trained model to answer — a forward pass only, with no gradients, activations discarded as soon as they are consumed.
3. Training — producing the numbers in the first place — repeated forward passes, backward passes and optimizer updates over a dataset.
4. Quantization — storing the numbers in fewer bits — reducing weight precision to INT8 or INT4 to cut memory and bandwidth, at some accuracy cost.
5. Safetensors — the safe modern weight file format — a JSON header of names, shapes and offsets over a raw tensor region, with no code execution.
6. Optimizer state — the extra numbers training needs — momentum and variance estimates, roughly two extra values per parameter for Adam.

46.98 Common wrong ideas

1. Wrong: the network understands what it is doing. Right: it computes a function. Weighted sums and bends, repeated. “Understanding” has no agreed technical definition here, and using the word imports assumptions nobody has earned. Whether internal structures deserve richer descriptions is active research, not a settled fact.
2. Wrong: more layers is always better. Right: past a point, deeper networks get harder to optimize, overfit more and cost more, without gaining accuracy. Plain networks beyond about 20 layers did not train at all until residual connections in 2015, and even now the right depth is chosen by measurement, not by adding more.
3. Wrong: it learned by itself. Right: a person chose the architecture, the loss, the optimizer, the learning rate, the schedule, the regularization, the data and when to stop. The only thing the machine did was solve, by repeated small steps, the exact optimization problem it was handed.
4. Wrong: neural networks work like brains. Right: the resemblance stopped at the 1943 model of a neuron. Real neurons spike, have timing, chemistry and internal state, and change their own structure. An artificial unit is a weighted sum and a bend. The name is historical, and today it is mostly marketing.
5. Wrong: AI is just a big pile of if-statements. Right: there are no branches at all in a dense network’s forward pass. It is a fixed sequence of multiplications, additions and one elementwise function, executed in the same order for every input. That is the opposite of branching.
6. Wrong: training finds the best possible weights. Right: it finds some set of weights where the gradient is near zero, reached from where it started. A different random start gives different weights and often similar accuracy. There is no guarantee of optimality for any non-convex model.
7. Wrong: a lower training loss means a better model. Right: training loss can always be driven down by memorizing. The only number that means anything is performance on data the model has never seen and that you never used to make a decision.
8. Wrong: the model stores its training data, so it can be asked to recall it. Right: the data shaped the parameters and was then discarded. Verbatim recall of some training examples does happen, and is a real privacy concern, but it is a side effect of over-fitting particular sequences, not a lookup mechanism.
9. Wrong: bigger datasets always help. Right: more of the same distribution has diminishing returns, and more badly labelled data actively hurts. Data quality, coverage and correct labelling usually beat raw volume.
10. Wrong: gradient descent gets stuck in local minima all the time. Right: in very high dimensions the common stationary points are saddles, not minima, because a minimum requires upward curvature in every direction at once. The practical obstacles are ill-conditioning, poor scaling and bad learning rates far more often than local minima.

46.99 Chapter summary in 20 lines

1. Machine learning writes a rule-shaped thing with adjustable numbers, and a procedure that tunes those numbers until outputs match examples.
2. It is curve fitting at heart, and saying so is accurate rather than dismissive; it is what lets you reason about the system.
3. The four pieces are always the same: a model, its parameters, a loss, and an optimizer that reduces the loss.
4. The history has two public failures. The perceptron of 1957 to 1958 was overclaimed, limited by Minsky and Papert in 1969, and funding collapsed.

5. Backpropagation existed from Linnainmaa in 1970 and Werbos in 1974 and was ignored until Rumelhart, Hinton and Williams published it in Nature in 1986.
6. In 2012 AlexNet won ImageNet with 15.3 percent top-5 error against 26.2 percent, trained on two consumer graphics cards, and the argument ended.
7. A parameter is a number the system may change; a hyperparameter is one you choose. Fitting $y = wx + b$ to five points has exactly two parameters.
8. Loss is one number saying how wrong you are. Squared error punishes large misses hardest; absolute error resists outliers. The choice changes the fit.
9. The loss surface is a landscape over the parameters, and training is walking downhill on it in fog, feeling only the slope under your feet.
10. Gradient descent steps against the gradient by a learning rate. Too small crawls; too large diverges to NaN. Normalizing inputs reshapes the ravine.
11. Momentum, RMSProp and Adam all exist to fix the same problem: plain descent zig-zags across narrow valleys instead of running along them.
12. Split your data into training, validation and test sets, and decide nothing using the test set. A widening gap between training and validation loss is overfitting, and the cure is early stopping, regularization or more data.
13. Classical methods are not obsolete. Gradient-boosted trees remain a leading choice for table-shaped data in 2026, though tabular foundation models are now competitive on small datasets and the boundary is moving.
14. One neuron is a weighted sum plus a bias, then a non-linear bend. It is not a brain cell and the analogy is historical.
15. Without the bend, stacked layers collapse algebraically into a single layer, so depth would buy exactly nothing. ReLU won because its slope is 1.
16. A forward pass is a matrix multiply, a bias add and a bend, once per layer. It is the same matrix multiply as Chapter 22, which is why GPUs run this field.
17. Backpropagation assigns each weight its share of the error by passing blame backwards layer by layer, then nudges every weight against its share.
18. That is the chain rule applied mechanically, which frameworks implement as reverse-mode automatic differentiation over a recorded computation graph.
19. Convolutional networks share small filters across an image, recurrent networks carry a running summary and cannot be parallelized over time, and the transformer of 2017 replaced them for sequences. See Chapter 48.
20. A trained model is a file of numbers, useless without the architecture that reads it. Training costs roughly $6 N D$ operations once; inference costs about $2 N$ per token, forever. See Chapter 49.

Parameters, Tokens and Embeddings

47.0 What this chapter gives you

1. You will be able to say exactly what one parameter is: a single number in a single position of a single table inside the model.
2. You will be able to take an architecture description and compute the parameter count by hand, and say which parts of the model hold the billions.
3. You will be able to tell parameters apart from hyperparameters, activations, context and tokens, and use all five words correctly in one sentence.
4. You will be able to turn a parameter count into a memory figure, for inference and for training, and explain the gap between them.
5. You will be able to explain quantization, name the common formats, and work out whether a given model fits on a given machine.
6. You will be able to say honestly why parameter count is a weak predictor of how useful a model is, and cite the evidence for that.
7. You will be able to explain byte-pair encoding by working it through by hand on a tiny corpus, and name the three other tokenizer families.
8. You will be able to predict roughly how many tokens a piece of text costs, and explain why Hindi costs several times more than English for one meaning.
9. You will be able to explain why a model cannot count the letters in a word, and why it fails at arithmetic on long numbers, from tokenization alone.
10. You will be able to explain what an embedding is, what similarity between embeddings means, and what a context window really limits.

47.1 What a parameter actually IS in a language model

■ 47.1.1 PLAIN - in simple words

1. A language model is, on the inside, a very large pile of numbers.
2. Nothing else. No sentences stored. No facts written down as text. Numbers.
3. One of those numbers is called a **parameter**.
4. Chapter 46 fitted a straight line to five points. That line had exactly two parameters: a slope and an intercept, both found by gradient descent, which is the process of nudging numbers downhill until the error stops falling.
5. A language model is the same idea with the count raised enormously. Two parameters becomes eight billion, and the shape of the thing being fitted becomes a network of layers instead of a line.
6. Everything else in this section is that one sentence unpacked.
7. So a parameter is not a setting you choose. It is not an option in a menu.

8. It is one plain number, like -0.0113 or 0.0392, sitting in one fixed slot.
9. The slots are arranged in rectangles, called matrices. Rows and columns.
10. A parameter has an address: this matrix, this row, this column.
11. When the model was trained, every one of those numbers was slowly adjusted until the model got good at predicting the next piece of text.
12. When training finished, the numbers were frozen and written to a file.
13. Every time you use the model, it reads the same frozen numbers.
14. Your question does not change them. Your conversation does not change them.
15. When somebody says a model has 7 billion parameters, they mean the file contains 7 billion such numbers.
16. That is the whole idea. A parameter is one learned number in one slot.

■ 47.1.2 PLAIN - a picture in your head

1. Picture an enormous sound mixing desk, the kind in a recording studio.
2. A mixing desk has knobs. Each knob controls how loudly one thing is heard.
3. Turn one knob and the guitar gets louder. Turn another and the drums fade.
4. Now imagine a desk with seven billion knobs instead of forty.
5. Each knob is set to some exact position, like 0.0392 out of a range.
6. Training was a very long process of nudging every knob a tiny amount, over and over, until the whole desk produced a good sound.
7. Shipping the model means writing down every knob position in a file.
8. Loading the model means setting all seven billion knobs back to those exact positions.
9. Running the model means pushing sound through the desk without touching a single knob.

Where this comparison breaks: on a real mixing desk, a human knows what each knob does, and there is a label under it. In a model, almost no single parameter has a meaning you can name. Its effect only exists in combination with millions of others. Also, a mixing desk has knobs a person turns during a performance. Model parameters are never turned during use. They are turned only during training, and then never again unless the model is retrained.

■ 47.1.3 PLAIN - a worked example

1. Here is a genuine kind of object from inside a model: a weight matrix.
2. This is a four-by-four corner of one, with values rounded for printing.

A 4 x 4 corner of one weight matrix, values rounded

	col 0	col 1	col 2	col 3
row 0	-0.0113	0.0271	-0.0004	0.0158
row 1	0.0392	-0.0087	0.0221	-0.0310
row 2	-0.0056	0.0143	0.0079	0.0002
row 3	0.0018	-0.0225	-0.0166	0.0407

3. Look at row 1, column 2. The value there is 0.0221.
4. That is a parameter. That one number. Not the row. Not the matrix. That.
5. Its full address inside a real model would be something like: layer 7, the query projection matrix, row 1, column 2.
6. There are 16 parameters in this little corner, because it is 4 rows times 4 columns.
7. The real matrix this corner came from would be 4096 by 4096.
8. That is $4096 \times 4096 = 16,777,216$ parameters in that one matrix alone.
9. A model of this size has hundreds of matrices like it.

10. Every one of those numbers started as a random value before training, and ended up at its final value because training pushed it there.
11. Nobody chose 0.0221. Nobody can explain 0.0221 on its own.

■ 47.1.4 PLAIN - what is really happening inside

1. The model's job at each step is to turn a list of numbers into another list of numbers.
2. The main operation it uses is multiply-and-add, done many times at once.
3. Take an incoming list of 4096 numbers. Call it the input vector.
4. Take one column of the weight matrix, also 4096 numbers.
5. Multiply them pairwise and add the results up. That gives one output number.
6. Repeat for every column. That gives the whole output vector.
7. So each parameter is a multiplier. It says how much one incoming number contributes to one outgoing number.
8. The parameter at row 1, column 2 says: take input number 1, multiply it by 0.0221, and add that into output number 2.
9. That is literally all a weight does. It is a multiplier in one cell of a giant multiply-and-add.
10. There is a second, smaller kind of parameter called a **bias**.
11. A bias is added rather than multiplied. It shifts an output up or down by a fixed amount regardless of the input.
12. Biases are also learned during training, and are also parameters.
13. Some model families use biases everywhere, some use almost none.
14. When the model runs, it walks through layer after layer doing this. The parameters never move. Only the numbers flowing through them change.

■ 47.1.5 TECHNICAL - the engineer's version

1. A **parameter** is one scalar element of a learnable tensor, updated by the optimizer during training and fixed at inference time.
2. In PyTorch terms, a parameter is one element of a tensor wrapped in `torch.nn.Parameter`, which is a tensor with `requires_grad=True` that is registered in the module's parameter list.
3. The words **weight** and **parameter** are used interchangeably in practice, and in almost every sentence you will read they mean the same thing.
4. The precise distinction is that a **weight** is a multiplicative parameter in a linear map, while a **bias** is an additive parameter. Both are parameters. So all weights are parameters, but not all parameters are weights.
5. Normalization scale and shift terms, often called gamma and beta, are also parameters. So are learned position embeddings where a model uses them.
6. In a decoder-only transformer of the Llama family, the linear layers carry no bias terms at all, and RMSNorm carries a scale but no shift. In GPT-2 the linear layers do carry biases and LayerNorm carries both scale and shift.
7. That difference is tiny in count. In GPT-2 small, all biases and all normalization parameters together are 121,344 out of 124,439,808 parameters, which is 0.098 per cent of the model.
8. Parameters are stored in the model's state dictionary under dotted names. Real names from a Llama-family checkpoint look like this:

```
model.embed_tokens.weight      [128256, 4096]
model.layers.0.self_attn.q_proj.weight [4096, 4096]
model.layers.0.self_attn.k_proj.weight [1024, 4096]
model.layers.0.mlp.gate_proj.weight [14336, 4096]
```

```
model.layers.0.input_layernorm.weight      [4096]
lm_head.weight                             [128256, 4096]
```

9. The bracketed pair is the tensor shape. The number of parameters in a tensor is the product of its dimensions.
10. Counting them in PyTorch is one line: `sum(p.numel() for p in model.parameters())`. Counting trainable ones only adds `if p.requires_grad`.
11. For a file on disk, safetensors stores a JSON header listing every tensor name, dtype and shape, so the count can be read without loading the model. For GGUF files, `gguf-dump` prints the same information.
12. Terminology note, and it is a standard one: in statistics a “parameter” is a quantity of a distribution, and in software a “parameter” is a function argument. Neither meaning applies here. The machine learning meaning is the one in this chapter.

Kind of parameter	Operation	Typical share
Linear layer weight	Multiply	Over 99 per cent
Bias	Add	Under 0.1 per cent
Norm scale (gamma)	Multiply	Under 0.01 per cent
Embedding row	Lookup	0.5 to 25 per cent

■ 47.1.6 WORDS – remember these

1. Parameter — one learned number in the model — one scalar element of a learnable tensor, updated by the optimizer, frozen at inference.
2. Weight — a number that multiplies something — a multiplicative parameter in a linear map, stored as an element of a weight matrix.
3. Bias — a number that is added on — an additive parameter applied after a linear map, one per output dimension.
4. Matrix — a rectangle of numbers — a rank-2 tensor with shape `[out_features, in_features]` for a linear layer.
5. Tensor — a box of numbers with a shape — an n-dimensional array with a dtype and a shape tuple.
6. State dictionary — the list of all the model’s numbers with names — an ordered mapping from parameter name to tensor, saved to disk.
7. Checkpoint — the saved file of a model — a serialized state dictionary, in formats such as safetensors, PyTorch .pt or GGUF.
8. Frozen — not changing any more — parameters with `requires_grad=False`, or any parameter during inference.

47.2 What 7B, 70B and 405B actually count

■ 47.2.1 PLAIN – in simple words

1. The B in 7B means billion. It is a count of parameters.
2. A 7B model contains about seven thousand million individual numbers.
3. A 405B model contains about four hundred and five thousand million of them.
4. Nothing else is being counted. Not files. Not facts. Not words known.
5. To get the total you add up the size of every matrix in the model.
6. Every matrix size is just rows times columns.

7. So the whole count is a long sum of simple multiplications.
8. The numbers are not round because the shapes are not round.
9. A model called “8B” often turns out to hold 8,030,261,248 parameters exactly.
10. The advertised name is rounded. The real count is exact and computable.
11. Once you know four or five shape numbers about a model, you can compute the total yourself in a few minutes.
12. That is what this section does, all the way through, for a real model.

■ 47.2.2 PLAIN – a picture in your head

1. Imagine a large office block with 126 identical floors.
2. Every floor has exactly the same furniture plan: seven big filing cabinets.
3. Some cabinets are wide, some are narrow, but every floor gets the same set.
4. To count the drawers in the whole building you do not walk every floor.
5. You count the drawers on one floor, then multiply by 126.
6. Then you add the reception desk at the bottom and the roof office at the top, because those exist only once.
7. That is exactly how a transformer parameter count works.
8. The floors are the layers. The identical furniture is the same set of matrices repeated in every layer.
9. The reception desk is the embedding table. The roof office is the output layer. Both exist once, not once per floor.

Where this comparison breaks: real buildings sometimes have odd floors. In most transformer models every layer genuinely is the same shape, which is why the multiply-by-number-of-layers trick works exactly. But some newer designs break this on purpose: mixture-of-experts models have layers with many parallel expert blocks, and only some are used per token, so the total count and the count actually used per token stop being the same number.

■ 47.2.3 PLAIN – a worked example

1. Let us count a real model completely: Llama 3.1 8B, released by Meta on 23 July 2024.
2. Five shape numbers describe it, and they are published:

Setting	Value
Layers	32
Hidden size (d)	4096
Feed-forward size	14336
Attention heads	32
Key-value heads	8
Vocabulary size	128256

3. Head size is hidden size divided by heads: $4096 / 32 = 128$.
4. Now count one layer. Attention first, four matrices:

```

q_proj  4096 x 4096  = 16,777,216
k_proj  4096 x 1024  =  4,194,304
v_proj  4096 x 1024  =  4,194,304
o_proj  4096 x 4096  = 16,777,216
attention total per layer = 41,943,040

```

5. The k and v matrices are narrower because of grouped-query attention: only 8 key-value heads, so $8 \times 128 = 1024$ columns instead of 4096.

6. Now the feed-forward part, three matrices:

```
gate_proj    4096 x 14336 = 58,720,256
up_proj      4096 x 14336 = 58,720,256
down_proj    14336 x 4096 = 58,720,256
feed-forward total per layer = 176,160,768
```

7. Two normalization vectors of 4096 each add 8,192 more.

8. One layer therefore holds $41,943,040 + 176,160,768 + 8,192 = 218,112,000$.

9. There are 32 layers: $32 \times 218,112,000 = 6,979,584,000$.

10. The embedding table is vocabulary times hidden size: $128,256 \times 4,096 = 525,336,576$.

11. The output layer is the same shape again: 525,336,576.

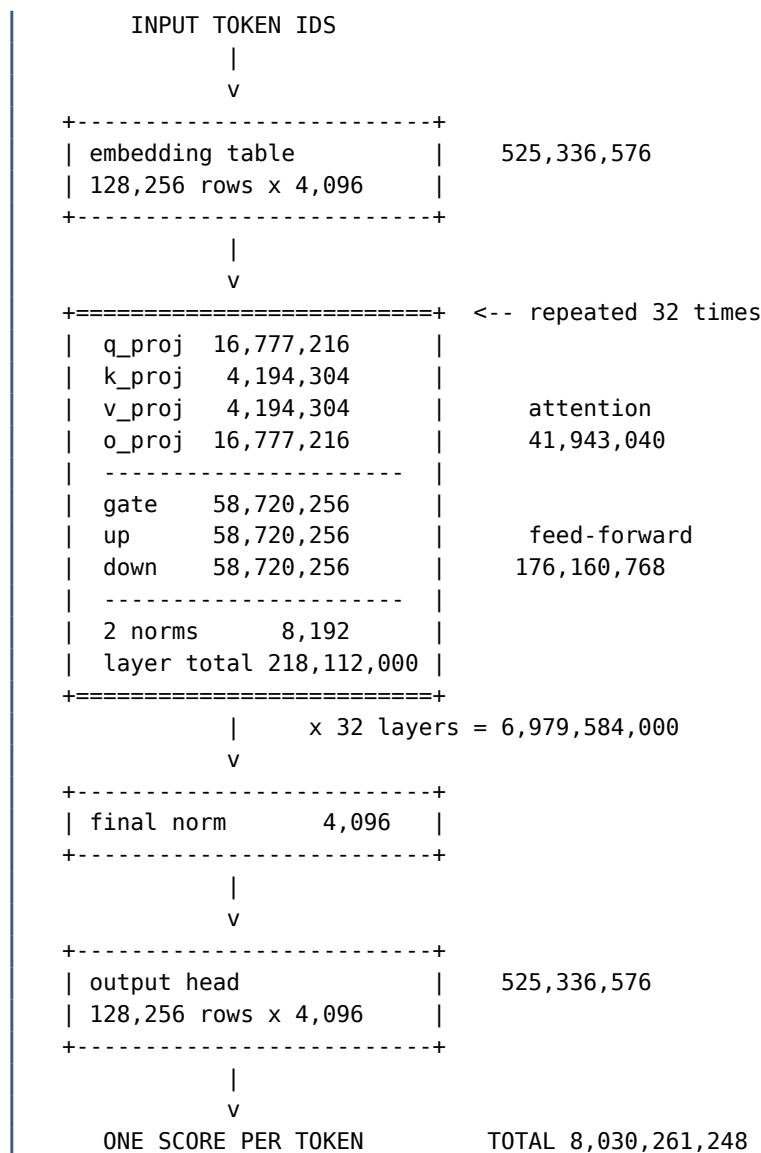
12. The final normalization adds 4,096.

13. Add them all: $6,979,584,000 + 525,336,576 + 525,336,576 + 4,096 = 8,030,261,248$.

14. That is 8.03 billion. The model is sold as “8B”. The arithmetic closes.

■ 47.2.4 PLAIN - what is really happening inside

1. Here is the whole 8B model drawn as a structure, with the parameter count written next to every piece.



2. Look at where that total came from and a clear pattern appears.
3. The feed-forward matrices are the biggest single item, by a wide margin.
4. In the 8B model, feed-forward is 70 per cent of all parameters.
5. Attention is about 17 per cent.
6. The embedding and output tables together are about 13 per cent.
7. Normalization is a rounding error: three thousandths of one per cent.
8. Now scale the model up and watch the shares move.
9. In the 70B model, feed-forward rises to 80 per cent and embeddings fall to 3 per cent, because the vocabulary stays the same size while the layers grow.
10. In the 405B model, feed-forward is 81 per cent and embeddings are 1 per cent.
11. So the honest one-line answer to “where do the billions live” is: in the feed-forward blocks, repeated once per layer.
12. That matters practically. When people compress models, they attack the feed-forward matrices first, because that is where the mass is.
13. It also explains why small models feel embedding-heavy. In GPT-2 small, the embedding table alone is 25 per cent of the whole model.

■ 47.2.5 TECHNICAL – the engineer’s version

1. The general formula for a decoder-only transformer with grouped-query attention, SwiGLU feed-forward and RMSNorm, with untied embeddings, is:

```

per_layer = 2*d*d          (q_proj and o_proj)
            + 2*d*n_kv*head_dim (k_proj and v_proj)
            + 3*d*d_ff      (gate, up, down)
            + 2*d          (two RMSNorm scales)

total = V*d                (input embedding)
       + L*per_layer       (all layers)
       + d                 (final RMSNorm)
       + V*d               (output head, if untied)

```

2. Here d is hidden size, d_{ff} feed-forward size, L layers, V vocabulary size, n_{kv} the number of key-value heads and $head\_dim$ the per-head width.
3. Applying it to the three published Llama 3.1 configurations gives exact totals that match the marketing names:

Model	Layers x d x d_{ff}	Computed total
Llama 3.1 8B	32 x 4096 x 14336	8,030,261,248
Llama 3.1 70B	80 x 8192 x 28672	70,553,706,496
Llama 3.1 405B	126 x 16384 x 53248	405,853,388,800

4. All three use vocabulary 128,256, head dimension 128, and 8 key-value heads. The 405B model has 128 query heads, which is why its k and v projections are only 16384×1024 while q and o are 16384×16384 .
5. Where the parameters sit, as a percentage of the total:

Block	8B	70B	405B
Feed-forward	70.20%	79.90%	81.25%
Attention	16.71%	17.12%	17.71%

Block	8B	70B	405B
Input embedding	6.54%	1.49%	0.52%
Output head	6.54%	1.49%	0.52%

- Norm parameters are 0.0033 per cent for 8B, 0.0019 per cent for 70B and 0.0010 per cent for 405B, and are omitted from the table above for width.
- Tied embeddings** are an implementation choice, not a standard: the output projection reuses the transposed input embedding matrix, saving $V \times d$ parameters. GPT-2 ties them. Llama 3.1 8B, 70B and 405B do not. Several small models such as Gemma tie them, which is why their embedding share is larger than you would guess.
- Verifying GPT-2 small the same way, including biases, gives exactly 124,439,808 parameters: 38,597,376 token embeddings, 786,432 learned position embeddings, 12 layers of 7,087,872 and a final LayerNorm of 1,536. The commonly quoted “117M” for GPT-2 small comes from an older count that excluded the position embeddings and some terms; 124M is the count that matches the released checkpoint.
- For mixture-of-experts models the count splits in two. Report the **total** parameter count and the **active** parameter count separately, because only a few experts run per token. A model with 8 experts of which 2 run has roughly a quarter of its feed-forward parameters active at any moment. Quoting only the total for such a model is a marketing choice, not a technical one.
- Tools: `sum(p.numel() for p in model.parameters())` in PyTorch, `model.num_parameters()` in Hugging Face transformers, the safetensors index JSON header, and `gguf-dump --no-tensors file.gguf` for GGUF metadata.

■ 47.2.6 WORDS - remember these

- Hidden size — how wide the model is — `d_model`, the width of the residual stream carried between layers, commonly 4096 or 8192.
- Feed-forward size — the width of the wide middle part — `d_ff`, usually between 2.7 and 4 times `d_model` in modern designs.
- Layer — one repeat of the model’s standard block — one transformer block containing attention, feed-forward and two normalizations.
- Vocabulary size — how many different tokens exist — the number of rows in the embedding matrix, for example 128,256 in Llama 3.1.
- Grouped-query attention — sharing key and value work across heads — GQA, where `n_kv` is smaller than `n_heads`, shrinking `k` and `v` projections and the key-value cache.
- Tied embeddings — reusing the input table as the output table — weight tying between `embed_tokens` and `lm_head`, saving $V \times d$ parameters.
- Active parameters — the ones that actually run for a token — in a mixture-of-experts model, the subset routed to per token, as against the total stored.

47.3 Parameters versus the four things they get confused with

■ 47.3.1 PLAIN - in simple words

- Five words get mixed up constantly. Here they are, one line each.
- Parameters** are the learned numbers inside the model, frozen after training.
- Hyperparameters** are the choices a human made before training started. They are not learned. Nobody trained them.

4. **Activations** are the temporary numbers created while the model runs on one input, and thrown away as soon as the answer is produced.
5. **Context** is the text you are giving the model right now. It is data.
6. **Tokens** are the small pieces that text is chopped into. Also data.
7. Now the sharp test. Which of these are in the model file on disk?
8. Only the parameters. The other four are not in the file.
9. Hyperparameters are recorded next to the file, in a small configuration note, so the loader knows what shape to build. But they were not learned.
10. Activations exist only for a moment while a request is being answered.
11. Context and tokens are whatever you happen to be asking about today.
12. So: parameters are what the model is. The rest is how it was built, what it is thinking about right now, and what you asked.

■ 47.3.2 PLAIN – a picture in your head

1. Think of a bakery.
2. The **recipe decisions** are the hyperparameters: oven temperature, tin size, how long to bake. A person chose these before any baking started.
3. The **baker's learned skill** is the parameters: the exact feel for how long to knead, built up over years and now fixed in the baker's hands.
4. The **ingredients on the counter today** are the context: this flour, this water, this order for a birthday cake.
5. The **individual scoops and spoonfuls** are the tokens: the units the ingredients get measured into.
6. The **half-made dough at each stage** is the activations: sticky, temporary, real, and gone the moment the bread is out.
7. Now the useful part. Changing the oven temperature is changing a hyperparameter, and you have to re-train the baker to suit it. Giving new ingredients is changing the context, and needs no re-training at all.

Where this comparison breaks: a baker learns a little from every loaf. A deployed model does not. Its parameters are identical after your millionth message and after your first. Any apparent learning inside a conversation is the context being carried along, not the parameters changing.

■ 47.3.3 PLAIN – a worked example

1. Here is one scenario with all five words used correctly in it.
2. A team trains a model. They choose 32 layers, hidden size 4096, learning rate 0.0003 and batch size 4 million tokens. Those are **hyperparameters**.
3. Training runs for weeks and produces 8,030,261,248 **parameters**, which are saved to a file of about 16 gigabytes.
4. A user types "Summarize this contract" and pastes 3,000 words of contract. That whole thing is the **context**.
5. The tokenizer chops that context into 4,182 **tokens**, each an integer.
6. The model runs. As those 4,182 tokens flow through 32 layers, it creates millions of temporary numbers at every layer. Those are **activations**, and they are discarded when the answer is finished.
7. Now the one-sentence version the reader asked for, all five in one line:
8. "The 8B model's frozen parameters were produced by a training run whose hyperparameters fixed 32 layers and a learning rate of 0.0003; at inference the 4,182 tokens of the user's context are turned into activations that flow through those parameters and are then thrown away."
9. Read it twice. Every one of the five words is doing a different job.

■ 47.3.4 PLAIN – what is really happening inside

1. Trace the lifetime of each of the five things, in order of when it exists.
2. Hyperparameters exist first, before there is a model at all. A human writes them into a configuration file.
3. The training program reads them and builds an empty model of that shape, filled with random numbers.
4. Training begins. The parameters change, millions of times, over weeks.
5. Training ends. The parameters are written to disk and never change again.
6. Later, somebody loads the file. Now the parameters are in memory.
7. A user sends text. That is the context. The tokenizer converts it to tokens.
8. The tokens flow in. At every layer the model computes new temporary numbers. Those are activations.
9. The activations are what carries the meaning of your specific question. The parameters are what does the transforming.
10. When the reply is finished, the activations are freed. The parameters are still sitting there, byte for byte identical.
11. This is why inference needs more memory than the model file size. The activations, and the cached keys and values for the conversation so far, need room too.
12. And it is why a longer conversation costs more memory but does not make the model bigger. The model is exactly as big as it was.

■ 47.3.5 TECHNICAL – the engineer's version

Thing	Learned?	Lives where
Parameter	Yes, by optimizer	Checkpoint, then GPU
Hyperparameter	No, chosen by human	config.json, run script
Activation	No, computed	Transient GPU buffers
Context	No, supplied	Request payload
Token	No, derived	Integer array

1. **Hyperparameters** split into two families that people wrongly lump together. Architectural hyperparameters fix the shape and therefore fix the parameter count: number of layers, hidden size, feed-forward size, head count, vocabulary size, maximum position count. Optimization hyperparameters do not change the count at all: learning rate, warmup steps, weight decay, batch size, dropout rate, gradient clipping threshold, number of epochs.
2. Sampling settings supplied per request, such as temperature, top-p, top-k and repetition penalty, are sometimes loosely called hyperparameters. They are inference settings. They change decoding, not the model.
3. **Activations** are the intermediate tensors of the forward pass. During inference with no gradient tracking, most are freed as soon as the next layer consumes them, so peak activation memory is a few hundred megabytes for a single sequence, not gigabytes.
4. During training the picture inverts. Activations must be retained for the backward pass. Using the accounting from the 2022 NVIDIA paper Reducing Activation Recomputation in Large Transformer Models, activation memory per layer per sample is about $sbh(34 + 5a*s/h)$ bytes in 16-bit, where s is sequence length, b batch, h hidden size and a head count.
5. For a 7B-class model at sequence length 2048 and batch 1 that is about 912 MiB per layer, so about 28.5 GiB across 32 layers for a single sample, before any recomputation is applied. Gradient checkpointing trades compute for memory to cut that sharply.

6. The **key-value cache** is a special, persistent kind of activation. It is the only activation deliberately kept across decoding steps, and it grows linearly with context length. It is covered with numbers in section 47.4.
7. **Context** is data on the request path. It has no gradient, no persistence and no effect on any stored parameter. This is the technical reason a model cannot “learn from your chat” unless a separate fine-tuning job is run.
8. **Tokens** are integers in the range 0 to V-1. They index rows of the embedding matrix. They are not parameters; the rows they select are.
9. Observing them: `config.json` and `generation_config.json` hold hyperparameters, `nvidia-smi` and `torch.cuda.max_memory_allocated()` show activation and cache growth, and any tokenizer’s `encode` returns the tokens.

■ 47.3.6 WORDS – remember these

1. Hyperparameter — a choice made before training — a configuration value not updated by gradient descent, either architectural or optimization-related.
2. Activation — a temporary number inside the model while it runs — an intermediate tensor of the forward pass, freed unless retained for backward.
3. Context — the text being processed right now — the input token sequence supplied per request, including any system prompt and history.
4. Key-value cache — the model’s notes on the conversation so far — cached per-layer key and value tensors reused across decoding steps.
5. Inference — using the model — a forward pass with no gradient computation and no parameter update.
6. Forward pass — pushing data through the model once — computing outputs layer by layer from inputs.
7. Gradient checkpointing — recomputing instead of storing — discarding activations during forward and recomputing them during backward to save memory at the cost of extra compute.

47.4 Parameters, memory and precision

■ 47.4.1 PLAIN – in simple words

1. Every parameter has to be stored somewhere, and storage takes space.
2. So the memory a model needs is a simple multiplication.
3. Memory = number of parameters x bytes used per parameter.
4. That is the whole rule. There is no magic in it.
5. How many bytes per parameter? That is your choice, and it is a trade.
6. Four bytes stores a number very accurately. Two bytes stores it less accurately but takes half the space.
7. One byte is less accurate still. Half a byte, which means four bits, is cruder again.
8. A 7 billion parameter model at two bytes each needs about 14 gigabytes.
9. The same model at half a byte each needs about 3.5 gigabytes.
10. Same model. Same parameters. Different amount of detail kept per number.
11. But the weights are not the only thing in memory when the model runs.
12. There is also the model’s running notes on your conversation, the temporary numbers, and the software’s own overhead.
13. And training is far worse again, because training keeps several extra copies of every parameter. We will work that out too.

■ 47.4.2 PLAIN – a picture in your head

1. Think of writing down the price of something.
2. You could write 12.345678. That is very precise and takes many characters.
3. You could write 12.35. Less precise, fewer characters.
4. You could write 12. Cruder still, fewest characters.
5. If you have to write down seven billion prices, the choice matters enormously for how much paper you need.
6. And for most purposes 12.35 is good enough. Nobody notices the missing digits when you are adding up a shopping basket.
7. That is exactly the precision trade in a model. Fewer digits per number, much less memory, slightly less exact answers.
8. Now the training part. To train, you do not only need the current prices.
9. You also need, for every price, a note of which direction to move it, and two running averages of how it has been moving lately.
10. So training needs about four to five sheets of paper for every one sheet that using the finished model needs.

Where this comparison breaks: numbers in a computer are not stored as decimal digits. They are stored in a floating point format with a sign, an exponent and a fraction, and the loss of accuracy is not a clean chopping of digits. It is a coarser grid of representable values, and the grid spacing changes with magnitude.

■ 47.4.3 PLAIN – a worked example

1. Take the same models at four different precisions. Here is the arithmetic for weights only, using 1 GiB = 1,073,741,824 bytes.
2. 7 billion × 4 bytes = 28,000,000,000 bytes = 26.1 GiB.
3. 7 billion × 2 bytes = 14,000,000,000 bytes = 13.0 GiB.
4. 7 billion × 1 byte = 7,000,000,000 bytes = 6.5 GiB.
5. 7 billion × 0.5 bytes = 3,500,000,000 bytes = 3.3 GiB.
6. Repeat for the other sizes and the full table falls out:

Model	fp32 GiB	fp16 GiB
7B	26.1	13.0
13B	48.4	24.2
70B (70.55B)	262.8	131.4
405B (405.85B)	1511.9	756.0

Model	int8 GiB	int4 GiB
7B	6.5	3.3
13B	12.1	6.1
70B (70.55B)	65.7	32.9
405B (405.85B)	378.0	189.0

7. Sanity check against reality: the published fp16 file for Llama 3.1 70B is about 141 GB in decimal gigabytes, which is 131 GiB. The arithmetic matches the real file.
8. Now add what inference needs beyond the weights.

9. The key-value cache for Llama 3.1 70B is 320 KiB per token. At 8,000 tokens that is 2.5 GiB. At 32,000 tokens it is 10 GiB.
10. Temporary activations for one sequence are roughly 0.5 to 1 GiB.
11. The framework's own overhead, buffers and fragmentation is another 1 to 2 GiB in practice.
12. So a realistic total for 70B at fp16 with an 8,000 token context is $131.4 + 2.5 + 1.0 + 2.0 =$ about 137 GiB, not 131.

■ 47.4.4 PLAIN – what is really happening inside

1. Training memory is a different story, and much bigger, for a simple reason.
2. Training must keep, for every single parameter, four extra things.
3. First, the parameter itself, usually kept twice: a fast low-precision copy used for the maths and an accurate master copy used for the update.
4. Second, the gradient, which is one number per parameter saying which way to nudge it.
5. Third and fourth, the optimizer's two running averages. The standard optimizer, called Adam, keeps a running average of recent gradients and a running average of recent squared gradients.
6. That is what "Adam's two moments" means: two extra numbers per parameter.
7. Count the bytes for the common mixed-precision recipe:

16-bit working weights	2 bytes
32-bit master weights	4 bytes
gradients	4 bytes
Adam first moment (m)	4 bytes
Adam second moment (v)	4 bytes

total	18 bytes per parameter

8. For a 7B model: $7,000,000,000 \times 18 = 126,000,000,000$ bytes = 117 GiB.
9. That is before a single activation is stored.
10. Add training activations, which for a 7B-class model at sequence length 2048 come to roughly 28 GiB per sample without recomputation.
11. So training a 7B model is a 150 GiB-plus problem, not a 14 GiB problem.
12. That is why a model you can happily run on one 24 GiB graphics card cannot be fully fine-tuned on it. Using needs weights. Training needs weights plus four more copies plus history.
13. It is also why techniques that train only a small extra set of parameters, which Chapter 50 covers, exist at all.

■ 47.4.5 TECHNICAL – the engineer's version

1. The precision formats, with their real layouts:

Format	Bits	Layout (sign/exp/frac)
fp32	32	1 / 8 / 23
fp16	16	1 / 5 / 10
bf16	16	1 / 8 / 7
fp8 E4M3	8	1 / 4 / 3

2. bfloat16 was introduced by Google for its Tensor Processing Units and keeps fp32's 8-bit exponent, so it has the same dynamic range as fp32 with far fewer fraction bits. That is why bf16 training rarely needs loss scaling while fp16 training usually does.
3. fp16 has a maximum finite value of 65504 and underflows below about $6e-8$, which is why gradients in fp16 are scaled up before the backward pass and scaled down afterwards.

- fp8 in the E4M3 and E5M2 variants was standardized across NVIDIA, Arm and Intel in a 2022 joint proposal, and is supported in hardware from NVIDIA's Hopper generation, announced March 2022. NVIDIA's Blackwell generation, announced 18 March 2024, added hardware support for 4-bit floating point.

- Key-value cache size per token is exact and worth memorizing:

```
| kv_bytes_per_token = 2 * L * n_kv * head_dim * bytes_per_element
```

- The leading 2 is for keys and values. Real figures at fp16:

Config	Per token	32k ctx	128k ctx
Llama 3.1 8B (GQA 8)	128 KiB	4.0 GiB	16.0 GiB
Llama 3.1 70B (GQA 8)	320 KiB	10.0 GiB	40.0 GiB
Llama 3.1 405B (GQA 8)	504 KiB	15.8 GiB	63.0 GiB
70B-shape without GQA	2560 KiB	80.0 GiB	320.0 GiB

- That last row is the point of grouped-query attention. Cutting key-value heads from 64 to 8 cuts the cache by a factor of 8, which is what makes long contexts affordable at all.
- Full-fine-tuning memory in bytes per parameter, by optimizer:

Optimizer state	Bytes/param	7B total
Adam, mixed precision	18	117 GiB
Adam, pure fp32	16	104 GiB
SGD with momentum	12	78 GiB
Inference, fp16	2	13 GiB

- Adam was published by Diederik Kingma and Jimmy Ba in 2014 as Adam: A Method for Stochastic Optimization; AdamW, the decoupled weight decay variant used by almost every modern language model, came from Ilya Loshchilov and Frank Hutter in 2017. Both keep two moment estimates per parameter, which is the source of the 8 bytes.
- ZeRO, from the 2019 Microsoft paper ZeRO: Memory Optimizations Toward Training Trillion Parameter Models, shards optimizer states, gradients and parameters across data-parallel ranks in three stages, so the per-device cost falls roughly by the number of devices. It does not reduce the total.
- Practical observation commands: `nvidia-smi --query-gpu=memory.used --format=csv`, `torch.cuda.memory_summary()`, and for serving, vLLM's reported KV cache blocks and `--gpu-memory-utilization` setting.

■ 47.4.6 WORDS - remember these

- Precision — how many digits are kept per number — the numeric format's bit width and its split between exponent and fraction.
- fp32 — the accurate, expensive format — IEEE 754 single precision, 4 bytes, 24 bits of significand including the implicit bit.
- bf16 — the training-friendly 16-bit format — brain floating point, fp32's exponent range with 8 bits of significand.
- Key-value cache — the model's stored notes on tokens seen so far — cached K and V tensors per layer, sized $2 \times L \times n_{kv} \times \text{head\_dim}$ per token.

5. Optimizer state — the extra bookkeeping training needs — Adam’s first and second moment estimates, one of each per parameter.
6. Mixed precision — computing in 16-bit, updating in 32-bit — a training recipe with a low-precision working copy and an fp32 master copy.
7. GiB versus GB — two different gigabytes — $2^{30} = 1,073,741,824$ bytes versus 10^9 bytes; file sizes are usually quoted in the decimal form and memory in the binary form, and confusing them costs you 7 per cent.

47.5 Quantization

■ 47.5.1 PLAIN - in simple words

1. **Quantization** means storing each of the model’s numbers using fewer bits.
2. That is the whole idea in one line. Fewer bits per number, smaller model.
3. The trained numbers are typically stored in 16 bits each.
4. Quantization rewrites them into 8 bits, or 4 bits, or sometimes fewer.
5. Nothing about the model’s shape changes. It still has 70 billion parameters.
6. Each of those 70 billion is just written down more roughly.
7. To do it you need a trick, because a 4-bit slot holds only 16 different values, and the real weights are spread over a wide range.
8. The trick is to store a small group of weights together with one shared scaling number.
9. The 4-bit value says “which of my 16 steps”, and the shared scale says “and here is how big a step is”.
10. Multiply them back together at load time or on the fly, and you get an approximation of the original number.
11. The approximation is not exact. The question is whether the error matters.
12. Mostly, at 8 bits and 5 bits, it barely does. At 4 bits it is small. Below that it starts to hurt in ways you can measure.

■ 47.5.2 PLAIN - a picture in your head

1. Imagine a shop that must label 10,000 items with prices.
2. The prices range from 1 rupee to 3,000 rupees.
3. The labels only have room for a single digit, 0 to 9.
4. That looks impossible, until you split the shop into departments.
5. In the pencil department, prices run 1 to 10 rupees, so you write on the department sign: “each step on the label = 1 rupee”.
6. In the furniture department, prices run 1,000 to 3,000, so that sign says “each step on the label = 200 rupees”.
7. Now a label reading 7 means 7 rupees in one department and 1,400 in the other, and one digit is enough everywhere.
8. The department sign is the **scale**. The department is the **group**.
9. Smaller departments mean more accurate prices but more signs to store.
10. That is the entire design space of quantization: how small the groups are, and how much extra you pay to store the signs.

Where this comparison breaks: real quantization also needs a **zero point** when the values are not centred on zero, so the label’s 0 does not have to mean a value of 0. And unlike a shop, a badly chosen department boundary does not just annoy a customer; it puts an outlier weight far outside the range and distorts every weight in the group with it.

■ 47.5.3 PLAIN – a worked example

1. Take eight real-looking weights and quantize them by hand.
2. The values: 0.42, -1.13, 0.05, 2.31, -0.77, 1.02, -2.90, 0.66.
3. Symmetric 8-bit first. Find the largest absolute value: 2.90.
4. Scale = $2.90 / 127 = 0.022835$.
5. Divide each value by the scale and round:

value	/scale	rounded int8
0.42	18.39	18
-1.13	-49.49	-49
0.05	2.19	2
2.31	101.16	101
-0.77	-33.72	-34
1.02	44.67	45
-2.90	-127.00	-127
0.66	28.90	29

6. To use them, multiply back by the scale. $18 \times 0.022835 = 0.4110$.
7. Original 0.42, recovered 0.4110. Error 0.0090.
8. Across all eight, the largest error is 0.0111. Small.
9. Now 4-bit, asymmetric, using minimum and maximum instead of absolute maximum.
10. Minimum -2.90, maximum 2.31. Scale = $(2.31 - (-2.90)) / 15 = 0.347333$.
11. Zero point = $\text{round}(2.90 / 0.347333) = 8$. So the stored code 8 means 0.
12. Quantized codes: 9, 5, 8, 15, 6, 11, 0, 10.
13. Recovered values: 0.3473, -1.0420, 0.0000, 2.4313, -0.6947, 1.0420, -2.7787, 0.6947.
14. Largest error now 0.1213, about eleven times worse than 8-bit. That is the price of four bits instead of eight.
15. Note the storage cost of the scale itself. If one 16-bit scale and one 16-bit zero point cover a group of 64 weights, the real cost is $4 + 32/64 = 4.5$ bits per weight, not 4.

■ 47.5.4 PLAIN – what is really happening inside

1. There are three levels of granularity, and they are the main design choice.
2. **Per-tensor**: one scale for an entire matrix. Cheapest to store, worst accuracy, because one outlier anywhere ruins the scale for everything.
3. **Per-channel**: one scale per row or per column. Much better, because a badly behaved output channel no longer poisons its neighbours.
4. **Per-group**: one scale for every fixed run of weights, typically 32, 64, 128 or 256 of them. Best accuracy, most scales to store.
5. Then there is the question of when you quantize.
6. **Post-training quantization** takes a finished model and compresses it. Fast, needs no training run, sometimes needs a few hundred sample texts to calibrate the scales.
7. **Quantization-aware training** simulates the rounding during training, so the model learns weights that survive rounding well. Much more expensive, better results at very low bit widths.
8. Finally: not everything gets squeezed equally.
9. The embedding table and the output layer are often kept at higher precision, because errors there hit every token directly and there is no later layer to absorb the damage.
10. Normalization scales, which are a tiny fraction of parameters, are almost always left at 16 or 32 bits.
11. Some formats keep specific matrices, such as the attention value projection and the feed-forward down projection, at a higher bit width because measurements showed those layers are the most

sensitive.

- And the key-value cache is a separate decision. You can run 4-bit weights with a 16-bit cache, or quantize the cache too.

■ 47.5.5 TECHNICAL - the engineer's version

- The affine quantization map, which is the standard, is:

```
q = clamp(round(x / s) + z, qmin, qmax)
x_hat = (q - z) * s
```

- Symmetric quantization fixes $z = 0$ and sets $s = \max|x| / q_{\max}$. Asymmetric quantization derives s from the range and z from the offset. Symmetric is cheaper at run time because the zero point drops out of the arithmetic.
- The named methods, with correct attribution:

Method	Year	One-line idea
LLM.int8()	2022	Split out outlier channels
GPTQ	2022	Second-order error correction
AWQ	2023	Protect salient weights
NF4 (QLoRA)	2023	Normal-distribution 4-bit

- LLM.int8(), by Tim Dettmers and colleagues in 2022, observed that a small number of feature dimensions carry extreme magnitudes and must stay in 16 bits, while the rest go to int8. This is mixed-precision decomposition.
- GPTQ, by Elias Frantar, Saleh Ashkboos, Torsten Hoefer and Dan Alistarh, posted October 2022 and published at ICLR 2023, quantizes weights column by column and uses approximate second-order information from a small calibration set to adjust the not-yet-quantized weights to compensate for the error just introduced.
- AWQ, Activation-aware Weight Quantization, by Ji Lin and colleagues at MIT, posted June 2023 and awarded Best Paper at MLSys 2024, observes that protecting roughly 1 per cent of weights, chosen by activation magnitude rather than weight magnitude, recovers most of the loss, and implements the protection by per-channel scaling rather than by keeping mixed precision.
- NF4, NormalFloat4, comes from the QLoRA paper by Tim Dettmers, Artidoro Pagnoni, Ari Holtzman and Luke Zettlemoyer, May 2023. Its 16 levels are placed at the quantiles of a normal distribution rather than evenly, because trained weights are approximately normally distributed. QLoRA also applies double quantization, quantizing the scales themselves, saving about 0.37 bits per parameter.
- GGUF is the file format used by llama.cpp, introduced on 21 August 2023 as the successor to GGML, GGUF and GGJT. Its k-quant types use a super-block of 256 weights subdivided into blocks of 32. Exact sizes, read from the ggml header:

Type	Bytes per 256	Bits per weight
Q2_K	84	2.625
Q3_K	110	3.4375
Q4_K	144	4.5
Q5_K	176	5.5
Q6_K	210	6.5625

9. The suffixes S and M mean small and medium mixes. Q4_K_M keeps the attention value projections and the feed-forward down projections at Q6_K in part of the network, so its measured average is about 4.8 bits per weight rather than 4.5. Checking against real published files: Llama 3.1 8B at Q4_K_M is 4.9 GB, and $8.03e9 \times 4.83 / 8 = 4.85e9$ bytes. The published 405B Q4_K_M file is 243 GB, and $405.85e9 \times 4.79 / 8 = 243e9$. The arithmetic closes both ways.
10. Quality cost, measured. From a January 2026 evaluation of llama.cpp quantization on Llama-3.1-8B-Instruct, WikiText-2 perplexity, lower better:

Format	Perplexity	Size cut
fp16 baseline	7.32	0%
Q8_0	7.33	46.9%
Q6_K	7.35	59.0%
Q5_K_M	7.40	64.4%
Q4_K_M	7.56	69.4%
Q3_K_M	7.96	75.0%
Q3_K_S	8.96	77.2%

11. Read that table honestly. Q8_0 costs 0.01 perplexity, which is noise. Q4_K_M costs 0.24, about 3.3 per cent relative, for a 69 per cent size cut. Q3_K_S costs 1.64, about 22 per cent relative, and that is a real degradation you will notice.
12. The same study reported that on aggregate downstream scores across GSM8K, HellaSwag, IFEval, MMLU and TruthfulQA, several quantized variants scored within noise of the fp16 baseline, and Q5_0 scored marginally above it. That last result is a reminder that benchmark noise at these margins is larger than the quantization effect, not evidence that quantizing helps.
13. Now the laptop arithmetic the question asks for. Assume about 75 per cent of installed memory is usable for the model, the rest going to the operating system and everything else.

Machine with 64 GiB, about 48 GiB usable:

```
70B at fp16  131.4 GiB weights -> does not fit
70B at int8   65.7 GiB weights -> does not fit
70B at Q4_K_M 39.7 GiB weights
              + 2.5 GiB KV at 8k
              + 2.5 GiB other   -> 44.7 GiB, fits
```

Machine with 16 GiB, about 12 GiB usable:

```
8B at fp16   15.0 GiB weights -> does not fit
8B at int8    7.5 GiB weights
              + 1.0 GiB KV at 8k
              + 1.3 GiB other   -> 9.8 GiB, fits
8B at Q4_K_M  4.5 GiB weights -> 6.8 GiB total, fits easily
70B at Q4_K_M 39.7 GiB weights -> does not fit
```

14. So the honest answer to “why can a 70B model run on a laptop at 4-bit but not at 16-bit” is: 131.4 GiB does not fit in 48 GiB and 39.7 GiB does, and that is the entire reason. Note the 64 GiB case only works with a modest context. Raise the context to 32,000 tokens and the cache alone adds 10 GiB, taking the total to about 52 GiB, which no longer fits.
15. On Apple silicon the calculation uses unified memory, and the default cap on how much the GPU may claim is around 65 to 75 per cent of installed RAM depending on machine and macOS version, which is where the 75 per cent rule of thumb above comes from.

16. Speed, separately from memory: at batch size 1, generation is memory bandwidth bound, so halving the bytes per weight roughly doubles tokens per second. That is often a bigger practical win than the memory saving.
17. Tools: llama-quantize in llama.cpp, AutoGPTQ and GPTQModel, autoawq, bitsandbytes for NF4 and int8, llm-compressor for vLLM-targeted formats, and optimum-quanto in the Hugging Face stack.

■ 47.5.6 WORDS - remember these

1. Quantization — storing each number in fewer bits — mapping high-precision weights onto a small integer grid with a scale and optional zero point.
2. Scale — how big one step of the integer grid is — the multiplier s in $\hat{x} = (q - z) * s$.
3. Zero point — where the integer grid's zero sits — the offset z that lets an asymmetric range be represented.
4. Group size — how many weights share one scale — commonly 32, 64 or 128; smaller means more accurate and more metadata.
5. Post-training quantization — squeezing a finished model — PTQ, applied to trained weights with at most a small calibration set.
6. Quantization-aware training — training with the rounding simulated — QAT, with fake quantization in the forward pass and straight-through gradients.
7. Perplexity — how surprised the model is by real text — the exponential of the mean negative log-likelihood per token; lower is better, and only comparable between models sharing a tokenizer.
8. Outlier channel — a feature dimension with extreme values — a dimension whose magnitude forces a bad scale for everything grouped with it.

47.6 Why capability does not scale linearly with parameter count

■ 47.6.1 PLAIN - in simple words

1. Bigger models are, on average, better. That much is true.
2. But parameter count is a weak way to guess how good a model is.
3. A model is the product of at least five things, not one.
4. How many parameters it has. How much text it was trained on. How good that text was. How much computing time was spent. What was done to it afterwards.
5. Get any one of those badly wrong and a huge model performs poorly.
6. Get them all right and a small model can beat a much larger one.
7. This is not a rare exception. It happens constantly.
8. In 2022 a 70 billion parameter model called Chinchilla beat a 280 billion parameter model called Gopher, using the same computing budget.
9. It won by being smaller and reading four times as much text.
10. That result changed how the whole field trains models.
11. So when a company says its model has more parameters than a rival's, that is a fact about the file size, not a claim about usefulness that you should accept without evidence.
12. The honest position: parameter count sets a ceiling. It does not tell you how close to that ceiling the model got.

■ 47.6.2 PLAIN - a picture in your head

1. Think of a student preparing for an exam.
2. Parameter count is how much the student could remember if they studied. Call it brain capacity.
3. Training data is how many books they read.

4. Data quality is whether the books were good ones or nonsense from the internet.
5. Training compute is how many hours they actually spent studying.
6. The stages after training are the coaching: practice papers, feedback on answers, learning how to lay out a solution clearly.
7. Now: a student with enormous capacity who read three books and never practised will fail.
8. A student with ordinary capacity who read widely, studied hard and got good coaching will do well.
9. Everyone knows this about students. The same thing is true about models, and the industry spent years talking as though only capacity mattered.

Where this comparison breaks: a student's capacity is fixed and unknown; a model's is chosen and exactly known. And a model has no motivation, no sleep and no forgetting, so the analogy tells you nothing about the training dynamics. It only tells you that one input out of five does not determine the output.

■ 47.6.3 PLAIN – a worked example

1. Take the Chinchilla experiment, published by Jordan Hoffmann and colleagues at DeepMind in March 2022, in Training Compute-Optimal Large Language Models.
2. Gopher: 280 billion parameters, trained on 300 billion tokens.
3. Chinchilla: 70 billion parameters, trained on 1.4 trillion tokens.
4. Both used about the same total computing budget. That is the key control.
5. Chinchilla is four times smaller and read four and a half times more.
6. On MMLU, a broad multiple-choice benchmark, Chinchilla scored 67.5 per cent, more than 7 percentage points above Gopher.
7. It also beat GPT-3 at 175 billion parameters, Jurassic-1 at 178 billion and Megatron-Turing NLG at 530 billion.
8. A 70B model beat a 530B model. That is a factor of seven and a half in parameter count, going the wrong way for the bigger model.
9. The paper's rule of thumb: for a fixed compute budget, model size and training tokens should grow together, roughly 20 tokens per parameter.
10. By that rule, GPT-3 at 175B should have been trained on about 3.5 trillion tokens. It was trained on 300 billion. It was undertrained by roughly a factor of ten.
11. So the entire generation of models before 2022 was too big for the amount of text it had read.

■ 47.6.4 PLAIN – what is really happening inside

1. Training reduces a single measured quantity: the average surprise the model feels when it sees real text.
2. That quantity falls smoothly and predictably as you add parameters, add data or add compute. Those smooth curves are called scaling laws.
3. A scaling law is a fitted curve, not a law of nature. It describes what has been measured over the ranges measured.
4. The curves are also very shallow. To halve the error you need far more than double the size.
5. And low surprise on text is not the same thing as being useful to a person.
6. That gap is closed by the stages after pre-training: supervised fine-tuning on demonstrations, and preference training on human comparisons.
7. Those stages change very few things about the model's knowledge and a great deal about its behaviour.

8. A raw pre-trained model of 70 billion parameters is often unusable as an assistant. The same model after post-training is helpful. Same parameters, different usefulness.
9. Data quality does something similar. Training on carefully filtered, deduplicated, high-quality text produces much better models per parameter than training on raw web scrape.
10. So the causal chain from parameter count to usefulness runs through at least four other factors, each of which can dominate.

■ 47.6.5 TECHNICAL - the engineer's version

1. Separate the three kinds of claim, as the reader asked.
2. **Established fact.** Loss falls as a power law in parameters, data and compute over several orders of magnitude. Kaplan and colleagues at OpenAI established this in January 2020 in Scaling Laws for Neural Language Models.
3. **Established fact.** Kaplan's recommended allocation was corrected by Hoffmann and colleagues in March 2022. Under a fixed compute budget, the compute-optimal ratio is roughly 20 training tokens per parameter, and pre-2022 models were badly undertrained.
4. **Established fact.** The Chinchilla scaling coefficients were partially disputed. Epoch AI published a replication attempt in 2024 finding that the paper's third estimation approach had fitting problems, though the broad conclusion, that data and parameters should scale together, survived.
5. **Established fact, and important.** Compute-optimal is not deployment-optimal. If a model will serve billions of requests, it is rational to train a smaller model far past its Chinchilla point, because inference cost is paid forever and training cost is paid once. Llama 3 8B was trained on about 15 trillion tokens, which is roughly 1,875 tokens per parameter, nearly 100 times the Chinchilla ratio, and it is a much better 8B model for it.
6. **Established fact.** Data curation matters enormously per parameter. Microsoft's phi line, starting with the June 2023 paper Textbooks Are All You Need, showed that filtered and synthetic textbook-style data lets very small models reach scores that previously needed models many times larger, though critics noted benchmark contamination risk in such setups.
7. **Established fact.** Mistral 7B, released 27 September 2023, outperformed Llama 2 13B on all benchmarks its authors reported. A 7B beat a 13B. Neither architecture nor data was held constant, which is exactly the point: parameter count alone did not decide it.
8. **Active research.** How much of frontier capability comes from post-training as against pre-training scale; whether reasoning-focused post-training and test-time compute change the scaling picture; how to measure data quality before spending the compute.
9. **Marketing claim.** Any headline of the form "our model has N parameters, therefore it is better". Also any comparison that quotes total parameters for a mixture-of-experts model against dense parameters for a rival without saying so.
10. **Where experts disagree.** Some argue continued scaling of pre-training is still the main driver of progress; others argue returns from pre-training scale have flattened and that data quality, post-training and inference-time compute now dominate. Both camps include serious researchers, and the public evidence does not settle it, partly because frontier training details are no longer published.
11. Practical consequence for an engineer: never select a model by parameter count. Select by measured performance on your own task, then by cost and latency. Parameter count is only useful for predicting memory and speed, which it predicts extremely well.

Model	Params	Tokens trained	Ratio
GPT-3 (2020)	175B	300B	1.7
Gopher (2021)	280B	300B	1.1
Chinchilla (2022)	70B	1.4T	20
Llama 3 8B (2024)	8B	~15T	~1875

■ 47.6.6 WORDS - remember these

1. Scaling law — a curve saying how much better a bigger model gets — a fitted power law relating loss to parameters, data or compute.
2. Compute-optimal — the best size for a fixed training budget — the parameter-and-token split minimizing loss at fixed FLOPs, about 20 tokens per parameter per Chinchilla.
3. Undertrained — too big for the text it read — a model whose token count is far below its compute-optimal allocation.
4. Overtrained (deliberately) — trained far past compute-optimal on purpose — a smaller model given many more tokens to reduce lifetime inference cost.
5. Pre-training — the long first phase — self-supervised next-token prediction on a very large corpus.
6. Post-training — everything done afterwards — supervised fine-tuning, preference optimization and safety training, covered in Chapter 50.
7. MMLU — a broad general-knowledge exam for models — Massive Multitask Language Understanding, 57 subjects of multiple-choice questions, released 2020.

47.7 Tokens: why text is not fed in as words or characters

■ 47.7.1 PLAIN - in simple words

1. A model works on numbers, so text must become numbers first.
2. The obvious plan is: one number per word. It fails, for four reasons.
3. First, there is no fixed list of all words. New ones appear every day.
4. Second, typos. “recieve” is not in any word list, so it becomes an unknown.
5. Third, many languages do not put spaces between words at all.
6. Fourth, a word list big enough to be useful is enormous, and the model would need one row of numbers per word, which is very expensive.
7. The opposite plan is: one number per letter. That fails too.
8. A page of text becomes thousands of letters, so sequences get very long, and long sequences are expensive.
9. Worse, the model must then learn spelling from scratch before it can learn anything about meaning.
10. So neither extreme works. The answer is in between: chop text into pieces that are usually bigger than a letter and often smaller than a word.
11. Those pieces are called **tokens** or **subwords**.
12. Common words become one token. Rare words become several pieces.
13. A word nobody has ever seen still gets encoded, because in the worst case it falls back to very small pieces.
14. Nothing is ever unknown. That is the property that makes it work.

■ 47.7.2 PLAIN – a picture in your head

1. Think about how a printer's typeset used to work.
2. One option: cut a separate metal block for every word in the language. Millions of blocks, and a new one every time somebody coins a word.
3. Another option: keep only 26 letter blocks and build everything letter by letter. Very few blocks, but setting one page takes forever.
4. What real printers did was in between. They kept letters, and they also kept **ligatures**: single blocks for common combinations like "fi" and "ffl".
5. Frequent combinations got their own block because it saved work.
6. Rare combinations were built up from letters.
7. That is exactly what subword tokenization does, chosen automatically by counting which combinations are frequent in a big pile of text.

Where this comparison breaks: a printer's ligatures were chosen by human craftsmen for how they looked. A tokenizer's pieces are chosen by a counting algorithm for how often they occur, and many of them look strange to a human eye. There is also no requirement that a token be a meaningful unit. Tokens regularly cut straight through the middle of a morpheme.

■ 47.7.3 PLAIN – a worked example

1. Here is byte-pair encoding worked completely, on a tiny corpus.
2. The corpus is four words with counts: low appears 5 times, lower 2 times, newest 6 times, widest 3 times.
3. Start by splitting every word into characters, with an underscore marking the end of a word.
4. The starting symbol set is 11 symbols: _ d e i l n o r s t w.
5. Now count every adjacent pair, weighted by word frequency, and merge the most frequent pair. Repeat.

```

step 1: top pairs  es=9  st=9  t_=9  we=8  lo=7  ow=7
           merge e+s -> es
           l o w _      (5)
           l o w e r _  (2)
           n e w e s t _ (6)
           w i d e s t _ (3)

step 2: top pairs  est=9  t_=9  lo=7  ow=7  ne=6  ew=6
           merge es+t -> est
           n e w e s t _ (6)
           w i d e s t _ (3)

step 3: merge est+_ -> est_      (count 9)
           n e w e s t _ (6)
           w i d e s t _ (3)

step 4: merge l+o -> lo          (count 7)
step 5: merge lo+w -> low        (count 7)
           l o w _      (5)
           l o w e r _  (2)

step 6: merge n+e -> ne          (count 6)
step 7: merge ne+w -> new        (count 6)
step 8: merge new+est_ -> newest_ (count 6)
step 9: merge low+_ -> low_      (count 5)
step 10: merge w+i -> wi         (count 3)

```

6. After 10 merges the words look like this: low_ (5), low e r _ (2), newest_ (6), wi d est_ (3).
7. The learned merge list, in order, is: es, est, est_, lo, low, ne, new, newest_, low_, wi.
8. The vocabulary is now 11 starting symbols + 10 merges = 21 tokens.
9. Notice what happened. “newest” became a single token because it was frequent.
10. “lower” is still in pieces, because it appeared only twice.
11. “est_” became a token that means, roughly, the superlative ending. Nobody told the algorithm about grammar. It fell out of counting.
12. Real training does this 30,000 to 200,000 times instead of 10 times, on billions of words instead of 16.

■ 47.7.4 PLAIN – what is really happening inside

1. Training a tokenizer and using one are two different programs.
2. Training: count pairs, merge the most frequent, repeat until the vocabulary reaches the target size. Save the ordered merge list and the vocabulary.
3. Using: split the incoming text into characters, then apply the saved merges in exactly the order they were learned. Same order, every time.
4. That ordering is what makes the process deterministic. The same text always gives the same tokens.
5. Then each final piece is looked up in the vocabulary to get its integer id.
6. A crucial detail: modern tokenizers do not start from characters. They start from bytes.
7. There are exactly 256 possible byte values, so every possible input, in any language, in any encoding, including broken text, can always be represented.
8. That is why you never see an “unknown token” error from a modern model.
9. The cost of the byte fallback is that unusual scripts get split into individual bytes, and one character can become three or four tokens.
10. There is also a pre-tokenization step. Before any merging, the text is split by a fixed pattern, usually keeping leading spaces attached to words and splitting runs of digits into groups.
11. That pattern is why “ the” with a leading space is a different token from “the” without one, and both exist in the vocabulary.

■ 47.7.5 TECHNICAL – the engineer’s version

1. The four families you need to be able to name and tell apart:

Family	Chooses merges by	First described
BPE	Highest pair frequency	Gage 1994; Sennrich 2016
WordPiece	Highest likelihood gain	Schuster, Nakajima 2012
Unigram LM	Prunes a large set	Kudo 2018
SentencePiece	Framework, not algorithm	Kudo, Richardson 2018

2. **Byte-pair encoding** was invented as a data compression algorithm by Philip Gage, published in The C Users Journal in February 1994. Rico Sennrich, Barry Haddow and Alexandra Birch adapted it to machine translation in Neural Machine Translation of Rare Words with Subword Units, posted 2015 and published at ACL 2016. GPT-2 in 2019 moved BPE to operate on raw bytes, giving the 256-symbol base alphabet and eliminating unknown tokens.
3. **WordPiece** came from Mike Schuster and Kaisuke Nakajima at Google in the 2012 ICASSP paper Japanese and Korean Voice Search. It merges the pair that most increases the likelihood of the training data under a unigram language model, rather than the most frequent pair. Concretely

it maximizes $\text{count}(xy) / (\text{count}(x) * \text{count}(y))$, so a pair of two rare symbols can beat a pair of two common ones. BERT, released by Google in October 2018, uses WordPiece, and marks word continuations with a double hash prefix.

4. **Unigram language model** tokenization was introduced by Taku Kudo in the 2018 ACL paper Subword Regularization. It works in the opposite direction: start with a large candidate vocabulary, score every candidate by how much removing it would hurt the corpus likelihood, and iteratively prune the worst until the target size is reached. Because it keeps a probability per token, it can sample alternative segmentations of the same string, which is used as a data augmentation technique.
5. **SentencePiece** is a library, not an algorithm, from Taku Kudo and John Richardson, EMNLP 2018 system demonstrations. It implements both BPE and Unigram, and its real contribution is treating input as a raw stream with no language-specific pre-tokenization, encoding the space character itself as a visible symbol so that detokenization is exactly reversible. That is what makes it work for Japanese, Chinese and Thai, which do not delimit words with spaces. T5, Llama 1 and Llama 2 use SentencePiece; Llama 3 moved to a tiktoken-style byte-level BPE with a 128,000-entry vocabulary.
6. Real vocabulary sizes, for scale:

Tokenizer	Vocabulary	Used by
GPT-2 (r50k)	50,257	GPT-2, GPT-3
cl100k_base	100,277	GPT-3.5, GPT-4
o200k_base	200,019	GPT-4o and later
Llama 3	128,256	Llama 3.x family

7. Larger vocabularies mean fewer tokens per document, which means cheaper attention and shorter sequences, but a larger embedding and output matrix. Going from 50k to 200k vocabulary at hidden size 4096 costs about 1.2 billion extra parameters across the two tables.
8. Special tokens are added outside the learned merges: sequence start, sequence end, padding, and chat role markers such as a header start marker. These are reserved ids that the merge process can never produce from text, which is a deliberate security property: user text cannot forge a role boundary.
9. Tools: tiktoken for the OpenAI encodings, tokenizers and AutoTokenizer in the Hugging Face stack, sentencepiece for the T5 and Llama 2 lineage, and llama_tokenize in llama.cpp.

■ 47.7.6 WORDS - remember these

1. Token — a small piece of text the model works in — an integer id indexing one row of the embedding matrix, produced by the tokenizer.
2. Subword — a piece smaller than a word — a token that is part of a word, such as “ization” in “tokenization”.
3. Byte-pair encoding — repeatedly glue the most common pair together — BPE, a greedy merge algorithm producing an ordered merge list.
4. Byte-level — starting from raw bytes, not letters — using the 256 byte values as the base alphabet so no input is ever unrepresentable.
5. Pre-tokenization — the split done before merging — a regular expression pass that fixes where merges may not cross, such as across whitespace.
6. Vocabulary — the full list of possible tokens — the id-to-string table, size V , equal to the number of rows in the embedding matrix.

7. Special token — a marker that is not ordinary text — a reserved id such as sequence start or a chat role header, unreachable by merging user text.
8. Detokenization — turning tokens back into text — concatenating the token strings; exactly reversible in SentencePiece and byte-level BPE.

47.8 What tokenization does in practice

■ 47.8.1 PLAIN - in simple words

1. Now let us look at real text going through a real tokenizer.
2. All the counts below are measured, not estimated.
3. English prose splits neatly. Most common words are one token each.
4. Numbers get chopped into groups of digits, and the groups do not line up with how you would read the number.
5. Web addresses split into many small pieces, because slashes, dots and question marks all break the text apart.
6. Program code splits reasonably well, because tokenizers are trained on a lot of code these days.
7. Hindi and other Indian scripts split badly on older tokenizers and much better on newer ones, but still worse than English.
8. An emoji is one character to you and often two tokens to an older model.
9. Two consequences follow directly, and they explain famous model failures.
10. The model cannot see letters, so it cannot reliably count them or reverse a word.
11. The model sees digits in arbitrary groups, so arithmetic on long numbers goes wrong for reasons that have nothing to do with reasoning.

■ 47.8.2 PLAIN - a picture in your head

1. Imagine reading a book through a narrow slot that shows a few characters at a time, but you do not control where the slot boundaries fall.
2. Somebody else cut the page into strips before handing it to you.
3. For English the strips mostly line up with words, so reading is easy.
4. For a number like 17394205 the strips fell at 173, 942, 05.
5. You can read the strips perfectly. You just never saw the digits as digits in their proper columns.
6. Now somebody asks you how many times the letter r appears in “strawberry”.
7. You never saw letters. You saw three strips: “str”, “aw”, “berry”.
8. You can guess, from having read the answer somewhere. You cannot count.

Where this comparison breaks: the model is not blind to letters in principle. It has seen enough spelled-out text during training to have learned a lot about which letters are inside which tokens. So it often gets these right. The point is that it is doing recall and inference rather than direct observation, which is why it fails unpredictably rather than never.

■ 47.8.3 PLAIN - a worked example

1. Real measurements, taken with the actual OpenAI tokenizers. cl100k_base is the GPT-4 era encoding; o200k_base is the later one.

Text	cl100k	o200k
The quick brown fox... (44 ch)	10	10
I live in India.	5	5

Text	cl100k	o200k
strawberry	3	3
1234567890	4	4
A 55-character URL	17	18
A 32-character Python function	12	12

2. The exact splits, which are the interesting part:

```
"strawberry" cl100k -> "str" "aw" "berry"
"strawberry" o200k -> "st" "raw" "berry"
"1234567890" both -> "123" "456" "789" "0"
"17394205" both -> "173" "942" "05"
"unbelievable" cl100k -> "un" "belie" "vable"
"tokenization" both -> "token" "ization"
```

3. Look at “17394205”. The groups are 173, 942, 05. That is not thousands, hundreds, tens. It is an arbitrary cut.

4. Now the code example, tokenized by cl100k_base into 12 tokens:

```
"def" " add" "(a" " ," " b" " )":NL" " " " " return" " a" " "+" " b"
(and a final newline token), where NL stands for a newline
```

5. Note that the four-space indent became a single token of three spaces plus a space attached to “return”. Indentation is compressed, which is why code tokenizes efficiently.
6. Now Devanagari. Take the Hindi sentence meaning “I live in India”, which is मैं भारत में रहता हूँ। In roman letters that is “main bharat mein rahta hoon”. It is 22 characters and 58 bytes of UTF-8, because each Devanagari character takes three bytes where an English letter takes one.

Encoding	Tokens for the Hindi	English is
GPT-2 (r50k)	34	5
cl100k_base	24	5
o200k_base	6	5

7. On the GPT-4 era tokenizer, the same meaning costs 24 tokens in Hindi and 5 in English. That is 4.8 times more, for identical content.
8. A longer measured pair: a 57-character Hindi sentence about tokenization costs 61 tokens on cl100k_base and 21 on o200k_base. The equivalent English sentence costs 11 on both.
9. So on cl100k_base the Hindi is 5.5 times the cost. On o200k_base it is 1.9 times. The gap narrowed a lot but did not close.
10. Emoji: the grinning face character, code point U+1F600, is one character and four UTF-8 bytes. cl100k_base spends 2 tokens on it. o200k_base spends 1.

■ 47.8.4 PLAIN – what is really happening inside

1. Why does Devanagari cost so much on the older tokenizers?
2. Because the tokenizer’s merges were learned from a corpus that was mostly English. Frequent English sequences got merged into single tokens.
3. Devanagari sequences were rare, so few merges were learned for them.
4. What is left is the byte fallback. Each Devanagari character is three bytes in UTF-8, and if no merge covers it, those three bytes become separate tokens.
5. That is the 3x penalty in its rawest form, before you even count the missing word-level merges.

6. Newer tokenizers were trained on far more multilingual text and given much larger vocabularies, so many whole Hindi words now have their own token. That is why o200k_base needed only 6 tokens where cl100k_base needed 24.
7. Now the letter-counting failure, mechanically.
8. The model receives the id for “berry” as a single integer. It receives no information whatsoever about the five letters inside it, except what it learned indirectly during training.
9. Asking it to count the letter r in “strawberry” is asking it to recall a fact about the internal composition of three token ids.
10. It can often do this, because spelling appears in training data. It fails when the tokens are unusual, or when the counting has to be exact.
11. Reversing a word is worse, because it requires the letters in order, which is even less directly available.
12. Arithmetic fails for the same class of reason. When 17394205 arrives as 173, 942, 05, the place values that column-wise addition depends on are not aligned with anything the model can see.
13. This is why some model families deliberately force digits to be split one at a time, and why models are much better at arithmetic when allowed to write the working out step by step rather than answering in one shot.

■ 47.8.5 TECHNICAL – the engineer’s version

1. Measured compression ratios, cl100k_base, on real text samples:

Text type	Chars/token	Tokens/word
Plain narrative prose	3.78	1.25
Ordinary email	4.58	1.12
Technical prose	4.87	1.07
Python code	3.76	2.07
Compact JSON	2.76	3.22

2. The common rules of thumb, roughly 4 characters per token and roughly 0.75 words per token in English, are conventions rather than standards, and the table shows they hold for prose and fail badly for structured data. JSON at 2.76 characters per token is 45 per cent worse than the rule of thumb suggests, which matters when you are budgeting a prompt full of API output.
3. The unfairness is documented. Aleksandar Petrov, Emanuele La Malfa, Philip Torr and Adel Bibi published Language Model Tokenizers Introduce Unfairness Between Languages at NeurIPS 2023, measuring the same content across many languages and finding token-count ratios up to 15 times between the best and worst served languages, with over 4 times difference remaining even for character and byte-level encoders.
4. The three consequences they name are exactly the ones that hurt an Indian developer: cost, because APIs bill per token; latency, because more tokens means more forward passes; and effective context, because a fixed window holds far less meaning in a penalized language.
5. Work the third one out. A 128,000-token window at 4.8 characters per token holds about 614,000 characters of English. At the cl100k_base Hindi rate of about 0.93 characters per token, the same window holds about 119,000 characters of Hindi. The window is the same. The content it holds is a fifth.
6. Mitigations that actually work: choose a model whose tokenizer was trained with substantial data in your language, measure with the real tokenizer rather than trusting a rule of thumb, and check whether the provider’s multilingual pricing is per token, which it almost always is.

7. On the letter-counting failure: the standard explanation is tokenization, and it is largely right, but be precise. Character-level models also make counting mistakes, and models can spell tokens correctly when asked step by step. So tokenization makes the information indirect rather than absent. That is an important distinction and it is often stated too strongly.
8. On arithmetic: the tokenizer's digit grouping is a genuine and measurable cause. Research on numeric tokenization has shown that forcing single-digit tokenization, or right-to-left digit grouping, measurably improves multi-digit arithmetic. Some model families now tokenize digits individually for this reason.
9. Practical tools: `tiktoken.get_encoding("cl100k_base").encode(text)` returns the ids; `enc.decode_single_token_bytes(i)` shows exactly what each id is; Hugging Face's `tokenizer.tokenize(text)` shows the string pieces.

```
import tiktoken
enc = tiktoken.get_encoding("cl100k_base")
ids = enc.encode("The cat sat on the mat.")
print(len(ids), ids)
for i in ids:
    print(i, enc.decode_single_token_bytes(i))
```

■ 47.8.6 WORDS - remember these

1. Compression ratio — how much text fits in one token — measured characters per token or tokens per word for a given tokenizer and text type.
2. Byte fallback — what happens when no merge covers the text — encoding characters as their individual UTF-8 bytes, one token each.
3. Tokenizer fertility — how many tokens a tokenizer spends per word — a standard fairness measure; higher fertility means a worse-served language.
4. Digit grouping — how numbers get cut up — the pre-tokenizer rule that splits runs of digits, commonly into groups of up to three.
5. UTF-8 — the standard way text becomes bytes — 1 byte for ASCII, 2 for Latin accented and Greek, 3 for Devanagari and most CJK, 4 for emoji.
6. Effective context — how much meaning your window actually holds — the window size divided by your language's tokens-per-meaning cost.

47.9 Token economics

■ 47.9.1 PLAIN - in simple words

1. When you use a model through a company's service, you pay per token.
2. Not per question. Not per minute. Per token in and per token out.
3. This is not arbitrary. Tokens are what the machine actually does work on.
4. One token in costs the model one pass through the multiply-and-add machinery, though those can be done for many tokens at once.
5. One token out costs a full pass through the entire model, and those cannot be done in parallel, because each output token depends on the one before.
6. That asymmetry is why output tokens cost more than input tokens.
7. Typically output is priced at four to six times input.
8. So a long question with a short answer is cheap. A short question with a very long answer is not.
9. There is a second, sneakier cost. Long conversations resend everything.
10. If you have a chat with 50 turns, every new question sends all 50 previous turns as input, because the model has no memory between requests.

11. So the input token count grows with the conversation and you pay for the whole history every single time.
12. That is the single most common reason a bill is bigger than expected.

■ 47.9.2 PLAIN – a picture in your head

1. Think of a translator you hire, who charges by the page.
2. Pages they read cost you a little. Pages they write cost you more, because writing is slower than reading.
3. That is input versus output pricing.
4. Now the important twist. This translator has no memory at all.
5. Every time you ask a follow-up question, you must hand them the entire previous conversation to read again, from the beginning.
6. On question one you hand over one page. On question fifty you hand over sixty pages, and pay to have all sixty read again.
7. Some translators offer a discount: if you hand them the exact same bundle of pages you handed them a few minutes ago, they charge much less for those.
8. That discount is prompt caching, and it is real.

Where this comparison breaks: a human translator reading sixty pages takes an hour. A model processes the input tokens of a long prompt in parallel, so sending a long history costs money and some latency but nothing like proportionally more wall-clock time than a short one.

■ 47.9.3 PLAIN – a worked example

1. Build a realistic support assistant and cost it.
2. Every request contains: a system prompt of 800 tokens, four retrieved document chunks totalling 4,000 tokens, 1,200 tokens of conversation history, and the user's 150-token question.
3. Input total: $800 + 4,000 + 1,200 + 150 = 6,150$ tokens.
4. The answer averages 350 tokens of output.
5. Prices per million tokens, checked on 13 August 2026, input then output:

Model	Input \$/1M	Output \$/1M	Per request
Claude Sonnet 5	2.00	10.00	\$0.0158
Claude Opus 5	5.00	25.00	\$0.0395
gpt-5.6-sol	2.50	15.00	\$0.0206
Gemini 3.6 Flash	1.50	7.50	\$0.0119

6. Work one out fully, for Claude Sonnet 5.
7. Input: $6,150 / 1,000,000 \times \$2.00 = \$0.0123$.
8. Output: $350 / 1,000,000 \times \$10.00 = \$0.0035$.
9. Total per request: \$0.0158.
10. At 10,000 requests a day that is \$158 a day, or \$4,740 a month.
11. On Claude Opus 5 the same traffic is \$11,850 a month.
12. On a small cheap model such as gpt-5.6-luna at \$0.10 in and \$0.60 out, the same traffic is \$247.50 a month.
13. Notice that the input tokens dominate: \$0.0123 of the \$0.0158, or 78 per cent. Cutting retrieved context is worth far more than shortening answers.

■ 47.9.4 PLAIN – what is really happening inside

1. Watch what happens when the context grows and everything else stays fixed.

Input tokens	Cost per request	Per 10,000
6,150	\$0.0158	\$158
14,150	\$0.0318	\$318
38,150	\$0.0798	\$798
106,150	\$0.2158	\$2,158
406,150	\$0.8158	\$8,158

2. Those are all the same question, the same answer, and the same model. The only change is how much context was stuffed in.
3. Going from 6,000 to 400,000 tokens of context multiplies the bill by 52.
4. This is why “just put the whole document in the context window” is a financial decision, not only a technical one.
5. Now caching. The system prompt and the retrieved chunks are often identical between requests within a short window.
6. Providers offer a large discount on input tokens that are an exact prefix match of a recent request. One current example prices cached input at one tenth of fresh input.
7. Rework the example on a provider charging \$2.50 fresh, \$0.25 cached and \$15.00 output. Without caching: \$0.0206 per request.
8. With the 4,800-token system-plus-documents prefix cached: 4,800 at \$0.25 plus 1,350 at \$2.50 plus 350 at \$15.00 = \$0.0098.
9. That is a 52 per cent saving for changing nothing but the ordering and stability of the prompt.
10. The rule that follows: put the stable parts of your prompt first and the variable parts last, because prefix caching only matches from the start.

■ 47.9.5 TECHNICAL – the engineer’s version

1. Why output costs more, precisely. Processing input is the **prefill** phase: all input tokens go through the model in one batched forward pass, and the hardware is compute-bound and highly efficient.
2. Producing output is the **decode** phase: one token at a time, each requiring a full pass over every weight, and at small batch sizes the hardware is memory-bandwidth-bound and mostly idle in its arithmetic units.
3. So a decoded token costs far more machine time than a prefilled one, and the pricing reflects that. The typical ratio of 4x to 6x is a convention that has held across providers, not a standard.
4. Reasoning models add a third category. Tokens generated internally before the visible answer are billed as output tokens even though you never see them, and they can be several times the visible answer’s length. Budget for them explicitly.
5. Long-context surcharges are now common. As of August 2026, one provider prices prompts over 200,000 tokens at double the input rate and 1.5 times the output rate; another prices a long-context tier at double input and 1.5 times output above a threshold. Check the specific model’s page, because this changes often.
6. Batch APIs typically halve the price in exchange for asynchronous delivery within a window, commonly 24 hours. If your workload is not interactive, this is the largest single discount available.

7. Estimating before you build: tokenize a representative sample of your real prompts with the actual tokenizer and take the mean. Do not estimate from character counts if your content is JSON, code or a non-Latin script, because the rule of thumb is wrong by up to a factor of five in those cases.
8. A checklist that reduces token spend, in order of typical impact:
 1. Retrieve less. Four chunks instead of twenty.
 2. Cache the stable prefix; put variable text last.
 3. Summarize old conversation turns instead of resending.
 4. Use a small model for routing and a large one only when needed.
 5. Cap max output tokens per request.
 6. Batch anything that is not interactive.
 7. Strip whitespace and boilerplate from injected documents.
9. Self-hosting changes the shape of the cost, not always its size. You pay for the machine by the hour whether it is busy or not, so self-hosting wins at high steady utilization and loses badly at low or spiky utilization.
10. Observability: every provider returns a usage object with input and output token counts per response. Log it per request with a feature label, or you will not be able to attribute the bill later.

■ 47.9.6 WORDS – remember these

1. Input token — a token you send — a token in the prompt, processed in the prefill phase, billed at the lower rate.
2. Output token — a token the model writes — a token produced in the decode phase, billed at the higher rate, including hidden reasoning tokens.
3. Prefill — reading the prompt — the parallel forward pass over all input tokens, populating the key-value cache.
4. Decode — writing the answer — the sequential per-token generation loop, one full forward pass per token.
5. Prompt caching — paying less for a repeated prefix — reusing a stored key-value cache for an exact matching prompt prefix, at a discounted rate.
6. Batch API — cheaper if you can wait — asynchronous processing at roughly half price with a delivery window, commonly 24 hours.
7. Context stuffing — putting everything in the prompt — a strategy whose cost grows linearly in tokens and whose attention cost grows faster.

47.10 Embeddings: turning a token id into a vector

■ 47.10.1 PLAIN – in simple words

1. After tokenization you have a list of integers. The model cannot use those directly.
2. An integer id like 8415 is just a name. Token 8416 is not “one more than” token 8415 in any meaningful way.
3. So each id has to be turned into something with actual structure.
4. It is turned into a list of numbers. A long one: often 4,096 numbers.
5. That list is called the token’s **embedding**.
6. Where does the list come from? A big table.
7. The table has one row per possible token, and one column per number in the list.
8. If there are 128,256 possible tokens and each embedding is 4,096 numbers, the table is 128,256 rows by 4,096 columns.
9. Turning an id into an embedding is just reading row number 8415 of the table.
10. That is a lookup. No multiplication, no cleverness. Fetch the row.

11. The numbers in the table were learned during training, so the table is made of parameters, exactly like every other matrix in the model.
12. And that is the entrance to the model. Text became ids. Ids became rows of numbers. Now the layers can do arithmetic.

■ 47.10.2 PLAIN – a picture in your head

1. Think of a very long index card box, with one card per word.
2. On each card, instead of a definition, there are 4,096 numbers.
3. To look up a word you find its card and copy the numbers off it.
4. The numbers mean nothing to you individually. Number 1,872 on the card for “cat” is 0.031, and there is no sentence explaining why.
5. But the cards are arranged in a very particular way in an imaginary space.
6. Cards for similar things ended up near each other. The card for “cat” is close to the card for “dog”, and far from the card for “invoice”.
7. Nobody sorted them deliberately. They ended up there because training pushed words used in similar ways toward similar numbers.
8. So the meaning is in the arrangement, not in any single number on any card.

Where this comparison breaks: index cards sit on a table, which is two dimensional. These cards sit in a space with 4,096 directions. Almost every intuition you have about “near” and “far” from two dimensions gets weaker in 4,096, and some of them break completely. That gets its own honest treatment below.

■ 47.10.3 PLAIN – a worked example

1. Take a real sentence and follow it in.
2. The sentence: “The cat sat on the mat.”
3. Tokenized with cl100k_base it becomes 7 tokens with these real ids:

Id	Piece
791	The
8415	(space)cat
7731	(space)sat
389	(space)on
279	(space)the
5634	(space)mat
13	.

4. Note that “The” at the start and “the” in the middle are different tokens with different ids, because one has a leading space and a capital letter.
5. Now the lookup. The embedding matrix for an 8B-class model is 128,256 x 4,096.
6. Row 791 is fetched. It is 4,096 numbers. Row 8415 is fetched. Also 4,096.
7. Do that for all seven ids and stack them.
8. The result is a block of numbers 7 rows by 4,096 columns.
9. In the shape notation engineers use, the tensor went from [7] to [7, 4096].
10. That is 28,672 numbers describing a six-word sentence.
11. Size of the table itself: 128,256 x 4,096 = 525,336,576 parameters.
12. At two bytes each, that one table is about 1.05 gigabytes, which is 6.5 per cent of the whole 8B model.

■ 47.10.4 PLAIN – what is really happening inside

1. The lookup is mathematically a matrix multiplication, done the lazy way.
2. The formal version: make a vector of 128,256 numbers that is all zeros except a single 1 in position 8415. That is called a one-hot vector.
3. Multiply that one-hot vector by the embedding matrix. The result is row 8415.
4. Every implementation skips the multiplication and just indexes the row, because multiplying by 128,255 zeros is a waste.
5. That is why the operation is called an embedding lookup rather than an embedding multiply, even though it is a linear layer.
6. One thing is missing at this point: order.
7. The lookup gives the same vector for “cat” whether it is the first word or the fiftieth. Position information has to be added separately.
8. Older models added a learned position vector to the token vector. Modern models mostly rotate the query and key vectors inside attention instead, which Chapter 48 covers.
9. From here the stacked embeddings flow into layer 1, then layer 2, and so on.
10. Critically, the vector at each position keeps being updated as it passes through layers. It starts as “the meaning of this token in general” and becomes “the meaning of this token in this sentence”.
11. At the very end, the vector at the last position is multiplied by the output matrix to produce one score per possible token, and the highest scores are the model’s prediction of what comes next.

■ 47.10.5 TECHNICAL – the engineer’s version

1. The embedding matrix has shape $[V, d]$ and is exactly $V \times d$ parameters.

Model	V	d	Embedding params
GPT-2 small	50,257	768	38,597,376
Llama 3.1 8B	128,256	4,096	525,336,576
Llama 3.1 70B	128,256	8,192	1,050,673,152
Llama 3.1 405B	128,256	16,384	2,101,346,304

2. In PyTorch this is `nn.Embedding(V, d)`, whose forward pass is `F.embedding`, which is an index-select, not a matmul.
3. The output projection, `lm_head`, has shape $[V, d]$ as well and produces **logits**: one raw score per vocabulary entry. Softmax over logits gives the next-token distribution.
4. If embeddings are tied, `lm_head.weight` is the same storage as `embed_tokens.weight`. Tying was popularized by Press and Wolf in 2016 in Using the Output Embedding to Improve Language Models, and independently by Inan, Khosravi and Socher the same year.
5. Common hidden sizes, which are also the embedding dimensions: 768 for GPT-2 small, 1024 and 1600 for larger GPT-2 variants, 2048 for many 1B-class models, 4096 for 7B and 8B, 5120 for 13B, 8192 for 70B, 16384 for 405B.
6. Now the honest note about dimensionality that the reader asked for.
7. You cannot visualize 4,096 dimensions. Nobody can. Any picture you have seen of word vectors as arrows on a page is a projection down to two dimensions by a method such as PCA, t-SNE or UMAP, and those projections distort distances substantially. t-SNE in particular preserves local neighbourhoods and does not preserve global distances at all, so cluster sizes and the gaps between clusters in a t-SNE plot are not meaningful.

8. What does carry over from 2D and 3D: dot products, cosine of the angle between vectors, Euclidean distance, addition and subtraction of vectors, and the idea that direction can encode something.
9. What does not carry over, and this is the part people get wrong:
 1. In high dimensions almost every pair of random vectors is nearly orthogonal, so a cosine of 0 means “unrelated”, not “opposite”.
 2. Distances concentrate: the ratio between the nearest and farthest point in a random set shrinks toward 1 as dimension grows, which is one face of the curse of dimensionality.
 3. The volume of a high-dimensional ball is almost entirely near its surface, so intuitions about “the middle of the cloud” are wrong.
 4. There is far more room than you expect. In d dimensions you can pack exponentially many nearly-orthogonal directions, which is why a 4,096-wide space can hold far more than 4,096 distinguishable concepts.
10. That last point has a name in current research: **superposition**, the hypothesis that models represent many more features than they have dimensions, by using nearly-orthogonal directions and tolerating small interference. Anthropic’s 2022 paper Toy Models of Superposition is the standard reference. This is active research, not settled fact.

■ 47.10.6 WORDS – remember these

1. Embedding — a token turned into a list of numbers — a row of the embedding matrix, a vector in $\mathbb{R}^d$.
2. Embedding matrix — the lookup table — a $[V, d]$ parameter tensor mapping token ids to vectors.
3. One-hot vector — all zeros with a single 1 — the formal input to an embedding layer, never materialized in practice.
4. Logits — raw scores before they become probabilities — the $[V]$ output of the final projection, converted by softmax.
5. Dimension — how many numbers are in one vector — d , the model’s hidden size, typically 768 to 16,384.
6. Curse of dimensionality — high-dimensional space behaves strangely — the family of effects including distance concentration and near-orthogonality of random vectors.
7. Superposition — packing more ideas than there are directions — the hypothesis that features are stored along nearly-orthogonal directions with tolerated interference. Active research.

47.11 What embeddings capture

■ 47.11.1 PLAIN – in simple words

1. If two words have similar embeddings, the model treats them similarly.
2. So the natural question is: how do you measure “similar” for two lists of numbers?
3. Two standard answers, both simple.
4. **Distance**: treat each list as a point, and measure how far apart the points are. Closer means more similar.
5. **Angle**: treat each list as an arrow from the origin, and measure the angle between the arrows. Smaller angle means more similar.
6. The angle version is the one almost everybody uses, and it is called **cosine similarity**.
7. It gives 1 for arrows pointing the same way, 0 for arrows at right angles, and -1 for arrows pointing opposite ways.

8. The angle version ignores how long the arrows are and looks only at direction, which is usually what you want.
9. There is also a famous claim that these vectors capture relationships, not just similarity.
10. The claim is that king minus man plus woman lands near queen.
11. It is a real effect. It is also considerably weaker than the popular story, and the popular demonstration was rigged in a specific way. Section 47.11.3 and 47.11.5 give the honest account.

■ 47.11.2 PLAIN – a picture in your head

1. Think of a map of a country, where each town is a dot.
2. Two towns close together on the map are similar in location. That is distance.
3. Now draw an arrow from the capital to every town.
4. Two towns roughly in the same direction from the capital, one near and one far, have arrows at a small angle. That is cosine similarity.
5. On a map, “50 km north-east” is a direction you can apply anywhere. Start at any town, go 50 km north-east, arrive somewhere new.
6. The analogy claim is that word space has directions like that. There is a direction that means “make it female”, and applying it to king lands on queen.
7. The map picture makes clear why this could work and why it might not.
8. On a real map, “50 km north-east” from a coastal town lands in the sea.

Where this comparison breaks: a map has two dimensions and every direction is meaningful. Word space has thousands, and only a tiny number of directions correspond to anything a human would name. Most directions in the space mean nothing at all.

■ 47.11.3 PLAIN – a worked example

1. Cosine similarity, computed by hand on small vectors.
2. Take four-number toy vectors, made up for illustration:

```
cat = [0.8, 0.1, 0.7, 0.2]
dog = [0.7, 0.2, 0.8, 0.1]
car = [0.1, 0.9, 0.0, 0.8]
```

3. The formula: $\text{cosine} = (a \cdot b) / (|a| \times |b|)$, where $a \cdot b$ is the sum of pairwise products and $|a|$ is the square root of the sum of squares.
4. Step 1, the dot product of cat and dog: $0.8 \times 0.7 + 0.1 \times 0.2 + 0.7 \times 0.8 + 0.2 \times 0.1 = 0.56 + 0.02 + 0.56 + 0.02 = 1.16$.
5. Step 2, the length of cat: $\text{sqrt}(0.64 + 0.01 + 0.49 + 0.04) = \text{sqrt}(1.18) = 1.0863$.
6. Step 3, the length of dog: also 1.0863.
7. Step 4: $\text{cosine} = 1.16 / (1.0863 \times 1.0863) = 1.16 / 1.18 = 0.9831$.
8. An angle of 10.6 degrees. Very similar.
9. Now cat against car. Dot product = $0.08 + 0.09 + 0.00 + 0.16 = 0.33$. Lengths 1.0863 and 1.2083. Cosine = $0.33 / 1.3126 = 0.2514$.
10. An angle of 75.4 degrees. Nearly unrelated.
11. Compare with straight-line distance: cat to dog is 0.20, cat to car is 1.41. Both measures agree here, which is common but not guaranteed.
12. Now the analogy, on a three-number toy set:

```
king    = [0.92, 0.81, 0.12]
man     = [0.88, 0.14, 0.09]
woman  = [0.13, 0.16, 0.91]
```

```
queen    = [0.15, 0.74, 0.86]
princess = [0.20, 0.60, 0.88]
```

13. king - man + woman = [0.17, 0.83, 0.94].
14. Cosine of that result against every word in the toy set:

Word	Cosine with result
queen	0.9999
princess	0.9911
woman	0.8556
king	0.6041
man	0.3092

15. Queen wins. The demonstration works. But look at the third row: woman, one of the inputs, scores 0.8556, well above king and man.
16. That third row is where the honest correction lives.

■ 47.11.4 PLAIN - what is really happening inside

1. The standard way to answer an analogy from vectors is called 3CosAdd.
2. Compute $b - a + c$, then find the vocabulary word whose vector has the highest cosine with the result.
3. There is a rule buried in the standard implementation: the three input words are excluded from the search.
4. So when you compute king - man + woman, the code refuses to answer king, man or woman, no matter how well they score.
5. That matters a great deal, because very often the highest-scoring word is one of the inputs.
6. The result you are shown is therefore the best answer after the most likely answers have been forbidden.
7. That does not make the effect fake. Queen really does score highly, and consistently, on many word pairs.
8. It makes the effect smaller and less magical than the demonstrations imply.
9. When researchers removed the exclusion rule and let the inputs compete, accuracy collapsed. The numbers are in the technical block below.
10. There is a second, separate honesty point about these vectors.
11. In word2vec, each word has exactly one vector, forever. "Bank" gets one vector that has to serve both the river bank and the money bank.
12. Modern models do not work that way. The vector for a word changes depending on the sentence it appears in.

■ 47.11.5 TECHNICAL - the engineer's version

1. Word2vec was published by Tomas Mikolov and colleagues at Google in 2013, in two papers: Efficient Estimation of Word Representations in Vector Space in January 2013, and Distributed Representations of Words and Phrases and their Compositionality at NeurIPS 2013. The analogy result appeared in a third, Linguistic Regularities in Continuous Space Word Representations, by Mikolov, Yih and Zweig at NAACL in June 2013.
2. GloVe followed from Jeffrey Pennington, Richard Socher and Christopher Manning at Stanford, EMNLP 2014, fitting vectors to global co-occurrence counts rather than local context windows.

3. The vector offset method, 3CosAdd, ranks candidate d by $\cos(d, b - a + c)$. Omer Levy and Yoav Goldberg proposed 3CosMul in 2014 in Linguistic Regularities in Sparse and Explicit Word Representations, which multiplies similarities instead of adding offsets and performs better.
4. Now the correction, which is well documented and not fringe.
5. Tal Linzen's 2016 RepEval paper Issues in Evaluating Semantic Spaces Using Word Analogies showed that a large part of reported analogy accuracy is explained by simple baselines: for many analogy items, just returning the nearest neighbour of c , ignoring a and b entirely, scores well.
6. Malvina Nissim, Rik van Noord and Rob van der Goot published Fair Is Better than Sensational: Man Is to Doctor as Woman Is to Doctor in Computational Linguistics, volume 46 issue 2, 2020. They showed the standard implementations forbid returning any of the three input words, and that this is not a detail.
7. Their measured result: when they modified the algorithms to allow input words as answers, 3CosAdd accuracy fell from 71 per cent to 21 per cent, and 3CosMul fell from 73 per cent to 45 per cent.
8. Read that carefully. On the standard benchmark, roughly two thirds of 3CosAdd's apparent success disappears when the exclusion is lifted. The frequent correct-if-allowed answer is one of the inputs, usually b .
9. So the accurate statement is: the offset direction carries real relational information, and it is much weaker than the "king - man + woman = queen" story suggests. Both halves of that sentence are true, and popular accounts usually give only the first half.
10. The same paper makes a separate methodological point about bias claims: asking a system "man is to doctor as woman is to what" while forbidding it from answering "doctor" guarantees a different answer, and then reporting that different answer as evidence of bias is unsound. Bias in embeddings is real and separately demonstrated; that particular demonstration is not sound evidence of it.
11. **Static versus contextual embeddings.** Word2vec and GloVe are static: one vector per word type, fixed. Contextual embeddings give a different vector per occurrence.
12. ELMo, from Matthew Peters and colleagues at the Allen Institute, NAACL 2018, Deep Contextualized Word Representations, was the first widely used contextual model. BERT, from Jacob Devlin and colleagues at Google, posted October 2018 and published at NAACL 2019, made it standard.
13. In a transformer, the row fetched from the embedding matrix is static, and it becomes contextual immediately: after the first attention layer the vector at that position has mixed in information from the other tokens.
14. So in the sentence "I sat on the river bank" and "I went to the bank for a loan", the input embedding for "bank" is byte-for-byte identical, and the representation at layer 12 is quite different. That is the whole point of the architecture.

Property	Static (word2vec)	Contextual
Vectors per word	One	One per occurrence
Handles polysemy	No	Yes
Needs a model to compute	No, table lookup	Yes, forward pass
Typical year	2013 to 2017	2018 onward

■ 47.11.6 WORDS – remember these

1. Cosine similarity — how aligned two lists of numbers are — the dot product divided by the product of the norms; 1 identical direction, 0 orthogonal.

2. Euclidean distance — straight-line distance between two points — the square root of the sum of squared differences.
3. Dot product — pairwise multiply and add — the unnormalized similarity, which grows with vector length as well as alignment.
4. Vector offset — subtracting one vector from another — the $b - a$ term in the 3CosAdd analogy method, treated as a relation direction.
5. Static embedding — one fixed vector per word — word2vec or GloVe style, a pure lookup with no sentence context.
6. Contextual embedding — a different vector each time — the hidden state at a position after attention has mixed in surrounding tokens.
7. Polysemy — one word with several meanings — the phenomenon static embeddings cannot represent and contextual embeddings can.

47.12 Embeddings as a tool in their own right

■ 47.12.1 PLAIN - in simple words

1. So far embeddings have been a part inside a language model.
2. They are also a useful product on their own, sold separately.
3. The idea: instead of one vector per token, get one vector for a whole sentence, paragraph or document.
4. Now you can compare pieces of text by comparing numbers.
5. That single capability unlocks five very common jobs.
6. **Semantic search**: find documents that mean the same thing as a query, even if they share no words with it.
7. **Clustering**: group thousands of support tickets into themes automatically.
8. **Recommendation**: suggest articles near the ones a reader liked.
9. **Deduplication**: find near-identical items that keyword matching misses.
10. **Retrieval for a model**: fetch the few relevant paragraphs from a large collection and paste them into the prompt.
11. That last one is the foundation of retrieval-augmented generation, which Chapter 51 covers in full.
12. The thing that makes all five work is the same: meaning became geometry, and geometry is cheap to compute with.

■ 47.12.2 PLAIN - a picture in your head

1. Imagine a library where every book has been assigned a position in a very large warehouse.
2. Books are not shelved by author or title. They are shelved by what they are about.
3. Two books about monsoon farming end up on adjacent shelves even if one is in English and one in Hindi and they share no words.
4. To answer a question you do not read anything. You compute where the question would sit in the warehouse, walk to that spot, and take the nearest ten books.
5. That is semantic search, exactly.
6. Keyword search is the old catalogue: it finds books whose title contains your exact words, and misses everything phrased differently.

Where this comparison breaks: a warehouse has aisles you can walk. A vector space has thousands of dimensions and no aisles, so finding the nearest items needs a specialized index. Doing it by brute force means comparing against every item, which is fine for ten thousand documents and far too slow for a hundred million.

■ 47.12.3 PLAIN – a worked example

1. Suppose four short documents and one query, each already turned into a four-number vector by an embedding model.

```
query = [0.75, 0.15, 0.60, 0.20]

doc A "how to feed a kitten" = [0.80, 0.10, 0.70, 0.20]
doc B "puppy training basics" = [0.70, 0.20, 0.80, 0.10]
doc C "car insurance renewal" = [0.10, 0.90, 0.00, 0.80]
doc D "truck loading limits" = [0.20, 0.80, 0.10, 0.90]
```

2. Compute the cosine of the query against each.
3. The query's length is $\sqrt{0.5625 + 0.0225 + 0.36 + 0.04} = \sqrt{0.985} = 0.9925$.
4. Query and doc A: dot = $0.60 + 0.015 + 0.42 + 0.04 = 1.075$. Doc A's length is 1.0863. Cosine = $1.075 / (0.9925 \times 1.0863) = 0.9971$.
5. Query and doc B: dot = $0.525 + 0.03 + 0.48 + 0.02 = 1.055$. Cosine = 0.9786.
6. Query and doc C: dot = $0.075 + 0.135 + 0.00 + 0.16 = 0.37$. Cosine = 0.3085.
7. Query and doc D: dot = $0.15 + 0.12 + 0.06 + 0.18 = 0.51$. Cosine = 0.4196.

Document	Cosine	Rank
A, kitten feeding	0.9971	1
B, puppy training	0.9786	2
D, truck loading	0.4196	3
C, car insurance	0.3085	4

8. The two animal documents rank above the two vehicle documents, and no word from the query appears in any of them.
9. In a real system the vectors have 384, 768, 1024 or 3072 numbers instead of four, and there are millions of documents, but the computation per pair is exactly this one.

■ 47.12.4 PLAIN – what is really happening inside

1. Where does a whole-sentence vector come from?
2. Run the text through a model and you get one vector per token. You need one vector for the whole thing.
3. Three common ways to collapse them.
4. **Mean pooling**: average all the token vectors. Simple, works well, still the most common method.
5. **CLS pooling**: use the vector at a special marker token placed at the start, which was trained to summarize the sequence.
6. **Last-token pooling**: in a decoder-only model, use the vector at the final position, which has seen everything before it.
7. But a raw language model's vectors are not well suited to comparison straight out of the box.
8. So embedding models are trained specially, with a method called contrastive training.
9. You show the model many pairs. Some pairs mean the same thing, some do not.
10. You push the matching pairs' vectors together and the non-matching pairs' vectors apart.
11. After enough pairs, cosine similarity in that space genuinely tracks meaning similarity, which it does not automatically do otherwise.
12. Finally, vectors are usually normalized to length 1, which makes cosine similarity and dot product the same computation, and makes the search faster.

■ 47.12.5 TECHNICAL – the engineer’s version

1. Embedding models are separate, smaller models from the generative ones, and they are usually encoder-style rather than decoder-style.
2. Typical output dimensions and what they cost to store, per million documents:

Dimensions	Bytes at fp32	Per 1M docs
384	1,536	1.5 GB
768	3,072	3.1 GB
1,024	4,096	4.1 GB
3,072	12,288	12.3 GB

3. Sentence-BERT, by Nils Reimers and Iryna Gurevych, EMNLP 2019, is the paper that made sentence embeddings practical, by fine-tuning BERT with a siamese network so that cosine similarity between sentence vectors is meaningful. Before it, comparing raw BERT sentence vectors performed worse than averaging GloVe vectors.
4. Contrastive objectives in common use: InfoNCE, multiple-negatives ranking loss, and in-batch negatives, where every other item in the batch is treated as a negative example for free.
5. Matryoshka representation learning, from a 2022 paper by Aditya Kusupati and colleagues, trains a model so that the first 256 or 512 numbers of a 3,072- number vector are usable on their own. Several current embedding APIs expose this as a dimension parameter, letting you trade accuracy for storage without re-embedding.
6. Nearest-neighbour search at scale uses an approximate index rather than brute force. HNSW, Hierarchical Navigable Small World graphs, from Yury Malkov and Dmitry Yashunin in 2016, is the most widely deployed. IVF with product quantization, from the FAISS library released by Facebook AI Research in 2017, is the other common family.
7. Approximate means approximate: these indexes return the true nearest neighbours most of the time, not always, and the recall is a tunable parameter traded against speed. That is a real correctness consideration and is often glossed over.
8. Vector databases and vector indexes in wide use as of 2026 include FAISS, pgvector for PostgreSQL, Qdrant, Weaviate, Milvus, Chroma and the vector features of Elasticsearch and OpenSearch.
9. Retrieval-augmented generation, named in a 2020 paper by Patrick Lewis and colleagues at Facebook AI Research, is embedding search plus prompt construction: embed the query, retrieve the top k chunks by cosine, paste them into the context, and let the generative model answer from them. Chapter 51 covers the whole pipeline including chunking and reranking.
10. Honest limitation: pure vector search is worse than keyword search at exact matching, such as product codes, error numbers and names. Production systems almost always combine both, which is called hybrid search, usually merged with reciprocal rank fusion.
11. A second honest limitation: an embedding has no notion of truth or recency. Two documents that contradict each other on a fact will sit close together, because they are about the same thing.

■ 47.12.6 WORDS – remember these

1. Embedding model — a model whose output is a vector, not text — usually an encoder trained with a contrastive objective for similarity.
2. Pooling — turning many token vectors into one — mean, CLS or last-token aggregation over the sequence.
3. Contrastive training — pull matches together, push non-matches apart — an objective such as InfoNCE using in-batch negatives.

4. Semantic search — search by meaning, not words — ranking documents by cosine similarity between query and document embeddings.
5. Approximate nearest neighbour — fast, nearly-correct nearest search — index structures such as HNSW or IVF-PQ that trade recall for speed.
6. Hybrid search — combining keyword and vector search — merging BM25 and vector rankings, commonly with reciprocal rank fusion.
7. Normalization — scaling every vector to length 1 — makes dot product equal cosine similarity and speeds up search.

47.13 The context window

■ 47.13.1 PLAIN – in simple words

1. The **context window** is the maximum number of tokens the model can be given at once, including everything: system instructions, history and your question.
2. It is measured in tokens, not words and not characters.
3. It is a hard limit set by the model, not a preference.
4. Why is there a limit at all? Because of how attention works.
5. Attention lets every token look at every other token.
6. With 100 tokens that is 10,000 pairs. With 1,000 tokens, a million pairs.
7. Double the length and the number of pairs goes up four times, not two.
8. So cost grows with the square of the length. Chapter 48 works through why.
9. When you exceed the window, one of two things happens.
10. Either the service refuses the request with an error, or something silently gets cut off, usually the oldest part of the conversation.
11. Silent cutting is the dangerous one, because the model then answers confidently based on text it never received.
12. And there is a further catch. A model that accepts a long input does not necessarily use all of it well.

■ 47.13.2 PLAIN – a picture in your head

1. Think of a desk with a fixed area.
2. You can spread out papers on it, and you can read anything on the desk instantly.
3. But when the desk is full and you add another page, one falls off the edge.
4. Nobody tells you which one fell off, or that anything fell at all.
5. A bigger desk helps. It also costs more, and it takes longer to scan.
6. Now the awkward part. Even with a huge desk, if you spread out four hundred pages, you will reliably notice the first few and the last few and be much vaguer about the middle.
7. That is not a memory limit. It is an attention limit, and models have a measurable version of the same thing.

Where this comparison breaks: a desk keeps papers between work sessions. A context window does not persist at all. When the request ends, everything on the desk is swept away, and the next request starts empty. Anything that appears to persist was resent by the application.

■ 47.13.3 PLAIN – a worked example

1. Work out what a window actually holds.
2. At about 4 characters per token, a 128,000-token window holds roughly 512,000 characters of English.

3. A typical printed page is about 2,000 characters, so that is about 256 pages.
4. But you never get all of it. Reserve space for the answer, and for the system prompt.
5. Now the attention arithmetic, which is why longer costs so much more:

Tokens	Pairs (n squared)	Relative to 512
512	262,144	1x
2,048	4,194,304	16x
8,192	67,108,864	256x
32,768	1,073,741,824	4,096x
131,072	17,179,869,184	65,536x
1,048,576	1,099,511,627,776	4,194,304x

6. Going from 512 to 1,048,576 tokens is 2,048 times the length and about four million times the attention pair count.
7. Memory grows too, but linearly, through the key-value cache: 40 GiB for a 70B model at 128,000 tokens at 16-bit, as computed in section 47.4.
8. So a million-token context is not a small engineering step from a thousand-token one. It requires different algorithms, not just more memory.

■ 47.13.4 PLAIN – what is really happening inside

1. What happens when the input is too long? Four common strategies.
2. **Reject**: return an error. Honest, and the only one that cannot silently mislead.
3. **Truncate**: cut the input to fit. Usually the oldest turns go first. Cheap, and information is genuinely lost.
4. **Sliding window**: keep a moving window of the most recent tokens and drop what falls out of the back. Used in streaming settings.
5. **Summarize and compact**: replace old turns with a shorter summary generated by the model, keeping recent turns verbatim. This is what most long-running assistants do.
6. Each strategy loses something, and you should know which one your framework is using, because they fail differently.
7. Now the harder problem. Suppose the whole document does fit.
8. The model still does not attend evenly across it.
9. Researchers put a single fact somewhere in a long context and asked a question that requires it, varying only the position.
10. Accuracy was highest when the fact was at the very beginning or the very end, and lowest when it was in the middle.
11. The shape of the curve is a U. The finding is called “lost in the middle”.
12. So placing your most important instruction in the middle of a long prompt is a measurably bad idea.

■ 47.13.5 TECHNICAL – the engineer’s version

1. Context windows have grown by more than three orders of magnitude in seven years:

Model and year	Context window
GPT-2, 2019	1,024
GPT-3, 2020	2,048

Model and year	Context window
Llama 2, 2023	4,096
Llama 3.1, 2024	131,072

- As of 13 August 2026, current commercial figures: Claude Opus 5, Claude Sonnet 5 and Claude Fable 5 accept 1,000,000 input tokens with up to 128,000 output tokens; Claude Haiku 4.5 accepts 200,000. The GPT-5.6 family accepts about 1.05 million input tokens, which is 2^{20} , with 128,000 output. These figures change often; check the provider's model page.
- The cost of attention is $O(n^2)$ in time for the score matrix and $O(n)$ in memory for the key-value cache. FlashAttention, from Tri Dao and colleagues in 2022, does not change the $O(n^2)$ compute but avoids materializing the $n \times n$ matrix in high-bandwidth memory, which removes the $O(n^2)$ memory term and is what made long contexts practical. Chapter 48 covers the mechanism.
- Position encoding is the second limit. RoPE, rotary position embedding from Jianlin Su and colleagues in 2021, encodes position by rotating query and key vectors. Extending a trained model beyond its training length needs the rotation frequencies adjusted, by methods such as position interpolation or the NTK-aware and YaRN scalings. A model advertised at 128,000 tokens is often a model trained at 8,000 and extended.
- Lost in the middle:** Nelson Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni and Percy Liang, arXiv July 2023, published in TACL 2024. Across multi-document question answering and key-value retrieval, performance was highest when the relevant information was at the start or end of the input and degraded significantly in the middle, and extended-context models were not reliably better than their shorter counterparts at using the extra length.
- RULER**, from Cheng-Ping Hsieh and colleagues at NVIDIA, April 2024, extended the needle-in-a-haystack test with multi-hop tracing and aggregation tasks and evaluated 17 long-context models. Its headline finding: although all claimed 32,000 tokens or more, only about half maintained satisfactory performance at 32,000, despite near-perfect scores on the simple retrieval test.
- So state the distinction plainly. **Claimed context** is the number the API will accept without an error. **Effective context** is the length at which the model still performs near its short-context accuracy. The second number is smaller, is model-specific, and is rarely published by vendors.
- The needle-in-a-haystack test, popularized by Greg Kamradt in November 2023, is a weak measure on its own: passing it shows the model can find one exact verbatim string, which is much easier than reasoning over dispersed information.
- Practical rules that follow: put critical instructions at the beginning and repeat them at the end; retrieve fewer, better chunks rather than more; measure your own task at your own lengths rather than trusting the advertised number; and log token counts so you notice when truncation starts.
- Tools: every API returns input token counts in its usage object; Anthropic and OpenAI both expose a token counting endpoint or library so you can check before sending; tiktoken and AutoTokenizer let you count offline.

■ 47.13.6 WORDS – remember these

- Context window — the most tokens the model accepts at once — the maximum input sequence length, a fixed architectural and training limit.
- Truncation — cutting input to make it fit — dropping tokens, usually oldest first, often silently.
- Sliding window — a moving view of recent tokens — keeping the last n tokens and discarding what falls out.
- Lost in the middle — the dip in accuracy for content in the centre — the U-shaped position-versus-accuracy curve documented by Liu and colleagues.

5. Effective context — how much of the window is actually usable — the length at which task accuracy still holds up, usually shorter than the claimed window.
6. RoPE — how position is encoded in modern models — rotary position embedding, rotating query and key vectors by an angle proportional to position.
7. Needle in a haystack — the simple long-context test — planting one verbatim fact and asking for it back; necessary but far from sufficient.

47.14 Putting it together: one prompt from text to numbers

■ 47.14.1 PLAIN – in simple words

1. Everything in this chapter meets in one short journey.
2. You type a sentence. It is characters.
3. The tokenizer chops it into pieces and looks each piece up, giving integers.
4. Each integer picks one row out of the embedding table, giving a list of numbers.
5. Those lists are stacked into a block, one row per token.
6. The block goes into layer 1, comes out changed, goes into layer 2, and so on.
7. Each layer transforms the block using its own frozen parameters.
8. At the end, the row belonging to the last token is turned into one score for every possible next token.
9. The scores are turned into probabilities, one is picked, and that becomes the next token.
10. Then the whole thing repeats with that token added on the end.
11. Nothing in the model changed at any point. Only the numbers flowing through.

■ 47.14.2 PLAIN – a picture in your head

1. Picture a factory line with 32 stations.
2. At the entrance, a machine cuts the incoming text into standard-sized parts and stamps each with a part number.
3. A stores clerk takes each part number and fetches the matching tray of 4,096 components.
4. The trays travel down the line. Every station reworks all the trays a little, using its own fixed jigs and tools.
5. At the last station, the final tray is compared against a catalogue of 128,256 possible products, and the closest matches are scored.
6. One is chosen and shipped. Then the whole line runs again with that product added to the input.

Where this comparison breaks: on a factory line each station works on one item at a time. In a transformer every station works on all positions at once, and crucially each position can look at the others, which no factory station does.

■ 47.14.3 PLAIN – a worked example

1. The prompt: “The cat sat on the mat.” Model: an 8B-class model with hidden size 4,096, 32 layers, vocabulary 128,256.

```
stage 0  raw text
        "The cat sat on the mat."      23 characters

stage 1  tokenize (cl100k_base)
        7 tokens
        791, 8415, 7731, 389, 279, 5634, 13
        "The" " cat" " sat" " on" " the" " mat" "."
        shape: [7]
```

```

stage 2  embedding lookup
         fetch rows 791, 8415, ... from a
         [128256, 4096] table
         shape: [7, 4096]   = 28,672 numbers

stage 3  through layer 1 ... layer 32
         shape stays [7, 4096] at every layer boundary
         inside a layer it widens to [7, 14336] in the
         feed-forward middle and comes back

stage 4  final normalization
         shape: [7, 4096]

stage 5  take the last row only, position 6
         shape: [4096]

stage 6  multiply by the output matrix [128256, 4096]
         shape: [128256]       one logit per token

stage 7  softmax -> probabilities, then sample
         one token id comes out, say 13 for "."

stage 8  append and repeat from stage 2 with 8 tokens

```

2. Real sizes at each stage, at 16-bit: stage 2 holds 28,672 numbers, which is 56 KiB. Stage 6 produces 128,256 numbers, which is 251 KiB.
3. The key-value cache after this prompt holds 7 tokens $\times$ 128 KiB = 896 KiB.
4. Every one of those numbers is an activation. Every number in the tables they passed through is a parameter.

■ 47.14.4 PLAIN – what is really happening inside

1. Notice the two places where the vocabulary size appears.
2. Once at the entrance, as a table you index into. Once at the exit, as a table you multiply against.
3. Those two are the same shape and are sometimes literally the same numbers.
4. Notice that the shape [7, 4096] is stable through the whole stack.
5. That stable shape is called the residual stream. Every layer reads from it and writes back into it by adding.
6. Notice that only the last row is used to predict the next token, during generation. The other rows were computed anyway, and their keys and values are cached so they do not need recomputing next time.
7. Notice what the second pass costs. It processes one new token, not eight. All previous work is in the cache.
8. That is why the first token of a reply is slower than the rest: the first one pays for the whole prompt, and each later one pays for a single token.

■ 47.14.5 TECHNICAL – the engineer's version

1. The shapes with a batch dimension, which is how it really runs:

Stage	Tensor shape	dtype
Token ids	[B, S]	int64
After embedding	[B, S, 4096]	bf16

Stage	Tensor shape	dtype
FFN intermediate	[B, S, 14336]	bf16
Logits	[B, S, 128256]	fp32

- Logits are usually computed in fp32 even in a bf16 model, because softmax over 128,256 entries is numerically sensitive.
- During prefill, S is the full prompt length and all positions are computed. During decode, S is 1 and only the new position is computed, with the cache supplying the rest.
- Per-token generation cost in floating point operations is approximately $2 \times N$, where N is the parameter count, ignoring attention over the cache. For an 8B model that is about 16 billion operations per token.
- Prompt processing cost is approximately $2 \times N \times S$, so a 6,150-token prompt on an 8B model is about 98 trillion operations, which is why prefill is compute-bound and decode is bandwidth-bound.
- Memory traffic per decoded token at batch 1 is roughly the entire weight file, because every weight is read once. At 4-bit that is about 4.5 GiB per token for an 8B model, so a machine with 100 GB/s of memory bandwidth caps out near 21 tokens per second regardless of how fast its arithmetic is.
- That single fact explains most local inference performance questions, and it is why quantization speeds up generation as well as shrinking it.
- Inspecting the journey in practice: `tokenizer(text, return_tensors="pt")` gives stage 1; `model.get_input_embeddings()(ids)` gives stage 2; `model(ids, output_hidden_states=True).hidden_states` gives every stage 3 boundary as a tuple of 33 tensors for a 32-layer model.

■ 47.14.6 WORDS - remember these

- Residual stream — the stable block of numbers every layer reads and writes — the [B, S, d] tensor threaded through the network by residual connections.
- Prefill — processing the prompt — one parallel forward pass over all input positions, filling the key-value cache.
- Decode — producing one token — a forward pass with sequence length 1 reusing the cache.
- Logit — a raw score for one possible next token — one entry of the [V] output vector before softmax.
- Softmax — turning scores into probabilities — exponentiate and divide by the sum, giving a distribution over the vocabulary.
- Time to first token — how long before the reply starts — the latency of prefill, growing with prompt length.
- Memory bandwidth bound — limited by moving numbers, not by computing — the regime single-stream decoding runs in, where tokens per second is set by bytes read per token.

47.98 Common wrong ideas

- Wrong: more parameters means a smarter model. Right: parameter count sets a ceiling and predicts memory and speed very well, but Chinchilla at 70B beat Megatron-Turing NLG at 530B, and Mistral 7B beat Llama 2 13B. Data quantity, data quality, training compute and post-training all matter as much or more.
- Wrong: a token is a word. Right: a token is a subword piece. English averages about 1.1 to 1.3 tokens per word, JSON averages over 3, and a Hindi sentence can cost 24 tokens where the English equivalent costs 5.

3. Wrong: the context window is the model's memory, and a model that accepts 1,000,000 tokens uses all of them equally. Right: it is the input size limit for one request, nothing persists between requests, any apparent memory is your application resending the history, and accuracy dips for content in the middle, with RULER finding only about half of models claiming 32,000 tokens performed well at that length.
4. Wrong: quantization makes a model dumber in proportion to the bits removed. Right: the curve is very flat then steep. Going 16-bit to 8-bit cost 0.01 perplexity on a measured Llama-3.1-8B test; 16-bit to Q4_K_M cost 0.24; going to Q3_K_S cost 1.64. The first two thirds of the saving is nearly free.
5. Wrong: embeddings understand meaning. Right: they encode statistical co-occurrence as position in a space. That is enough for similarity to be useful and is not understanding. Two contradictory documents about the same fact sit close together, because they are about the same thing.
6. Wrong: hyperparameters are just small parameters. Right: hyperparameters are chosen by a human before training and are never learned. Parameters are learned by gradient descent and are never chosen.
7. Wrong: a 7B model needs 14 GB to train. Right: 14 GB is inference at 16-bit. Full training with Adam needs about 18 bytes per parameter for states alone, which is 117 GiB for 7B, before activations.
8. Wrong: king minus man plus woman equals queen proves embeddings capture relationships cleanly. Right: it is a real but weak effect, and the standard evaluation forbids returning the input words. With that exclusion removed, 3CosAdd accuracy fell from 71 per cent to 21 per cent.
9. Wrong: the model cannot count letters because it is stupid. Right: it receives "strawberry" as three integer ids and never sees individual letters. It answers from indirect knowledge, which is why it succeeds sometimes and fails unpredictably.
10. Wrong: input and output tokens cost the same. Right: output is typically priced four to six times higher, because generating is sequential and bandwidth-bound while reading the prompt is parallel and compute-bound.

47.99 Chapter summary in 20 lines

1. A parameter is one learned number sitting in one position of one matrix, set during training and frozen thereafter; "weight" and "parameter" are used interchangeably, with biases being additive parameters rather than weights.
2. A model's parameter count is the sum of the sizes of all its matrices, and it is computable by hand from five published shape numbers.
3. Worked exactly, Llama 3.1 8B holds 8,030,261,248 parameters: 32 layers of 218,112,000 plus two 525,336,576 vocabulary tables plus a final norm.
4. The billions live in the feed-forward blocks: 70 per cent for 8B, 80 per cent for 70B, 81 per cent for 405B, with attention near 17 per cent throughout.
5. Hyperparameters are pre-training human choices, activations are temporary numbers created while running, context is the input text, tokens are the pieces it is chopped into. Only parameters are in the model file.
6. Memory equals parameter count times bytes per parameter: 7B is 26.1 GiB at fp32, 13.0 at fp16, 6.5 at int8 and 3.3 at int4.
7. Inference needs more than the weights: a key-value cache of 320 KiB per token for a 70B model, plus activations and framework overhead.
8. Training needs about 18 bytes per parameter for weights, master copy, gradients and Adam's two moments, so a 7B model needs about 117 GiB of state before activations are counted.

9. Quantization stores each weight in fewer bits using a shared scale and optional zero point, per tensor, per channel or per group of 32 to 256.
10. GPTQ, AWQ, NF4 and the GGUF k-quants are the common formats; Q4_K is exactly 4.5 bits per weight, and Q4_K_M averages about 4.8 because sensitive tensors are kept at Q6_K.
11. A 70B model at Q4_K_M needs about 39.7 GiB of weights and fits a 64 GiB machine at a modest context; at fp16 it needs 131.4 GiB and does not.
12. Capability does not scale linearly with parameters: Chinchilla in 2022 showed models were undertrained, recommended about 20 tokens per parameter, and a 70B model beat a 530B one at equal compute.
13. Text is tokenized into subwords because whole words give an unbounded vocabulary and characters give sequences that are far too long.
14. Byte-pair encoding repeatedly merges the most frequent adjacent pair; on a four-word corpus, ten merges produce a 21-token vocabulary in which “newest” is a single token and “est_” emerges as a suffix without anyone naming it.
15. WordPiece merges by likelihood gain, Unigram prunes down from a large candidate set, and SentencePiece is the framework that makes both work without language-specific pre-tokenization.
16. English runs about 4 characters or 1.1 to 1.3 tokens per word; JSON is far worse; and a Hindi sentence costing 5 tokens in English cost 24 on the GPT-4-era tokenizer and 6 on the later one, which is real, measurable cost and context inequality.
17. Models cannot reliably count letters, reverse words or do long arithmetic because they never see letters or aligned digits, only token ids.
18. An embedding turns a token id into a row of a $[V, d]$ table; the numbers mean nothing individually and everything collectively, and 4,096 dimensions cannot be visualized, with distance concentration and near-orthogonality breaking most 2D intuitions.
19. Similarity is measured by cosine of the angle; the king minus man plus woman result is real but much weaker than told, since removing the input-word exclusion dropped 3CosAdd accuracy from 71 to 21 per cent.
20. The context window is limited because attention cost grows with the square of length; exceeding it truncates or errors; and accepting a long context is not the same as using it well, as the lost-in-the-middle and RULER results show.

The Transformer

48.0 What this chapter gives you

1. You will be able to say what came before the transformer, why it worked, and the exact reason the field abandoned it in about eighteen months.
2. You will be able to explain attention in ordinary words to somebody who has never seen a formula, using a sentence whose meaning flips on one adjective.
3. You will be able to write the scaled dot-product attention formula from memory and say what every symbol in it does.
4. You will be able to compute a full attention pass by hand on a four-token sentence and get numbers that match a real implementation.
5. You will be able to explain the causal mask, draw it as a matrix, and say what changes when a model does not use one.
6. You will be able to explain multi-head attention, work out the tensor shapes for a real model configuration, and say honestly what heads do and do not do.
7. You will be able to explain why attention has no sense of order at all, and describe sinusoidal, learned, ALiBi and rotary position encoding.
8. You will be able to draw a whole transformer block from memory, with residual connections and normalization in the right places, and say why.
9. You will be able to trace one prompt through a real model as a series of tensor shapes, from token ids to a probability over the vocabulary.
10. You will be able to explain the quadratic cost of attention, compute a real key-value cache size, and name the fixes that are in production today.

48.1 What came before, and why it was not enough

■ 48.1.1 PLAIN – in simple words

1. Before 2017, the standard way to handle a sentence was to read it one word at a time, left to right, the way a person reads aloud.
2. The machine that did this is called a **recurrent neural network**, usually shortened to RNN. Recurrent just means it feeds its own output back into itself.
3. It kept a single running summary of everything it had read so far. That summary is called the **hidden state**. It is a list of numbers.
4. Read word one, update the summary. Read word two, update the summary again. Keep going to the end.
5. That is the whole design. One summary, updated once per word, in order.
6. It worked. It was the best thing anybody had for sequences for about twenty-five years.

7. It had two problems, and the second one killed it.
8. Problem one: information from early words faded. By word two hundred, word three had almost no influence left on the summary.
9. Problem two, and this is the one that mattered: you cannot do the work for word ten until you have finished the work for word nine.
10. That is a rule you cannot break, because word ten literally needs the summary that word nine produced.
11. A modern graphics chip has tens of thousands of small calculators that all want work at the same time.
12. A design that insists on doing one word, then the next, then the next, cannot feed those calculators. They sit idle.
13. So training was slow, and it stayed slow no matter how much hardware you bought.
14. The transformer's central idea is to throw away the left-to-right rule and let every word look at every other word at once.
15. Once you do that, all the words can be processed in parallel, and the machine finally has enough work to do.

■ 48.1.2 PLAIN – a picture in your head

1. Imagine you are told a long story down a telephone line, one word at a time, and you are not allowed to write anything down.
2. You are allowed to keep exactly one small notepad in your head with, say, twenty facts on it. That is the hidden state.
3. Every new word arrives and you must decide what to erase to make room.
4. At word ten you still remember the beginning clearly.
5. At word two hundred the beginning has been overwritten many times. It is not gone in principle, but almost nothing of it survives.
6. Now imagine the person telling the story is speaking as fast as you can process. You cannot skip ahead. You cannot ask a friend to handle the second half while you do the first.
7. The story is strictly serial. That is the cost.
8. The transformer is the opposite arrangement. The whole story is printed on a sheet of paper in front of you, all at once, and you may look at any word whenever you want.

Where this comparison breaks: a real hidden state is not a list of twenty readable facts. It is a vector of a few hundred to a few thousand numbers with no individually meaningful entries. And nothing “decides what to erase” in a deliberate way. A learned matrix multiplication and a gate value between 0 and 1 produce that effect, and the effect looks like forgetting from outside.

■ 48.1.3 PLAIN – a worked example

1. Here is why the fading happens, with real arithmetic.
2. In training, the correction signal for word 3 has to travel backwards from word 200 through 197 update steps.
3. At each step it gets multiplied by a number. Call that number the shrink factor. If it is less than 1, the signal shrinks.
4. Here is what repeated multiplication does:

Shrink factor	After 10 steps	After 100 steps
0.5	0.00098	7.9e-31

Shrink factor	After 10 steps	After 100 steps
0.8	0.107	2.0e-10
0.9	0.349	2.7e-05
0.99	0.904	0.366

- At a shrink factor of 0.9, after 100 steps the signal is 0.0000266 of its original size. It has effectively vanished.
- That is the **vanishing gradient problem**, and over a sequence it is vanishing over time steps rather than over layers.
- Now the opposite. If the factor is 1.1 instead of 0.9, after 100 steps the signal is 13,780 times bigger. Training explodes and produces nonsense.
- So you need the factor to sit very near 1.0 for hundreds of steps in a row, which is a very hard thing to arrange by accident.
- Now the parallel problem, also with real arithmetic.
- Take a sequence of 1,000 tokens.
- An RNN needs 1,000 steps that must happen one after another. Nothing about the sequence can be done out of order.
- A transformer needs 1 step for the whole sequence, because the operation is one big matrix multiplication over all 1,000 positions at once.
- That is not a small speed-up. It is the difference between a chip running at a few per cent of its capacity and a chip running near its limit.

■ 48.1.4 PLAIN - what is really happening inside

- The RNN update is one line: new summary equals a function of the old summary and the new word.
- Written plainly: $h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$.
- h_t is the summary after word t . x_t is the word. W_h and W_x are learned matrices. $\tanh$ squashes everything into the range -1 to +1.
- Notice that h_{t-1} appears on the right-hand side. That is the whole dependency chain, and it is unbreakable.
- To learn, you unroll this into a chain 200 links long and push the error backwards along it, which is the repeated multiplication just shown.
- The **LSTM**, invented by Sepp Hochreiter and Jürgen Schmidhuber in 1997, was the fix for the fading.
- It added a second track, called the cell state, that flows forward with addition rather than repeated multiplication.
- Addition does not shrink a signal, so the correction can travel a long way without dying.
- It also added **gates**: small learned circuits that output a number between 0 and 1 for each position, deciding how much to keep, how much to let in and how much to output.
- The **GRU**, from Kyunghyun Cho and colleagues in 2014, is a simpler version with two gates instead of three, and usually performs about the same.
- These fixes worked well enough for real products. Google Translate switched to an eight-layer LSTM system in September 2016.
- But gates fix the fading. They do not fix the serial rule. Step t still needs step $t-1$. That part is structural.
- Meanwhile a second problem appeared in translation. The standard design was an **encoder-decoder**: one RNN reads the whole source sentence into a single vector, and a second RNN writes

the translation out of it.

14. Everything in a fifty-word sentence had to fit through one fixed-size vector. That vector is a bottleneck, and it is the same size whether the sentence has five words or fifty.
15. In September 2014, Dzmitry Bahdanau, Kyunghyun Cho and Yoshua Bengio published a fix. Instead of one vector, keep all the encoder's per-word states, and let the decoder look back and pick out the relevant ones.
16. That "look back and weight what matters" step is attention. It was invented as an accessory bolted onto an RNN.
17. The 2017 transformer paper's move was to delete the RNN and keep only the accessory.

■ 48.1.5 TECHNICAL – the engineer's version

1. The vanishing and exploding gradient problem in recurrent networks was identified by Sepp Hochreiter in his 1991 diploma thesis at the Technical University of Munich, and analysed formally by Yoshua Bengio, Patrice Simard and Paolo Frasconi in 1994.
2. The mechanism is the chain rule applied through time. The gradient of the loss at step T with respect to the state at step t contains the product of Jacobians from t to T .
3. If the largest singular value of the recurrent Jacobian is consistently below 1, that product decays geometrically. Above 1, it grows geometrically.
4. With tanh activation the derivative is at most 1.0 and typically well below it, so decay is the common case. With sigmoid gates the derivative peaks at 0.25.
5. Exploding gradients are cheaply fixed by gradient clipping, which rescales the gradient vector when its norm passes a threshold. Vanishing gradients are not fixable that way, because you cannot amplify information that is gone.
6. LSTM, from Hochreiter and Schmidhuber, Neural Computation volume 9 issue 8, 1997, introduced the constant error carousel: an additive cell state path whose local derivative is close to 1.
7. The forget gate was not in the original LSTM. Felix Gers, Jürgen Schmidhuber and Fred Cummins added it in 1999 and 2000, and the version with the forget gate is what everybody now means by "LSTM".
8. Parameter counts per unit differ: a vanilla RNN cell has one weight matrix pair, an LSTM has four, and a GRU has three.

Model	Year	Gates	Sequential steps
Elman RNN	1990	0	n
LSTM	1997	3	n
GRU	2014	2	n
Transformer	2017	none	1

9. Sequence-to-sequence learning with a fixed context vector was published by Ilya Sutskever, Oriol Vinyals and Quoc Le in September 2014, using a 4-layer LSTM with 1,000 units per layer and a 1,000-dimensional context vector.
10. Additive attention, also called Bahdanau attention, was published the same month by Bahdanau, Cho and Bengio in the paper "Neural Machine Translation by Jointly Learning to Align and Translate". It scores each encoder state with a small feed-forward network and softmaxes the scores.
11. Multiplicative attention, from Minh-Thang Luong, Hieu Pham and Christopher Manning in 2015, replaced that small network with a dot product. That is the direct ancestor of the 2017 formula.

12. Google Neural Machine Translation, deployed in September 2016, used 8 encoder and 8 decoder LSTM layers with attention, residual connections and model parallelism across 8 GPUs. It cut translation errors substantially and was still fundamentally serial.
13. The training-time argument is about arithmetic intensity. An RNN step at batch size B computes a matrix times a matrix of shape $[B, d]$ by $[d, d]$. With $B = 32$ and $d = 1024$ that is 67 MFLOP, then a synchronization, then the next step.
14. A transformer layer at sequence length 1,000 computes $[1000, d]$ by $[d, d]$, which at $d = 4096$ is 33.6 GFLOP in a single call, with no dependency inside the call.
15. Modern accelerators need large matrix-matrix operations to reach peak. This is covered in Chapter 22, which explains why a GPU is a wide parallel machine rather than a fast serial one. Long thin operations leave most of the chip idle.
16. This is established fact and was stated openly in the abstract of the 2017 paper: the models are “more parallelizable and requiring significantly less time to train”.

■ 48.1.6 WORDS - remember these

1. Recurrent neural network — a network that reads one item at a time and keeps a running summary — a network with a cyclic dependency $h_t = f(h_{t-1}, x_t)$, unrolled over time for training.
2. Hidden state — the running summary — the vector carried from one time step to the next, of size equal to the layer width.
3. Vanishing gradient — the correction signal dying out over distance — the geometric decay of a product of Jacobians whose singular values are below 1.
4. LSTM — a recurrent cell with gates that remembers longer — long short-term memory, with input, forget and output gates and an additive cell state.
5. GRU — a simpler gated cell — gated recurrent unit, with update and reset gates and no separate cell state.
6. Encoder-decoder — one network reads, another writes — a sequence-to-sequence architecture with a context representation passed between them.
7. Bottleneck vector — squeezing a whole sentence into one fixed list of numbers — the fixed-dimension context vector in pre-attention seq2seq.
8. Attention — looking back and weighting what matters — a learned weighted average over a set of value vectors, with weights from a scoring function.

48.2 Attention in plain language, before any mathematics

■ 48.2.1 PLAIN - in simple words

1. Read this sentence carefully: “the trophy did not fit in the suitcase because it was too big”.
2. What does “it” mean here? The trophy.
3. You did not need a rule book. You worked it out.
4. Now read the same sentence with one word changed: “the trophy did not fit in the suitcase because it was too small”.
5. What does “it” mean now? The suitcase.
6. The sentence is identical except for the last word, and the meaning of “it” flipped completely.
7. Stop and notice what you just did. To understand the word “it”, you looked back at other words in the sentence.
8. You did not look back at all of them equally. “the”, “did” and “in” contributed almost nothing.
9. You looked hard at “trophy”, “suitcase”, “fit” and the final adjective.
10. Then you combined what you found and produced a meaning for “it”.

11. That is attention. Look back, weight some things more than others, combine.
12. The model does exactly this, for every word, at every layer, with the weights worked out by learned matrices rather than by a person.
13. There is one important extra bit. The model does this for every word at once, not just for the confusing ones.
14. So while it is deciding what “it” points to, it is simultaneously deciding what “big” applies to and what “fit” takes as its subject.
15. And it does not know in advance which words are the confusing ones. Every position gets the same treatment.

■ 48.2.2 PLAIN – a picture in your head

1. Picture a meeting where a decision has to be made, and eight people are in the room.
2. You are the chairperson. You ask a question.
3. Everybody in the room has something to say. Some of it is highly relevant. Most of it is not.
4. You do not take a plain average of all eight opinions, because that would drown the useful answer in noise.
5. Instead you listen to each person, decide how much their answer bears on your question, and give each one a share of your final decision.
6. Person 3 gets 60 per cent of your attention because they clearly know. Person 7 gets 25 per cent. The other six split the remaining 15 per cent.
7. Your conclusion is a blend, weighted by relevance.
8. That is one attention operation. Your question is the query. Each person’s relevance is a weight. Each person’s opinion is a value.
9. Now the part that makes it a transformer: every person in the room is doing this at the same time, each with their own question, including you.

Where this comparison breaks: in a real meeting, relevance is judged by understanding. In attention, relevance is a dot product between two learned vectors, which is a geometric measure of alignment. It correlates with what we would call relevance because training pushed it that way, and it has no independent notion of what the question is about. Also, the weights always add up to exactly 1, so a position cannot decide that nothing in the sentence is relevant. It must spread its 1.0 somewhere, and it often dumps it on a token that carries no meaning, which is a real and measured effect.

■ 48.2.3 PLAIN – a worked example

1. Take the sentence with “big”, and consider the position holding “it”.
2. Here is roughly how a trained model distributes attention from “it”, based on what published attention maps for coreference look like. These are illustrative figures to show the shape, not a measurement of one named model.

Sentence: the trophy did not fit in the suitcase
because it was too big

Attention from the position holding "it":

trophy	#####	0.41
suitcase	#####	0.18
big	#####	0.15
fit	#####	0.09
it	#####	0.08
the	##	0.03
other	###	0.06

```

-----
1.00

```

3. Now swap “big” for “small”. The same position, in the same model, shifts:

```

Sentence: the trophy did not fit in the suitcase
          because it was too small

```

```

suitcase ##### 0.39
trophy   ##### 0.17
small    ##### 0.16
fit      ##### 0.10
it       ##### 0.08
the      ##    0.03
other    #### 0.07
-----
1.00

```

- One word changed at the end of the sentence, and the top weight moved from “trophy” to “suitcase”.
- This is the point. The weights are not fixed by the identity of the word “it”. They are computed fresh from the actual contents of this sentence.
- That is why the same word gets a different representation in different sentences, which is the thing older methods could not do.
- Chapter 47 covered embeddings, which are the fixed starting vectors for tokens. The embedding for “it” is one single vector, the same every time.
- Attention is what turns that one fixed vector into a different vector in every different context. That is the entire job.
- This sentence pair is not invented for this book. It is a **Winograd schema**, a test format proposed by Hector Levesque, Ernest Davis and Leora Morgenstern in 2011, named after Terry Winograd, who used an example of this shape in 1972.
- The point of the format is that no amount of word-frequency statistics solves it. You have to model the situation being described.

■ 48.2.4 PLAIN – what is really happening inside

- Every token position holds a vector of numbers. Call it the position’s current understanding of itself.
- Attention runs in three moves, and every position does all three.
- Move one: each position emits a **query**, meaning “here is what I am looking for”. It is a vector, computed from the position’s own current vector.
- Move two: each position also emits a **key**, meaning “here is what I am, if anybody is looking”. Also a vector, also computed from the same input.
- Move three: each position emits a **value**, meaning “here is what I will contribute if you pick me”.
- Now the matching. Position 10’s query is compared against every key in the sentence, including its own.
- The comparison is a dot product: multiply the two vectors element by element and add up. A big result means the two vectors point in a similar direction.
- That gives one raw score per position. For a 12-token sentence, position 10 gets 12 scores.
- Those 12 scores go through softmax, which turns any list of numbers into positive numbers that add to exactly 1.
- Now they are weights. 0.41 on “trophy”, 0.18 on “suitcase”, and so on.

11. Finally, take every position's value vector, multiply it by that position's weight, and add them all up.
12. The result is position 10's new vector. It now contains a blend of the whole sentence, weighted by what position 10 was looking for.
13. That new vector replaces the old one, and the next layer does the whole thing again with different learned matrices.
14. Repeat 32 times for a typical 8-billion-parameter model, and information from anywhere can reach anywhere, by many different routes.

■ 48.2.5 TECHNICAL - the engineer's version

1. The operation just described is **self-attention**: queries, keys and values are all derived from the same input sequence.
2. **Cross-attention** is the same operation where queries come from one sequence and keys and values from another. It is what connects the decoder to the encoder in a translation model.
3. Self-attention is a set operation, not a sequence operation. It is permutation-equivariant, which is proved numerically in section 48.7.
4. The pairwise score matrix has shape $[n, n]$ for a sequence of length n . Every position produces one row.
5. The distance between two tokens does not appear anywhere in the computation. Position 1 and position 4,000 are exactly as easy to connect as position 1 and position 2. That is the structural difference from an RNN, where the path length between positions i and j is $|i - j|$ update steps.
6. Attention weight maps are widely published and easy to produce with the `output_attentions=True` flag in Hugging Face transformers, or with the BertViz tool from Jesse Vig, 2019.
7. Two honest warnings about reading those maps.
8. **The honest version**: an attention weight is not an explanation. Sarthak Jain and Byron Wallace published "Attention is not Explanation" in 2019, showing that alternative attention distributions can produce identical predictions. Sarah Wiegrefe and Yuval Pinter replied with "Attention is not not Explanation" the same year, arguing the test was too strict. The dispute is not settled. Attention weights show where information flowed, not why.
9. Second warning: a large fraction of attention mass in trained models lands on the first token or on punctuation. This is called an **attention sink**, and was analysed by Guangxuan Xiao and colleagues in 2023 in the StreamingLLM work. The softmax must sum to 1, so a head with nothing to say has to put its weight somewhere, and it learns to park it on a fixed harmless position.
10. Established fact: attention computes content-dependent weighted averages, and this is measurable. Active research: what individual heads compute and whether their weights can be read as explanations. Marketing claim: any statement that the model "focuses on" or "pays attention to" something in the human sense.

■ 48.2.6 WORDS - remember these

1. Self-attention — every word looking at every word in the same sentence — attention where Q, K and V all derive from one input sequence.
2. Cross-attention — one sequence looking at a different sequence — attention where Q comes from the target and K, V from the source.
3. Query — what a position is looking for — a learned linear projection of the position's hidden state, of dimension d_k .
4. Key — what a position advertises about itself — a learned projection compared against queries by dot product, of dimension d_k .

5. Value — what a position contributes if selected — a learned projection that is averaged with the attention weights, of dimension d_v .
6. Softmax — turning any scores into shares that add to 1 — $\exp(x_i)$ divided by the sum of $\exp$ over all j , applied along the key axis.
7. Attention weight — how much one word listened to another — one entry of the row-stochastic $[n, n]$ matrix produced by softmax over scaled scores.
8. Attention sink — a position that soaks up leftover weight — a token, typically the first, that receives large attention mass with no semantic role, arising from the sum-to-one constraint.
9. Winograd schema — a sentence pair whose meaning flips on one word — a coreference test designed to be unsolvable by surface statistics.

48.3 Query, key and value, taught slowly

■ 48.3.1 PLAIN - in simple words

1. Three words do all the work in attention: query, key and value.
2. They sound abstract. They are not. They are three different views of the same token, made by three different learned recipes.
3. The **query** is the question this position is asking. “I am a pronoun. I need to know what noun I stand for.”
4. The **key** is the label this position wears. “I am a singular concrete noun, and I am the subject of the sentence.”
5. The **value** is the content this position offers up if selected. “Here is my actual meaning, ready to be copied into whoever picks me.”
6. Every single position produces all three, every time.
7. So a position is simultaneously asking a question, advertising an answer, and holding content to hand over.
8. The matching happens between queries and keys only. Values take no part in deciding who wins.
9. Once the weights are decided, the values are what actually get blended and passed on.
10. That separation is the useful part of the design. What makes something findable is allowed to be different from what it contains.
11. And the crucial fact people miss: all three come from the same starting vector. There is no separate store of keys and values somewhere.
12. Take one token’s vector, multiply it by matrix number 1 to get the query, matrix number 2 to get the key, matrix number 3 to get the value.
13. Those three matrices are learned during training and frozen afterwards. They are ordinary parameters, in the sense of Chapter 47.

■ 48.3.2 PLAIN - a picture in your head

1. Picture a library, but a strange one where you may only ask one question and must get one answer back immediately.
2. You walk in with a question written on a card. That is your **query**.
3. Every book in the library has a small index card clipped to its spine. The card says what the book is about, in a compressed form. That is the **key**.
4. The book’s actual pages are the **value**.
5. You do not read any books. You hold your question card next to every index card in turn and score how well they line up.
6. Book 3’s card scores highly. Book 7’s card scores moderately. Most cards score near zero.

7. Then you do something no real library allows. You do not pick one book. You take a weighted blend of every book's contents: 60 per cent of book 3, 25 per cent of book 7, and small slivers of the rest.
8. That blend is your answer.
9. The index card is deliberately not the book. A card can say "this is a cookbook about bread" while the book contains actual recipes.
10. That is the whole reason keys and values are separate. Findability and content are different jobs.

Where this comparison breaks: in a real library the index card was written by a librarian who read the book. Here both the card and the book come from the same raw vector through two different learned matrices, so they are not independent. Also, a library has a fixed catalogue. Here the keys are recomputed from scratch for every input, so the same word gets a different key in a different sentence. And a real search returns one book. Attention always returns a blend and can never return nothing, because the weights are forced to sum to 1.

■ 48.3.3 PLAIN – a worked example

1. Now the projection, with numbers small enough to check on paper.
2. We will use a four-token sentence: "the cat was hungry".
3. Each token arrives as a vector of 4 numbers. In a real model this would be 4,096 numbers, from the embedding table of Chapter 47. Here it is 4.
4. Stack the four token vectors as rows. That is the input matrix X, shape [4 tokens, 4 dimensions]:

X (4 tokens x 4 dimensions)

	d0	d1	d2	d3
the	1	0	1	0
cat	0	1	0	1
was	1	1	0	0
hungry	0	1	1	0

5. **The honest version:** real embedding values are floating-point numbers like -0.0113 or 0.0392, never neat integers. Integers are used here only so you can verify every multiplication by hand.
6. Now the three learned matrices. Each is shape [4, 3], because we are choosing to project from 4 dimensions down to 3.

W_Q (4x3)	W_K (4x3)	W_V (4x3)
1 0 0	1 0 0	2 0 1
1 1 2	1 0 2	0 2 0
0 1 0	1 1 1	1 1 2
1 1 0	0 1 2	0 1 1

7. To get the query for "hungry", take the row for "hungry", which is [0, 1, 1, 0], and multiply it by W_Q.
8. Doing it column by column:
9. Column 0: $(0 \times 1) + (1 \times 1) + (1 \times 0) + (0 \times 1) = 1$.
10. Column 1: $(0 \times 0) + (1 \times 1) + (1 \times 1) + (0 \times 1) = 2$.
11. Column 2: $(0 \times 0) + (1 \times 2) + (1 \times 0) + (0 \times 0) = 2$.
12. So the query for "hungry" is [1, 2, 2].
13. Do the same for every token and every matrix. The three results:

Q = X W_Q	K = X W_K	V = X W_V
the 1 1 0	the 2 1 1	the 3 1 3
cat 2 2 2	cat 1 1 4	cat 0 3 1

was	2	1	2	was	2	0	2	was	2	2	1
hungry	1	2	2	hungry	2	1	3	hungry	1	3	2

14. Check one more by hand. The key for “cat” is row $[0, 1, 0, 1]$ times W_K .
15. Column 0: $0 + 1 + 0 + 0 = 1$. Column 1: $0 + 0 + 0 + 1 = 1$. Column 2: $0 + 2 + 0 + 2 = 4$. So $[1, 1, 4]$. That matches the table.
16. Notice the shapes. X is $[4, 4]$. Each W is $[4, 3]$. Each of Q, K, V is $[4, 3]$.
17. The number of tokens never changes. Only the width changes.

■ 48.3.4 PLAIN - what is really happening inside

1. There is no lookup, no dictionary, no search index. There are three matrix multiplications.
2. A matrix multiplication of a $[4, 4]$ by a $[4, 3]$ is just 12 dot products, one per output cell, which is exactly what was done above by hand.
3. The words “query”, “key” and “value” are borrowed from database retrieval. They are a naming choice, not a mechanism.
4. Nothing enforces that W_Q learns “questions” and W_K learns “labels”. Those are the roles they end up playing because of where they sit in the formula.
5. W_Q and W_K only ever appear together, inside a dot product. Their product $Q K$ -transpose is what the model actually uses.
6. That means the pair behaves as a single learned bilinear form. You could fold them into one matrix and get identical scores.
7. They are kept separate because it costs far fewer parameters. Two matrices of $[d, d_k]$ cost $2 d d_k$ parameters, while one full $[d, d]$ matrix costs d squared. With $d = 4096$ and $d_k = 128$ that is 1,048,576 against 16,777,216.
8. W_V is genuinely separate, because its output does not go into any score. It goes into the answer.
9. In code, all three projections for a whole batch are usually done as one matrix multiplication with the three weight matrices concatenated, then split afterwards. That is an implementation detail for speed, not part of the idea.
10. On the hardware side, these are exactly the dense matrix multiplications Chapter 22 described as the operation a GPU is built to do. The whole sequence goes through in one call.

■ 48.3.5 TECHNICAL - the engineer’s version

1. The projections are three linear maps with no bias in most modern models: $Q = X W_Q$, $K = X W_K$, $V = X W_V$.
2. Shapes for one attention layer, with input X of shape $[B, n, d_{\text{model}}]$:

Tensor	Shape	Notes
X	$[B, n, d_{\text{model}}]$	layer input
W_Q	$[d_{\text{model}}, h \times d_k]$	learned
W_K	$[d_{\text{model}}, h_{\text{kv}} \times d_k]$	learned
W_V	$[d_{\text{model}}, h_{\text{kv}} \times d_v]$	learned

3. In the 2017 base model, $d_{\text{model}} = 512$, $h = 8$, $d_k = d_v = 64$, so each of the four projection matrices is 512 by 512 and holds 262,144 parameters.
4. In Llama 3.1 8B, $d_{\text{model}} = 4096$, 32 query heads, 8 key-value heads, head dimension 128. So q_{proj} is $[4096, 4096]$, while k_{proj} and v_{proj} are $[1024, 4096]$. That asymmetry is grouped-query attention, covered in 48.13.

5. Bias terms: GPT-2 uses biases on these projections. The Llama family, Mistral and Qwen do not. This is an implementation detail with negligible effect on quality and a small effect on parameter count.
6. Real parameter names in a checkpoint, which you can list without loading the model using the safetensors header:

```
model.layers.0.self_attn.q_proj.weight [4096, 4096]
model.layers.0.self_attn.k_proj.weight [1024, 4096]
model.layers.0.self_attn.v_proj.weight [1024, 4096]
model.layers.0.self_attn.o_proj.weight [4096, 4096]
```

7. Note the convention: PyTorch stores a linear layer's weight as [out_features, in_features] and computes x times W -transpose. So the printed shape is the transpose of the mathematical W in the formula. This trips up almost everybody once.
8. The low-rank factorization argument is exact. Scores depend only on the product $W_Q W_K$ -transpose, which has rank at most d_k . Choosing $d_k = 128$ with $d_{\text{model}} = 4096$ imposes a rank-128 constraint on the score matrix, and that constraint is the parameter saving.
9. Whether that rank limit hurts has been studied directly. Srinadh Bhojanapalli and colleagues at Google Research published "Low-Rank Bottleneck in Multi-head Attention Models" in 2020, showing that tying d_k to d_{model} divided by the number of heads does constrain what a head can express, and that decoupling the two can help.

■ 48.3.6 WORDS - remember these

1. Projection — turning a vector into a different-shaped vector — multiplication by a learned matrix, usually without a bias term.
2. Linear layer — a matrix multiply, optionally plus a bias — `nn.Linear` in PyTorch, storing weight as [out_features, in_features].
3. d_{model} — the width of the main pipe through the model — the residual stream dimension, 512 in the 2017 base model, 4096 in Llama 3.1 8B.
4. d_k — the width of one head's queries and keys — the per-head key dimension, 64 in the 2017 paper, 128 in most current models.
5. Bilinear form — one matrix sandwiched between two vectors — the effective scoring matrix $W_Q W_K$ -transpose, of rank at most d_k .
6. Low-rank factorization — writing a big matrix as two thin ones — replacing a $[d, d]$ matrix with $[d, r]$ times $[r, d]$, costing $2dr$ instead of d squared.

48.4 Scaled dot-product attention, with the actual mathematics

■ 48.4.1 PLAIN - in simple words

1. Here is the whole formula. It is one line, and it is the most quoted line in modern machine learning.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{Q K^T}{\sqrt{d_k}}\right) V$$
2. Read it right to left in four steps.
3. Step one: $Q K^T$. Compare every query with every key by dot product. Output is a square grid of scores, one row per token, one column per token.
4. Step two: divide by the square root of d_k , the width of a key. This is a fixed number, not learned. It stops the scores from getting too large.
5. Step three: softmax each row. Each row of scores becomes a row of weights that are all positive and add up to exactly 1.

- Step four: multiply by V. Each row of the output is a weighted average of all the value vectors, using that row's weights.
- That is it. Four steps, three of which are matrix multiplications and one of which is a division.
- The result has the same number of rows as the input. One output vector per token, exactly as many as went in.
- Everything else in this chapter is either a variation on this line, a way to make it faster, or a way to fix something it cannot do on its own.

■ 48.4.2 PLAIN – a picture in your head

- Picture a school report card, but sideways.
- Down the left are the students asking questions. Across the top are the students offering answers. Same list of students on both sides.
- In each cell you write a number: how useful is that answer for that question. This grid is Q K-transpose.
- Now the trick. Nobody trusts raw scores, so each row is converted into percentages that add to 100. That is softmax.
- Row 4 might read: 3 per cent, 60 per cent, 3 per cent, 34 per cent. That is student 4's opinion of who to listen to.
- Finally each student writes down their answer on a card. Student 4 makes a mixture: 3 per cent of card 1, 60 per cent of card 2, and so on.
- What student 4 walks away with is a blend of the whole class, weighted by their own judgement of relevance.
- And every student does this at the same time, so the entire grid is filled in one operation.

Where this comparison breaks: the “percentages” from softmax are never exactly zero. Every student always gets a nonzero share, even an irrelevant one, because the exponential function never reaches zero. And “how useful is that answer” is not a judgement. It is the cosine-like alignment of two vectors, scaled by their lengths. Two vectors can align strongly for reasons that have nothing to do with meaning.

■ 48.4.3 PLAIN – a worked example

- This is the full computation, using the Q, K and V produced in section 48.3 for the sentence “the cat was hungry”. Every number below was computed and checked with numpy.
- Recall the matrices:

Q				K				V			
the	1	1	0	the	2	1	1	the	3	1	3
cat	2	2	2	cat	1	1	4	cat	0	3	1
was	2	1	2	was	2	0	2	was	2	2	1
hungry	1	2	2	hungry	2	1	3	hungry	1	3	2

- Step one: Q K-transpose. Every entry is one dot product.
- Row “hungry”, column “cat”: q_hungry is $[1, 2, 2]$, k_cat is $[1, 1, 4]$.
- $(1 \times 1) + (2 \times 1) + (2 \times 4) = 1 + 2 + 8 = 11$.
- Row “hungry”, column “the”: $[1, 2, 2]$ against $[2, 1, 1]$ gives $2 + 2 + 2 = 6$.
- Row “hungry”, column “was”: $[1, 2, 2]$ against $[2, 0, 2]$ gives $2 + 0 + 4 = 6$.
- Row “hungry”, column “hungry”: $[1, 2, 2]$ against $[2, 1, 3]$ gives $2 + 2 + 6 = 10$.
- Doing all sixteen gives the raw score matrix:

$S = Q K^T$	the	cat	was	hungry
the	3	2	2	3
cat	8	12	8	12
was	7	11	8	11
hungry	6	11	6	10

10. Step two: divide by $\sqrt{d_k}$. Here $d_k = 3$, and $\sqrt{3} = 1.7320508$.

$S / \sqrt{3}$	the	cat	was	hungry
the	1.7321	1.1547	1.1547	1.7321
cat	4.6188	6.9282	4.6188	6.9282
was	4.0415	6.3509	4.6188	6.3509
hungry	3.4641	6.3509	3.4641	5.7735

11. Step three: softmax each row. Take the row for "hungry": [3.4641, 6.3509, 3.4641, 5.7735].
 12. Exponentiate each: e to the 3.4641 = 31.9477, e to the 6.3509 = 572.9812, e to the 3.4641 = 31.9477, e to the 5.7735 = 321.6624.
 13. Add them: $31.9477 + 572.9812 + 31.9477 + 321.6624 = 958.5392$.
 14. Divide each by the total: $31.9477 / 958.5392 = 0.0333$, $572.9812 / 958.5392 = 0.5978$, $31.9477 / 958.5392 = 0.0333$, $321.6624 / 958.5392 = 0.3356$.
 15. Check: $0.0333 + 0.5978 + 0.0333 + 0.3356 = 1.0000$. Good.
 16. The full attention weight matrix, every row summing to 1:

A	the	cat	was	hungry
the	0.3202	0.1798	0.1798	0.3202
cat	0.0452	0.4548	0.0452	0.4548
was	0.0436	0.4393	0.0777	0.4393
hungry	0.0333	0.5978	0.0333	0.3356

17. Read that matrix as language. "hungry" puts 0.5978 of its weight on "cat". Whatever "hungry" describes, it is mostly getting it from "cat".
 18. "was" splits its weight between "cat" at 0.4393 and "hungry" at 0.4393, which is what a linking verb joining a subject to a predicate would do.
 19. "the" is nearly flat, which is what a determiner with no strong question would do.
 20. This is a toy with hand-chosen matrices. Do not conclude that this pattern is what a trained model produces. It shows the shape of the computation.
 21. Step four: multiply A by V. Take the "hungry" row again:

$0.0333 \times [3, 1, 3]$	=	$[0.1000, 0.0333, 0.1000]$	from "the"
$0.5978 \times [0, 3, 1]$	=	$[0.0000, 1.7933, 0.5978]$	from "cat"
$0.0333 \times [2, 2, 1]$	=	$[0.0667, 0.0667, 0.0333]$	from "was"
$0.3356 \times [1, 3, 2]$	=	$[0.3356, 1.0067, 0.6712]$	from "hungry"

sum =		$[0.5022, 2.9000, 1.4022]$	

22. The complete output matrix, one row per token:

Output = A V	dim0	dim1	dim2
the	1.6405	2.1798	1.9607
cat	0.6807	2.8645	1.5452
was	0.7257	2.8350	1.5266
hungry	0.5022	2.9000	1.4022

23. Shape check: A is [4, 4], V is [4, 3], so the output is [4, 3]. Four tokens in, four tokens out, width 3.

24. Notice that “hungry” came in as [0, 1, 1, 0] and leaves as [0.5022, 2.9000, 1.4022]. It is no longer a context-free word. It now carries mostly “cat”.

■ 48.4.4 PLAIN - what is really happening inside

1. Now the piece everybody skips: why divide by the square root of d_k .
2. A dot product adds up d_k separate products. The more terms you add, the larger the total tends to be, in both directions.
3. If the entries of q and k are independent, have mean 0 and variance 1, then each product $q_i k_i$ has mean 0 and variance 1.
4. Adding d_k independent things with variance 1 gives something with variance d_k , so the typical size grows like the square root of d_k .
5. Measured with 20,000 random pairs per dimension:

d_k	Variance of $q.k$	Std dev	$\text{sqrt}(d_k)$
4	4.00	2.00	2.00
64	63.54	7.97	8.00
512	505.98	22.49	22.63
4096	4120.58	64.19	64.00

6. The measured standard deviation tracks $\text{sqrt}(d_k)$ exactly. That is the whole argument.
7. Now why large scores are bad. Softmax exponentiates. A gap of 10 between two scores becomes a ratio of about 22,026 to 1 after exponentiating.
8. Here is a real case with $d_k = 64$ and four random keys:

```
raw scores      2.153 -7.355 -9.911 -4.582
softmax(raw)    0.9987 0.0001 0.0000 0.0012

after dividing by sqrt(64) = 8
scaled scores   0.2691 -0.9194 -1.2389 -0.5728
softmax(scaled) 0.5110 0.1557 0.1131 0.2202
```

9. Without scaling, one key gets 99.87 per cent of the weight and the rest get essentially nothing. The attention has become a hard selection.
10. That is bad for two reasons. It throws away information, and worse, it kills learning.
11. Softmax’s gradient for an output p is proportional to p times $(1 - p)$.
12. At $p = 0.5$ that is 0.25. At $p = 0.999$ it is 0.000999. At $p = 0.99999$ it is 0.00001.
13. So once softmax saturates, the correction signal flowing back through it is almost zero, and the model stops being able to adjust those scores.
14. Dividing by $\text{sqrt}(d_k)$ rescales the scores to variance about 1 regardless of head width, keeping softmax in its responsive range at the start of training.
15. The 2017 paper states this reason directly in a footnote, and it is the only place in the paper where the word “suspect” appears about a design choice.
16. One more subtlety. The scaling matters most at initialization, when weights really are near-random and the variance argument holds exactly. Later in training the weights are not random and the argument is approximate. It still works, and nobody has found a reason to change it.

■ 48.4.5 TECHNICAL - the engineer’s version

1. The canonical formula, from equation 1 of “Attention Is All You Need”: $\text{Attention}(Q, K, V) = \text{softmax}(QK^T / \text{sqrt}(d_k)) V$.

2. Shapes, single head, batch B, sequence n:

Step	Operation	Output shape
scores	Q times K^T	$[B, n, n]$
scaled	divide by $\sqrt{d_k}$	$[B, n, n]$
weights	softmax over last dim	$[B, n, n]$
output	weights times V	$[B, n, d_v]$

- Floating-point operations, forward pass, single head: $2 n^2 d_k$ for the score matrix and $2 n^2 d_v$ for the weighted sum, so $4 n^2 d$ in total when $d_k = d_v = d$.
- Softmax is implemented in the numerically stable form: subtract the row maximum before exponentiating. Without that, exp of a score above about 88 overflows in float32 and above about 11 in float16.
- That max-subtraction is mathematically a no-op, since softmax is invariant to adding a constant to all inputs. It is purely a numerical safety measure and it is universal.
- Attention scores are almost always computed in float32 even when the model runs in bfloat16, because the softmax denominator accumulates n terms and bfloat16 has only 8 bits of mantissa.
- In PyTorch 2.0 and later, the whole thing is one call: `torch.nn.functional.scaled_dot_product_attention(q, k, v, is_causal=True)`, which dispatches to a fused kernel such as FlashAttention when the shapes and dtypes allow it.
- The reference implementation, which is worth being able to write from memory:

```
import torch, math

def attention(q, k, v, mask=None):
    # q: [B, h, n, dk] k,v: [B, h, n, dk]
    dk = q.size(-1)
    scores = q @ k.transpose(-2, -1) / math.sqrt(dk)
    if mask is not None:
        scores = scores.masked_fill(mask, float("-inf"))
    w = torch.softmax(scores.float(), dim=-1).to(v.dtype)
    return w @ v
```

- Alternatives to the dot product exist and lost. Additive attention (Bahdanau) uses a one-hidden-layer network and is theoretically stronger at large d_k but far slower, because it cannot be expressed as a single matrix multiply. The 2017 paper reports it performs comparably at small d_k .
- Scaling variants exist. Some papers use $1/d_k$, some use a learned temperature. The $1/\sqrt{d_k}$ convention is universal in practice, and it is a convention, not a standard. Nothing enforces it.
- Query-key normalization, which normalizes Q and K to unit length before the dot product, appeared around 2020 and is used in some large-scale training runs to prevent logit growth. It changes the scaling argument but not the structure.

■ 48.4.6 WORDS – remember these

- Dot product — multiply matching entries and add them up — the inner product of two vectors, equal to $|a||b|\cos(\theta)$.
- Score matrix — the grid of how well every query matched every key — the $[n, n]$ tensor $Q K^T$ before scaling and softmax.
- Scaling factor — the fixed divisor that keeps scores small — $1/\sqrt{d_k}$, chosen so the score variance is about 1 at initialization.

4. Saturation — when softmax has already picked a winner and stops responding — the regime where one probability approaches 1 and the Jacobian term $p(1-p)$ approaches 0.
5. Numerically stable softmax — subtracting the biggest score first — the standard implementation $\exp(x - \max) / \sum \exp(x - \max)$, avoiding overflow.
6. Attention output — the blended result for each position — the $[n, d_v]$ matrix of weighted sums of value vectors.
7. Additive attention — the older, slower scoring method — Bahdanau's $v^T \tanh(W_1 q + W_2 k)$, a small network instead of a dot product.

48.5 Masking: stopping the model reading ahead

■ 48.5.1 PLAIN – in simple words

1. A language model is trained by hiding the next word and asking it to guess.
2. That only works if the model genuinely cannot see the next word.
3. But attention, as built so far, lets every position see every position, including the ones after it.
4. So during training on the sentence “the cat was hungry”, the position for “was” can see “hungry”, and predicting “hungry” becomes trivial copying.
5. The model would score perfectly during training and be useless at generation time, when there is nothing after the current word to copy.
6. The fix is called a **mask**. Before the softmax, you go into the score grid and destroy every score that looks forward.
7. Destroy means set to negative infinity, because e to the power of negative infinity is 0, so softmax gives that position a weight of exactly zero.
8. You cannot just set the weights to zero after the softmax, because then the remaining weights would not add to 1. Doing it before the softmax means the surviving weights renormalize automatically.
9. The shape of the destroyed region is the triangle above the diagonal, so this is called a **causal mask**, or a look-ahead mask.
10. Causal here means “the past can affect the future, and the future cannot affect the past”, the ordinary meaning of cause.
11. There is a second, different kind of mask. When you run many sentences of different lengths together, you pad the short ones with filler tokens.
12. A **padding mask** blocks attention to that filler, so real words never average in meaningless positions.
13. And there are models that use no causal mask at all. They read the whole text both ways at once. Those are excellent at understanding a fixed piece of text and cannot generate.

■ 48.5.2 PLAIN – a picture in your head

1. Imagine an exam where the questions and answers are printed on the same long scroll, in order.
2. Question 1, answer 1, question 2, answer 2, all the way down.
3. You are given the scroll rolled up, and a rule: you may unroll it only as far as the question you are currently answering, never further.
4. So when you are on question 3, you can see questions 1, 2 and 3 and answers 1 and 2, and nothing beyond.
5. That is exactly the causal mask. Position t sees positions 1 through t .
6. Now here is the part that surprises people. The examiner does not hand you one question at a time. The examiner hands the whole class the whole scroll at once, with each student allowed a different unrolling limit.
7. Student 1 may see 1 word. Student 2 may see 2 words. Student 500 may see 500.
8. All 500 students work at the same time, in one operation, and each is graded on a genuinely fair prediction.
9. That is why masked training is fast. One pass over a 4,000-token document produces 4,000 separate prediction problems, all solved in parallel.

Where this comparison breaks: the students are not separate. They are 500 rows of one matrix, and the “unrolling limit” is not enforced by discipline but by arithmetic that makes forbidden information contribute exactly zero. Also, a real exam has one right answer per question. Here the target is the actual next token in the training text, which is one sample from a distribution of acceptable continuations, and the model is graded against that one sample.

■ 48.5.3 PLAIN – a worked example

1. Take the scaled score matrix computed in section 48.4 for “the cat was hungry”, and apply a causal mask.
2. Before masking, the scaled scores were:

	the	cat	was	hungry
the	1.7321	1.1547	1.1547	1.7321
cat	4.6188	6.9282	4.6188	6.9282
was	4.0415	6.3509	4.6188	6.3509
hungry	3.4641	6.3509	3.4641	5.7735

3. Now set everything strictly above the diagonal to negative infinity:

	the	cat	was	hungry
the	1.7321	-inf	-inf	-inf
cat	4.6188	6.9282	-inf	-inf
was	4.0415	6.3509	4.6188	-inf
hungry	3.4641	6.3509	3.4641	5.7735

4. The pattern is a lower triangle of real numbers and an upper triangle of negative infinity. This is the causal mask, and it is always this shape.
5. Now softmax each row. Row “the” has exactly one surviving score, so it gets weight 1.0 and everything else 0.
6. Row “cat” has two surviving scores, 4.6188 and 6.9282. Their difference is 2.3094, so e to the 2.3094 = 10.07, and the weights are $1 / 11.07 = 0.0903$ and $10.07 / 11.07 = 0.9097$.
7. The full masked attention matrix:

A causal	the	cat	was	hungry
the	1.0000	0.0000	0.0000	0.0000

cat	0.0903	0.9097	0.0000	0.0000
was	0.0778	0.7836	0.1386	0.0000
hungry	0.0333	0.5978	0.0333	0.3356

8. Every row still sums to exactly 1. The zeros are true zeros, not small numbers.
9. Compare the “was” row with the unmasked version. Unmasked it was [0.0436, 0.4393, 0.0777, 0.4393]. Masked it is [0.0778, 0.7836, 0.1386, 0].
10. The 0.4393 that was going to “hungry” did not disappear. It was redistributed proportionally across the three legal positions. That is what the renormalization in softmax does for free.
11. Multiplying by V gives the masked output:

Output (causal)	dim0	dim1	dim2
the	3.0000	1.0000	3.0000
cat	0.2710	2.8193	1.1807
was	0.5107	2.7057	1.1556
hungry	0.5022	2.9000	1.4022

12. Note the first row. Position 1 can only see itself, so its output is exactly its own value vector [3, 1, 3]. That is always true of the first token in a causal model, in every layer.
13. Note the last row. It is identical to the unmasked last row, because the last position had nothing ahead of it to mask anyway.

■ 48.5.4 PLAIN - what is really happening inside

1. In code, the mask is not really negative infinity. It is a large negative number, or a genuine negative infinity that the softmax handles.
2. In float32, a common choice is $-1e9$ or the smallest representable value. In float16 it must be about -65504 or smaller in magnitude than that, because float16 cannot hold $-1e9$.
3. Using a finite value works because e to the power of $-1e9$ underflows to exactly 0.0 in floating point. The result is identical to true zero.
4. **The honest version:** there is a real failure mode here. If an entire row is masked, every score becomes negative infinity, and softmax computes 0 divided by 0, giving NaN, which then poisons the whole model. Implementations guard against this. It happens in practice with badly constructed padding masks.
5. The mask is not a parameter. Nothing about it is learned. It is a fixed boolean pattern computed from the sequence length.
6. It costs no memory to store, in good implementations, because the triangular pattern is generated on the fly inside the kernel rather than materialized as an $[n, n]$ tensor.
7. Now the padding mask. Suppose you batch three prompts of lengths 5, 9 and 3.
8. To make one rectangular tensor, you pad all three to length 9 with a filler token id.
9. Without a mask, position 2 of the third prompt would average in six filler positions, and its output would be contaminated.
10. The padding mask sets the columns of the padded positions to negative infinity for every row.
11. Padding masks and causal masks combine by taking whichever is more restrictive at each cell, which in boolean terms is a logical OR of the two blocked patterns.
12. A neat trick used by almost every training framework avoids padding entirely: concatenate many documents into one long stream, cut it into fixed-length chunks, and never pad at all. This is called sequence packing.

■ 48.5.5 TECHNICAL – the engineer’s version

1. The causal mask M is defined by $M[i][j] = 0$ if j is less than or equal to i , and negative infinity otherwise. It is added to the scaled scores before softmax.
2. The masked formula: $\text{softmax}(QK^T / \sqrt{d_k} + M) V$.
3. In PyTorch, `torch.triu(torch.ones(n, n), diagonal=1).bool()` produces the blocked pattern, and `scores.masked_fill(mask, float("-inf"))` applies it.
4. Since PyTorch 2.0, `scaled_dot_product_attention(..., is_causal=True)` is preferred, because fused kernels skip the masked blocks entirely rather than computing and discarding them, roughly halving the work.
5. That saving is real and worth naming: a causal attention over n tokens does about $n^2/2$ useful score computations, not n^2 .
6. Encoder models use no causal mask. BERT, from Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova at Google in October 2018, attends both directions at every layer.
7. That is why BERT cannot be trained on next-token prediction. If every position sees every position, the answer is already visible.
8. BERT is trained instead on **masked language modelling**: replace 15 per cent of tokens with a special [MASK] token and predict the originals. The masking there is of the input tokens, not of the attention matrix. Two different meanings of the same word, and this confuses people constantly.
9. The consequences of the choice:

Property	Causal (decoder)	Bidirectional (encoder)
Sees future tokens	No	Yes
Can generate text	Yes	No
Training signal per token	Every token	15 per cent of tokens
Best at	Generation, chat	Classification, embedding

10. The training-efficiency line in that table is important and often missed. A causal model gets a prediction target at every one of n positions. A masked language model gets one at roughly $0.15n$ positions, so it extracts less signal per pass over the data.
11. Prefix language modelling is the hybrid: bidirectional attention over a prompt prefix, causal attention over the generated part. Used in UniLM in 2019 and in some of the T5 variants.
12. Sliding window attention adds a second constraint: block anything further back than w positions. Mistral 7B, released October 2023, used $w = 4096$. That is covered in 48.13.
13. Observation tools: printing the mask is one line, but for real debugging the useful check is that every attention row sums to 1 and contains no NaN. A single NaN in one row propagates to the entire output within one layer.

■ 48.5.6 WORDS – remember these

1. Mask — blocking some positions from being looked at — an additive $[n, n]$ tensor of 0 and negative infinity applied before softmax.
2. Causal mask — you may not read ahead — the strictly upper-triangular block pattern, also called a look-ahead or autoregressive mask.
3. Padding mask — ignore the filler used to square up a batch — a per-sequence column mask blocking positions beyond the true length.
4. Autoregressive — predicting the next thing from all the previous things — a factorization of the joint probability as a product of conditionals.

5. Masked language modelling — hiding some input words and guessing them — the BERT objective, corrupting about 15 per cent of input tokens.
6. Sequence packing — gluing documents together to avoid padding — concatenating a corpus into a token stream and slicing fixed-length windows.
7. Bidirectional — able to see both left and right — an unmasked encoder attention pattern.

48.6 Multi-head attention

■ 48.6.1 PLAIN – in simple words

1. One attention operation produces one set of weights per position. One opinion about what matters.
2. But a word usually needs several different things at once.
3. Take the word “hungry” in “the cat was hungry”. It needs to know what it describes, which is “cat”. It also needs to know its grammatical role. It also needs to know where it sits in the sentence.
4. Those are different questions, and one set of weights cannot answer them all, because the weights must add up to 1 and get spent once.
5. So the model runs several attention operations side by side, each with its own W_Q , W_K and W_V . Each one is called a **head**.
6. Head 1 might mostly look at the previous word. Head 2 might look at the subject of the sentence. Head 3 might look at matching brackets.
7. They all run at the same time on the same input, and they do not talk to each other while running.
8. When they finish, their outputs are laid side by side in one long row, which is called concatenation.
9. Then one final learned matrix mixes that long row back down to the original width. That matrix is W_O , the output projection.
10. The clever part is that heads cost nothing extra. If the model is 512 wide and you want 8 heads, each head is 64 wide, and 8 times 64 is 512.
11. So the total work is the same as one 512-wide head. You have bought variety for free by splitting rather than adding.
12. That is the trade: eight narrow views instead of one wide view, at identical cost.

■ 48.6.2 PLAIN – a picture in your head

1. Picture a panel of eight specialist doctors examining the same patient at the same time.
2. The cardiologist looks at the heart. The neurologist looks at the nerves. The radiologist looks at the scans.
3. Each one asks different questions, of the same patient, and gets a different answer.
4. None of them sees the whole picture. Each is deliberately narrow.
5. When they are done, all eight write their findings on one long sheet.
6. A general physician reads the whole sheet and produces one combined assessment. That is W_O , the output projection.
7. The panel is faster than eight sequential consultations, because they work simultaneously.
8. And the panel is better than one generalist, because a generalist would have to spend the same total effort spread across all the specialities.

Where this comparison breaks: real specialists chose their speciality and know what it is. Heads are not assigned roles. They are eight identical copies with different random starting values, and whatever specialization appears is an accident of training that nobody designed. Many heads end up doing nothing useful at all, which no hospital would tolerate. And a doctor examines the actual

patient. A head sees only a 64-dimensional slice of a vector that has already passed through many previous layers.

■ 48.6.3 PLAIN – a worked example

1. Work out every shape for the 2017 base model, with a 10-token sentence.
2. The configuration: $d_{\text{model}} = 512$, 8 heads, $d_k = d_v = 64$, since $512 / 8 = 64$.

input X	[1, 10, 512]
project with W_Q [512, 512]	[1, 10, 512]
reshape into heads	[1, 10, 8, 64]
transpose head axis forward	[1, 8, 10, 64]
same for K and V	[1, 8, 10, 64]
scores per head $Q K^T$	[1, 8, 10, 10]
softmax over last axis	[1, 8, 10, 10]
weights times V	[1, 8, 10, 64]
transpose back	[1, 10, 8, 64]
concatenate heads	[1, 10, 512]
project with W_O [512, 512]	[1, 10, 512]

3. Look at the first and last lines. The input was [1, 10, 512] and the output is [1, 10, 512]. The shape is unchanged, which is why blocks can be stacked.
4. Look at the score tensor: [1, 8, 10, 10]. That is 8 separate 10-by-10 grids, one per head. 800 numbers.
5. There is no reshaping magic here. “Reshape into heads” just means reading the same 512 numbers as 8 groups of 64, without moving any data.
6. Head 0 gets dimensions 0 to 63. Head 1 gets 64 to 127. And so on to head 7, which gets 448 to 511.
7. Parameters: W_Q , W_K , W_V and W_O are each 512 by 512, so 262,144 each, and 1,048,576 for the whole multi-head attention block.
8. That is exactly 4 times d_{model} squared, which is a formula worth memorizing: attention costs $4 d^2$ parameters per layer.
9. Now do the same for a modern model, Llama 3.1 8B: $d_{\text{model}} = 4096$, 32 query heads, 8 key-value heads, head dimension 128.
10. Note that 32 times 128 = 4096, so the query side still adds up to d_{model} .
11. But the key and value side uses only 8 heads, so 8 times 128 = 1024.

q_proj	[4096, 4096]	=	16,777,216	parameters
k_proj	[1024, 4096]	=	4,194,304	parameters
v_proj	[1024, 4096]	=	4,194,304	parameters
o_proj	[4096, 4096]	=	16,777,216	parameters

total attention		=	41,943,040	per layer

12. Each of the 8 key-value heads is shared by 4 query heads. That is grouped-query attention, and its purpose is to shrink the memory cost of generation, explained fully in 48.13.
13. Across 32 layers that is 1,342,177,280 attention parameters, which is 16.7 per cent of the model’s 8,030,261,248 total.

■ 48.6.4 PLAIN – what is really happening inside

1. Heads are not separate networks. They are one matrix multiplication whose output is interpreted as several groups.
2. W_Q is a single $[512, 512]$ matrix. Nothing in it says “this column belongs to head 3”. The head boundaries exist only in how the result is reshaped.
3. That is why the cost is identical to single-head attention with the full width. The reshape is free.
4. The heads become genuinely independent at the score step, because a query from head 3 is only ever dotted with keys from head 3.
5. This makes the attention computation **block diagonal** across heads. Eight small independent problems rather than one large one.
6. The only place heads mix is W_O at the end. Every output dimension of W_O is a weighted combination of all 512 concatenated numbers, so every head can influence every output.
7. That means W_O is not just a resize. It is the thing that decides how much each head’s finding is worth, and it is learned.
8. A useful way to see it: W_O can be split into 8 blocks of $[64, 512]$, one per head. Then the block’s output is added to the residual stream. Each head writes its own contribution independently, and they sum.
9. This decomposition is exactly how the interpretability literature analyses attention, and it is mathematically exact, not an approximation.
10. Heads within one layer cannot see each other’s results. Head 3 cannot use what head 5 found. That only becomes possible in the next layer.

■ 48.6.5 TECHNICAL – the engineer’s version

1. Multi-head attention, from the 2017 paper: $\text{concat}(\text{head}_1, \dots, \text{head}_h) W_O$, where $\text{head}_i = \text{Attention}(Q W_{Q_i}, K W_{K_i}, V W_{V_i})$.
2. Standard configurations:

Model	d_model	Heads	Head dim
Transformer base 2017	512	8	64
BERT-base 2018	768	12	64
GPT-2 small 2019	768	12	64
Llama 3.1 8B 2024	4096	32	128

3. The convention $d_k = d_{\text{model}} / h$ keeps total cost constant. It is a convention, not a requirement, and models that break it exist.
4. The 2017 paper ran an ablation: with d_{model} fixed at 512, 1 head scored 0.9 BLEU worse than 8 heads, and 32 heads was also slightly worse than 8. So more heads is not monotonically better once each head becomes too narrow.
5. What heads have been observed to learn, from published analysis:
 1. Positional heads that attend almost entirely to the previous token or the next token. These are the most reliable finding and appear in nearly every model.
 2. Syntactic heads that attend from a verb to its direct object, or from a determiner to its noun. Kevin Clark, Urvashi Khandelwal, Omer Levy and Christopher Manning documented these in BERT in 2019.
 3. Coreference heads that attend from a pronoun to its antecedent, also in the same 2019 analysis, at above-chance but well below perfect accuracy.

4. Induction heads, which complete a repeated pattern by finding an earlier occurrence of the current token and copying what followed it. Described by Catherine Olsson and colleagues at Anthropic in 2022 and connected to in-context learning.
6. **The honest version:** this is not settled science. Head interpretability is active research, not established fact. The named roles above are real published observations on specific models, and they do not mean that heads in general have clean human-nameable jobs.
7. Redundancy is well established and cuts the other way. Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich and Ivan Titov showed in 2019 that pruning 38 of 48 encoder heads in a translation model cost only 0.15 BLEU on English-Russian WMT data.
8. Paul Michel, Omer Levy and Graham Neubig published the same finding independently in 2019, in a paper titled “Are Sixteen Heads Really Better Than One?”. Many heads can be deleted at test time with little loss.
9. So the correct summary is: some heads do identifiable jobs, most heads are partly redundant, and the population as a whole is not a tidy division of labour.
10. Marketing claim to reject: any product page saying the model “has heads dedicated to reasoning” or similar. No such assignment is designed or verified.
11. Reproducing head analysis is straightforward. Load a model with `output_attentions=True`, take the `[B, h, n, n]` tensor, and for a previous-token head you will see nearly all mass on the first subdiagonal.

■ 48.6.6 WORDS – remember these

1. Head — one independent attention view — one of h parallel attention computations, each of width d_{model} / h by convention.
2. Multi-head attention — several attention views run side by side — h parallel scaled dot-product attentions, concatenated and projected by W_O .
3. Concatenation — laying results side by side into one long row — joining the h per-head outputs along the feature axis to restore width d_{model} .
4. Output projection — the matrix that mixes the heads back together — W_O , of shape $[d_{\text{model}}, d_{\text{model}}]$, applied after concatenation.
5. Head dimension — how wide one head is — d_k , typically 64 in 2017-era models and 128 in current ones.
6. Induction head — a head that continues a repeated pattern — a two-head circuit that matches the current token to a previous occurrence and copies the following token.
7. Head pruning — deleting heads that do not earn their keep — zeroing whole heads at inference, shown to cost little accuracy in several 2019 studies.

48.7 Position: attention has no idea what order things are in

■ 48.7.1 PLAIN – in simple words

1. Here is a fact about attention that catches everybody by surprise.
2. Attention has no concept of order. None. It does not know which word came first.
3. To attention, a sentence is a bag of items, not a sequence.
4. Prove it to yourself. Look back at the formula: $\text{softmax}(Q K^T / \sqrt{d_k}) V$.
5. Find the position number in it. There is none. There is no i , no t , no index of any kind.
6. Every operation is a dot product between two token vectors, and a dot product does not care where either token sits.
7. So “dog bites man” and “man bites dog” would produce exactly the same set of output vectors, just shuffled the same way as the input.
8. That would be a catastrophe for language, where order is most of the meaning.
9. So order has to be added by hand. The model is given, for each position, some extra numbers that say where that position is.
10. The original method was to add a fixed pattern of sine and cosine waves to each token’s vector before the first layer.
11. A second method is to learn a separate vector for position 1, position 2, and so on, exactly like an embedding table for positions.
12. The modern method, used by almost every current model, is different and cleverer. It rotates the query and key vectors by an angle proportional to their position.
13. It is called rotary position embedding, or RoPE, and the reason it won is that it makes the score between two tokens depend only on how far apart they are, not on where they are in absolute terms.

■ 48.7.2 PLAIN – a picture in your head

1. Imagine a set of magnetic word tiles scattered on a table, one word per tile.
2. Attention reads all the tiles at once and works out which ones relate to which others.
3. Now shake the table. The tiles land in a different arrangement. Attention gives exactly the same answer, because it never looked at the arrangement.
4. To fix this, you write a number on the back of each tile: 1, 2, 3, 4.
5. That is learned absolute position embedding. Simple, and it fails as soon as you meet tile 5,000 when you only ever wrote numbers up to 4,096.
6. Now a better fix. Instead of writing numbers, you tilt each tile by an angle. Tile 1 is tilted 10 degrees, tile 2 is tilted 20, tile 3 is tilted 30.
7. When two tiles are compared, what matters is the difference in their tilts. Tiles 1 and 2 differ by 10 degrees. Tiles 500 and 501 also differ by 10.
8. So the comparison naturally depends on the gap between positions, not the absolute positions.
9. That is rotary position embedding. Tilt by angle, compare by difference.

Where this comparison breaks: a tile has one tilt angle. A real RoPE vector is split into 64 pairs of numbers, and each pair is rotated at its own speed, from one full turn every 6.3 positions down to one full turn every two and a half million positions. So it is 64 clocks running at 64 different speeds, not one tilt. And the tilt is applied to the query and key vectors only, never to the values.

■ 48.7.3 PLAIN – a worked example

1. First, the proof that attention ignores order. Take the four-token example from 48.4, which produced this output:

Output	dim0	dim1	dim2
the	1.6405	2.1798	1.9607
cat	0.6807	2.8645	1.5452
was	0.7257	2.8350	1.5266
hungry	0.5022	2.9000	1.4022

- Now shuffle the input rows into the order hungry, cat, the, was, and run the identical computation with the identical matrices:

Output of shuffled input	dim0	dim1	dim2
hungry	0.5022	2.9000	1.4022
cat	0.6807	2.8645	1.5452
the	1.6405	2.1798	1.9607
was	0.7257	2.8350	1.5266

- Every row is identical to before. Only their order changed, in exactly the same way the input changed.
- That property has a name: **permutation equivariance**. Shuffle the input, and the output is shuffled the same way, with no other change.
- This was verified numerically, not asserted. The two matrices agree to floating-point precision.
- Now sinusoidal position encoding. The formula from the 2017 paper, for position pos and dimension index i:

$$\begin{aligned} \text{PE}(\text{pos}, 2i) &= \sin(\text{pos} / 10000^{(2i/d_{\text{model}})}) \\ \text{PE}(\text{pos}, 2i+1) &= \cos(\text{pos} / 10000^{(2i/d_{\text{model}})}) \end{aligned}$$

- With $d_{\text{model}} = 8$ there are 4 sine-cosine pairs. Their frequencies:

Pair	Frequency	Wavelength
0	1.0	6.28 positions
1	0.1	62.8 positions
2	0.01	628 positions
3	0.001	6283 positions

- The actual values for the first five positions:

pos 0:	0.0000	1.0000	0.0000	1.0000	0.0000	1.0000	0.0000	1.0000
pos 1:	0.8415	0.5403	0.0998	0.9950	0.0100	1.0000	0.0010	1.0000
pos 2:	0.9093	-0.4161	0.1987	0.9801	0.0200	0.9998	0.0020	1.0000
pos 3:	0.1411	-0.9900	0.2955	0.9553	0.0300	0.9996	0.0030	1.0000
pos 4:	-0.7568	-0.6536	0.3894	0.9211	0.0400	0.9992	0.0040	1.0000

- Read the columns. The first pair changes fast and wraps around quickly. The last pair barely moves at all across five positions.
- That is the design. Fast waves distinguish neighbours precisely, slow waves distinguish far-apart positions coarsely. It is a positional binary counter made of continuous waves.
- Now RoPE, with real numbers. Take a query pair $q = [1.0, 0.0]$ and a key pair $k = [0.6, 0.8]$. Their plain dot product is 0.6.
- RoPE rotates q by m times theta and k by n times theta, where m and n are the two positions. With $\text{theta} = 1.0$ radian:

m	n	m - n	Dot after rotation
1	0	1	0.997358
5	4	1	0.997358
100	99	1	0.997358
3	0	3	-0.481099
7	4	3	-0.481099

13. Look at the last column. Every pair with the same gap gives the same result, to six decimal places, whether the positions are 1 and 0 or 100 and 99.
14. That is the whole selling point of RoPE, demonstrated numerically. The score depends on relative distance only.

■ 48.7.4 PLAIN – what is really happening inside

1. Sinusoidal encoding is added to the token embedding, once, before layer 1. Token vector plus position vector, element by element.
2. That means position information and word information are mixed together in the same numbers from the start, and every layer sees the sum.
3. Learned absolute embeddings work identically, except the position vectors are parameters found by training rather than computed from a formula.
4. GPT-2 uses learned absolute embeddings: a table of 1,024 rows by 768 columns, which is 786,432 parameters spent purely on saying where things are.
5. Both of these have the same weakness. They are absolute. Position 3,000 has its own vector, unrelated to position 2,999.
6. And if the table has 1,024 rows, position 1,024 does not exist. The model cannot be run on longer text at all.
7. Sinusoids can at least be computed for any position, but a model trained only on positions 0 to 511 has never seen what position 800 looks like, and in practice quality collapses beyond the trained length.
8. RoPE takes a completely different route. It never adds anything. It rotates.
9. Split the query vector into consecutive pairs: dimensions (0,1), (2,3), (4,5), and so on. Treat each pair as a point on a flat plane.
10. Rotate pair j by an angle of position times θ_j , where θ_j gets smaller for later pairs, exactly like the sinusoid frequencies.
11. Do the same to the key vector, using the key's own position.
12. Now take the dot product. Rotating both vectors by the same amount leaves the dot product unchanged, so only the difference in rotation survives.
13. That is basic geometry: a dot product is invariant under rotation. Rotating q by m and k by n is the same as rotating q by (m - n) and leaving k alone.
14. So the score naturally becomes a function of m - n. No extra terms, no lookup table, no learned parameters at all.
15. RoPE is applied inside every attention layer, to Q and K only, every time. It is not applied to V, and it is not added to the residual stream.

■ 48.7.5 TECHNICAL – the engineer's version

1. Permutation equivariance is exact: for any permutation matrix P, $\text{Attention}(PX) = P \text{Attention}(X)$. This is why position injection is mandatory rather than helpful.

2. **Sinusoidal**, 2017. Fixed, no parameters, defined for all positions. The base 10,000 sets the slowest wavelength. The paper's stated reason for choosing sinusoids is that $PE(pos + k)$ is a linear function of $PE(pos)$ for any fixed offset k , so relative offsets are in principle learnable as a linear map. This was verified numerically for this chapter: for the fastest pair with $\theta = 1$, rotating $PE(5)$ by 3 radians reproduces $PE(8)$ exactly.
3. The paper also reports that learned and sinusoidal encodings performed almost identically, and chose sinusoidal for its extrapolation potential.
4. Sinusoid dot products fall with distance, which is a useful inductive bias. Measured at $d_{model} = 64$: the dot product of $PE(0)$ with itself is 32.0, with $PE(1)$ is 30.92, with $PE(10)$ is 21.05, and with $PE(200)$ is 11.44.
5. **Learned absolute**, used by BERT (2018), GPT-1, GPT-2 and GPT-3. A table of shape $[max\_positions, d_{model}]$. Hard cap on length, no extrapolation at all.
6. **Relative position representations**, from Peter Shaw, Jakob Uszkoreit and Ashish Vaswani in 2018, add a learned vector for each relative offset into the key and value computation.
7. **T5 relative position bias**, from Colin Raffel and colleagues in 2019, adds a learned scalar to the attention logit per bucket of relative distance, shared across layers, with logarithmically spaced buckets. Cheap and effective.
8. **ALiBi**, from Ofir Press, Noah Smith and Mike Lewis in 2021, adds a linear penalty to the attention logit: minus m times the distance, where m is a fixed per-head slope, typically a geometric sequence such as $1/2, 1/4, 1/8$. No parameters at all, and it extrapolates well beyond the training length. The paper's title is "Train Short, Test Long".
9. **RoPE**, from Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen and Yunfeng Liu, in the RoFormer paper first posted in April 2021. It is the dominant choice as of 2026. Llama, Mistral, Qwen, Gemma, DeepSeek and Phi all use it.
10. RoPE frequencies use $\theta_j = base^{(-2j/d_{head})}$. With base 10,000 and head dimension 128, the fastest pair has wavelength 6.3 positions and the slowest has wavelength 54,410 positions.
11. Raising the base stretches everything. With base 500,000, as used in Llama 3.1, the slowest pair has wavelength 2,559,196 positions. That is how a 128k-token context is supported without the slowest frequency wrapping.

Method	Year	Parameters	Extrapolates
Sinusoidal	2017	0	Poorly
Learned absolute	2018	$max\_pos \times d$	Not at all
T5 bias	2019	~ 32 per head	Moderately
ALiBi	2021	0	Well
RoPE	2021	0	With extension

12. Context extension. A model trained at 4,096 tokens fails at 32,768 because the rotation angles at position 30,000 are outside anything it saw.
13. **Position interpolation**, from Shouyuan Chen, Sherman Wong, Liangjian Chen and Yuandong Tian at Meta in June 2023, divides every position index by a scale factor s . To go from 4,096 to 32,768, use $s = 8$, so position 32,767 is presented to the model as 4,095.9.
14. That keeps every angle inside the trained range, at the cost of squashing fine positional detail. It requires a short fine-tune, typically 1,000 steps, and works remarkably well.
15. **NTK-aware scaling**, which appeared as a community proposal in mid-2023, changes the RoPE base instead of the positions, stretching low frequencies more than high ones so nearby tokens keep their resolution.

16. **YaRN**, from Bowen Peng, Jeffrey Quesnelle, Honglu Fan and Enrico Shippole in 2023, combines the two: interpolate the low-frequency dimensions, leave the high-frequency ones alone, apply a ramp in between, and add a temperature adjustment to the attention logits. It reaches long contexts with about ten times less fine-tuning data than plain interpolation.
17. Llama 3.1, released July 2024, used a variant of this family to reach a 131,072-token context from a shorter pretraining length, together with base 500,000.
18. Established fact: RoPE gives exact relative-distance dependence, and interpolation methods extend context in practice. Active research: how much real long-range capability extension gives, as opposed to merely not crashing. Benchmarks such as RULER, from NVIDIA in 2024, found many models claiming 32,000-token contexts performed well at much shorter lengths.

■ 48.7.6 WORDS – remember these

1. Permutation equivariance — shuffle the input, get the same output shuffled — $\text{Attention}(PX) = P \text{ Attention}(X)$ for any permutation matrix P .
2. Positional encoding — extra numbers telling the model where each token sits — any scheme injecting order information into an order-blind operation.
3. Sinusoidal encoding — fixed sine and cosine waves at many speeds — the 2017 scheme with frequencies $\text{base}^{(-2i/d)}$, base 10,000.
4. Absolute position — where a token is in the whole sequence — an index from 0 to $n-1$, encoded directly.
5. Relative position — how far apart two tokens are — the offset $m - n$, which is what a score should depend on.
6. RoPE — rotating queries and keys by an angle set by position — rotary position embedding, applying a block-diagonal rotation to Q and K .
7. RoPE base (θ) — the number controlling how slow the slowest wave is — 10,000 originally, raised to 500,000 or higher for long contexts.
8. Position interpolation — squeezing long positions into the trained range — dividing position indices by a scale factor s , then fine-tuning.
9. YaRN — a smarter context extension that treats fast and slow waves differently — wavelength-dependent interpolation with a logit temperature correction.
10. ALiBi — penalizing distant tokens with a straight-line penalty — adding minus m times $|i - j|$ to the attention logit, with fixed per-head slopes.

48.8 The rest of the block: feed-forward, residuals and normalization

■ 48.8.1 PLAIN – in simple words

1. Attention is only half of a transformer block. The other half is a small ordinary network applied to each position separately.
2. It is called the **feed-forward network**, or FFN, or the MLP. All three names mean the same thing.
3. It has exactly two steps: widen, then narrow. Multiply by a matrix that makes the vector about four times wider, apply a non-linear function, multiply by a matrix that brings it back to the original width.
4. The important word is “separately”. Every position goes through the same FFN, on its own, with no reference to any other position.
5. So attention is where positions talk to each other, and the FFN is where each position thinks about what it just heard.
6. Now the surprising part: this simple two-matrix block holds most of the model’s parameters. About 70 per cent for an 8-billion model, and about 80 per cent for larger ones.
7. Chapter 47 counted this exactly. For Llama 3.1 8B, the feed-forward matrices hold 176,160,768 parameters per layer against 41,943,040 for attention.
8. So attention gets all the attention, and the FFN holds the weights.
9. Two more pieces complete the block, and both are about making deep stacks trainable rather than about capability.
10. A **residual connection** means: whatever a sub-layer computes, add the original input back on. Output equals input plus change, not just change.
11. Without residuals, a 32-layer stack cannot be trained. The correction signal dies on the way back, exactly as it did in recurrent networks.
12. **Layer normalization** rescales each vector so its numbers have a predictable size, which stops values drifting to extremes as they pass through dozens of layers.
13. That is the whole block: attention, add the input back, normalize, feed-forward, add the input back, normalize. Then repeat.

■ 48.8.2 PLAIN – a picture in your head

1. Picture a long assembly line with 32 stations, each one modifying a product.
2. In a bad design, each station takes the previous station’s output and rebuilds it from scratch. If station 7 does something wrong, everything downstream is corrupted, and the factory manager cannot tell which station caused it.
3. In the residual design, each station is given the product and told: do not rebuild it. Just add your modification on top and pass it along.
4. Now the product carries a continuous history, and every station’s change is a small addition rather than a replacement.
5. If the manager finds a fault at the end, the complaint can travel back down the line unchanged, because the additions form a straight unbroken path.
6. That straight path is exactly what a residual connection creates for gradients. It is often called the residual stream, and it runs the whole depth of the model.
7. Normalization is the quality inspector between stations, resetting the product to a standard size and shape so the next station receives something in the range it expects.

Where this comparison breaks: nothing is physically added to a product, and the “modification” is a vector added element by element, which can cancel earlier contributions rather than accumulate. Also, real assembly stations have fixed jobs. These are all trained together, so a station’s job is defined

only by what the others ended up doing.

■ 48.8.3 PLAIN – a worked example

1. First, why residual connections matter, with arithmetic.
2. During backpropagation, the correction signal is multiplied by something at each layer. Call it the per-layer factor.
3. Without residuals, that factor depends entirely on the layer's weights and there is no reason for it to be near 1.

Per-layer factor	After 32 layers	After 96 layers
0.90	0.034	0.00004
0.95	0.194	0.0073
1.00	1.000	1.000
1.05	4.765	108.2

4. At 0.9 per layer, a 96-layer model receives 0.004 per cent of the signal at the bottom. Training the early layers becomes impossible.
5. With a residual connection, the layer computes $x + f(x)$, so the derivative is $1 + f'(x)$. The 1 is there whatever f does.
6. So there is always a path with factor exactly 1, and the product along that path stays exactly 1 no matter how deep the stack is.
7. That is the entire trick, and it came from image models: Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun published ResNet in December 2015, trained 152 layers, and won the ImageNet competition that year.
8. Now layer normalization, worked on a real vector of 8 numbers:

```
x      = 2.0  -1.0  0.5  4.0  -2.5  1.0  0.0  3.0
mean   = 0.875
variance = 3.921875
std dev = 1.980372
```

9. Subtract the mean and divide by the standard deviation:

```
normed = 0.5681 -0.9468 -0.1894 1.5780
        -1.7042 0.0631 -0.4418 1.0730
```

10. Check: the mean of the normalized vector is 0.0 and its standard deviation is 1.0. That is guaranteed by construction.
11. Then multiply by a learned scale and add a learned shift, one per dimension, so the layer can undo the normalization if that helps.
12. Crucially, the mean and variance were computed across the 8 features of this one token. Not across the batch. Not across the sequence.
13. That is why it works with any batch size, including batch size 1, which is the normal case when you chat with a model.
14. RMSNorm is the cheaper modern variant. It skips the mean subtraction and divides by the root mean square:

```
root mean square of x = 2.165064
normed = 0.9238 -0.4619 0.2309 1.8475
        -1.1547 0.4619 0.0000 1.3856
```

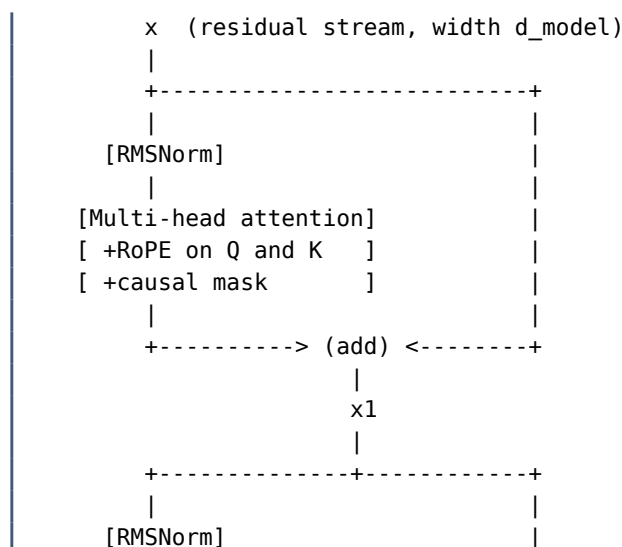
15. It has one learned parameter per dimension instead of two, and in practice performs the same. Llama, Mistral, Qwen and Gemma all use it.

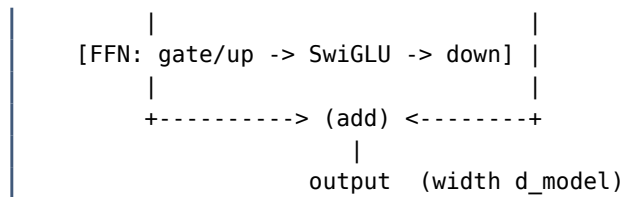
■ 48.8.4 PLAIN - what is really happening inside

1. The feed-forward network, written out: $\text{FFN}(x) = W_2 (\text{activation}(W_1 x + b_1))$
 ■ b_2 .
2. W_1 has shape $[d_{\text{model}}, d_{\text{ff}}]$ and W_2 has shape $[d_{\text{ff}}, d_{\text{model}}]$, with d_{ff} usually 4 times d_{model} .
3. The 4x ratio is a convention from the 2017 paper, where d_{model} was 512 and d_{ff} was 2048. It is not a law and modern models vary it.
4. The activation function was ReLU in 2017, then GELU in BERT and GPT-2, and now SwiGLU in most current models. Chapter 46 covered what these do.
5. SwiGLU uses three matrices rather than two: a gate, an up-projection and a down-projection, with the gate multiplied elementwise into the up path.
6. Because it has three matrices, models using SwiGLU shrink d_{ff} to about 8/3 of d_{model} to keep the parameter count similar. Llama 3.1 8B uses 14,336 with d_{model} 4096, which is 3.5x rather than the naive 4x.
7. Now pre-norm versus post-norm, which is a real and consequential difference.
8. **Post-norm**, the 2017 original: $x \rightarrow \text{attention} \rightarrow \text{add } x \rightarrow \text{normalize}$. The normalization sits after the addition, on the main path.
9. **Pre-norm**, used by almost everything since about 2019: $x \rightarrow \text{normalize} \rightarrow \text{attention} \rightarrow \text{add } x$. The normalization sits inside the branch, and the main path from input to output has nothing on it but additions.
10. That difference is everything. In pre-norm, the residual stream is a clean sum with no normalization interrupting it, so the gradient path really is unobstructed all the way down.
11. In post-norm, every residual addition is immediately rescaled, which shrinks the accumulated signal and makes deep stacks unstable.
12. The practical consequence: post-norm transformers need a learning-rate warmup or they diverge. Pre-norm ones train stably without it.
13. The 2017 paper used post-norm with 4,000 warmup steps, and that warmup was not optional.

■ 48.8.5 TECHNICAL - the engineer's version

1. The complete pre-norm decoder block, drawn:





2. Parameter accounting per layer, $d = d_{\text{model}}$:

Component	Parameters	GPT-2 small
Attention (4 matrices)	$4 d^2$	2,359,296
FFN (2 matrices, $4 \times$)	$8 d^2$	4,718,592
Two LayerNorms	$4 d$	3,072
Block total	$12 d^2 + 4d$	7,080,960

- So the FFN is exactly two thirds of a classic block's parameters, from the ratio $8 d^2$ to $12 d^2$.
- For Llama 3.1 8B with SwiGLU: attention is 41,943,040 and the FFN is $3 \times 4096 \times 14336 = 176,160,768$, so the FFN is 80.8 per cent of the layer.
- Across the whole model, including the two 525,336,576-parameter vocabulary tables, the FFN share is 70.2 per cent and attention is 16.7 per cent. These figures match Chapter 47 exactly.
- Layer normalization is from Jimmy Lei Ba, Jamie Ryan Kiros and Geoffrey Hinton, July 2016. It normalizes over the feature axis of a single example, unlike batch normalization from Sergey Ioffe and Christian Szegedy in 2015, which normalizes over the batch axis.
- That distinction is why layer norm won in language models: batch statistics are useless when sequences have different lengths and batch size can be 1.
- RMSNorm is from Biao Zhang and Rico Sennrich, 2019. It drops the mean subtraction and the shift term, saving one reduction and d parameters per norm, for roughly the same quality.
- Epsilon inside the square root prevents division by zero. Typical values are $1e-5$ for LayerNorm and $1e-5$ or $1e-6$ for RMSNorm. This is an implementation detail that occasionally matters for numerical reproducibility.
- Pre-norm was analysed formally by Ruibin Xiong and colleagues in the 2020 paper "On Layer Normalization in the Transformer Architecture", which showed that post-norm gradients at initialization scale with the square root of depth while pre-norm gradients do not, explaining the warmup requirement.
- Baevski and Auli in 2018 and Nguyen and Salazar in 2019 had already reported the practical benefit. GPT-2, in February 2019, used pre-norm.
- A third variant exists: sandwich norm, or normalization both before and after the sub-layer, used in some very large training runs including Gemma 2 in 2024, to control activation growth.
- The residual stream view: because every block writes $x + f(x)$, the model's activations at layer L are the embedding plus the sum of every sub-layer's output so far. This decomposition is exact and is the foundation of mechanistic interpretability work, notably "A Mathematical Framework for Transformer Circuits" from Anthropic in December 2021.
- Observation: activation norms typically grow with depth in trained transformers, often by an order of magnitude from layer 1 to the last layer. This is measurable with a forward hook and is a routine sanity check when training goes wrong.

■ 48.8.6 WORDS – remember these

1. Feed-forward network — the widen-then-narrow part applied to each position — FFN or MLP, two or three linear layers with a non-linearity, d_{ff} typically about $4 \times d_{model}$.
2. Position-wise — applied to each token independently — the same weights applied to every position with no mixing across the sequence axis.
3. Residual connection — adding the input back onto the output — the identity shortcut $x + f(x)$, from ResNet in 2015.
4. Residual stream — the main pipe running the depth of the model — the d_{model} -wide activation that every block reads from and adds to.
5. Layer normalization — rescaling one token's numbers to a standard size — normalizing over the feature axis per example, with learned scale and shift.
6. RMSNorm — cheaper layer normalization — dividing by the root mean square with no mean subtraction and no shift term.
7. Pre-norm — normalize before the sub-layer, add after — the arrangement that keeps the residual path free of normalization, standard since about 2019.
8. Post-norm — normalize after adding — the 2017 arrangement, which needs learning-rate warmup to train stably.
9. SwiGLU — a gated activation using three matrices — swish-gated linear unit, standard in the Llama, Mistral and Qwen families.

48.9 The whole model, assembled

■ 48.9.1 PLAIN – in simple words

1. Everything so far has been parts. Here is the whole machine, in order.
2. Step 1: your text is chopped into tokens and each token becomes an integer id. Chapter 47 covered this.
3. Step 2: each id is used to look up a row in the embedding table. Now every token is a vector.
4. Step 3: position information is injected. In modern models this happens inside attention, as a rotation, rather than as an addition here.
5. Step 4: the stack. N identical blocks, one after another. Each block is attention then feed-forward, both with residual connections and normalization.
6. Identical means identical in shape, not in values. Every block has its own separate parameters.
7. Step 5: one final normalization at the top of the stack.
8. Step 6: multiply by one last matrix that turns each d_{model} -wide vector into one number per token in the vocabulary. Those numbers are called **logits**.
9. Step 7: softmax over the vocabulary turns the logits into probabilities that add up to 1.
10. The output is one probability distribution per input position. For a 10-token prompt, you get 10 distributions.
11. During generation you only use the last one, because that is the prediction for what comes next.
12. During training you use all of them, because each position is a separate prediction problem, which is why masked parallel training is so efficient.

■ 48.9.2 PLAIN – a picture in your head

1. Picture a tall narrow building with 32 identical floors.
2. On the ground floor, a crowd of people walks in. One person per token. Each is handed a card with numbers on it.
3. On every floor, two things happen. First everybody talks to everybody and updates their card based on what they heard. Then each person sits alone and thinks, updating their card again.
4. Nobody joins and nobody leaves. The same number of people walk out of floor 32 as walked into floor 1.
5. The cards get richer on every floor. A person who arrived holding only the word “it” leaves floor 32 holding something that encodes what “it” refers to, what grammatical role it plays, and what is likely to come next.
6. At the top, only the last person in the queue is asked a question: given everything you now know, what word comes next?
7. That person’s card is compared against a list of 128,256 possible words, and a score is produced for each.

Where this comparison breaks: the people do not know anything and are not thinking. Each is a vector being multiplied by matrices. And the last person’s answer is not derived from a conversation, but from a single dot product of their card against every row of a very large table. Also, at training time every person is asked a question, not just the last one.

■ 48.9.3 PLAIN – a worked example

1. Trace one prompt through Llama 3.1 8B, showing the tensor shape at every stage. Configuration: vocabulary 128,256, d_model 4,096, 32 layers, 32 query heads, 8 key-value heads, head dimension 128, d_ff 14,336.
2. The prompt is six tokens long, batch size 1.

stage	shape	numbers
-----	-----	-----
text	"the cat was hungry"	-
token ids	[1, 6]	6
embedding lookup	[1, 6, 4096]	24,576
block 1 input	[1, 6, 4096]	24,576
norm	[1, 6, 4096]	24,576
q after projection	[1, 32, 6, 128]	24,576
k after projection	[1, 8, 6, 128]	6,144
v after projection	[1, 8, 6, 128]	6,144
k,v repeated to 32 heads	[1, 32, 6, 128]	24,576
scores	[1, 32, 6, 6]	1,152
weights after softmax	[1, 32, 6, 6]	1,152
attention output	[1, 32, 6, 128]	24,576
after o_proj	[1, 6, 4096]	24,576
after residual add	[1, 6, 4096]	24,576
norm	[1, 6, 4096]	24,576
ffn hidden (gate and up)	[1, 6, 14336]	86,016
after down_proj	[1, 6, 4096]	24,576
after residual add	[1, 6, 4096]	24,576
block 2 ... block 32	[1, 6, 4096]	24,576
final RMSNorm	[1, 6, 4096]	24,576
lm_head projection	[1, 6, 128256]	769,536
softmax over vocabulary	[1, 6, 128256]	769,536
take last position	[1, 128256]	128,256

3. Read down the shape column. From the embedding to the final norm, the shape never changes: $[1, 6, 4096]$. That is the residual stream, and it is constant width for the entire depth.
4. The only places the shape changes are inside a block, temporarily, and at the very end.
5. The last projection is the expensive one in terms of output size. It turns 24,576 numbers into 769,536 numbers, because the vocabulary is large.
6. That final matrix, `lm_head.weight`, is $[128256, 4096]$, holding 525,336,576 parameters. It is 6.5 per cent of the model on its own.
7. Some models tie this matrix to the input embedding table, using the same numbers for both. GPT-2 does this. Llama 3.1 8B does not.
8. Tying halves the vocabulary cost. Untying gives slightly better quality at larger scale. This is a real trade-off and both choices are in current use.

■ 48.9.4 PLAIN – what is really happening inside

1. A **logit** is a raw score before it becomes a probability. That is its only definition. It can be any real number, positive or negative.
2. The name comes from statistics, where the logit function is the logarithm of the odds. In this context it just means “the thing softmax eats”.
3. There is exactly one logit per vocabulary entry. For Llama 3.1, 128,256 of them, every single time a token is produced.
4. Each logit is one dot product: the final hidden vector of length 4,096 dotted with one row of the output matrix, also length 4,096.
5. So the model is asking, for every possible next token: how well does my current state line up with the direction that means this token?
6. Softmax then converts. Exponentiate every logit, add them all up, divide.
7. A logit of 4.0 against a logit of 3.2 becomes probabilities 0.57 and 0.26, because what matters is the difference, not the absolute values.
8. Adding 100 to every logit changes nothing at all. Softmax is invariant to a constant shift. That is why logits are not comparable across models or even across steps.
9. During training, the loss is the negative log of the probability the model gave to the token that actually came next. That is cross-entropy loss, from Chapter 46.
10. If the model gave the right token probability 0.5, the loss is 0.693. If it gave 0.01, the loss is 4.605. The worse the guess, the larger the penalty, growing without bound.
11. Nothing else is optimized. There is no separate objective for grammar, for facts, or for helpfulness during pretraining. One number, minimized.

■ 48.9.5 TECHNICAL – the engineer’s version

1. Full forward pass, in the order the code runs:

```

h = embed_tokens(ids)                # [B, n, d]
for layer in layers:                 # N identical blocks
    r = h
    h = rmsnorm(h, layer.attn_norm)
    h = attention(h, rope, causal_mask) # +RoPE inside
    h = r + h
    r = h
    h = rmsnorm(h, layer.ffn_norm)
    h = swiglu_ffn(h)
    h = r + h
h = rmsnorm(h, final_norm)           # [B, n, d]
logits = h @ lm_head.T               # [B, n, vocab]
```

2. Reference configurations, all published:

Model	Layers	d_model	Vocab
Transformer base 2017	6 + 6	512	37,000
GPT-2 small 2019	12	768	50,257
Llama 3.1 8B 2024	32	4096	128,256
Llama 3.1 70B 2024	80	8192	128,256

- Total parameter count for Llama 3.1 8B, worked exactly: 32 layers of 218,112,000, plus 525,336,576 for the input embedding, plus 525,336,576 for the output projection, plus 4,096 for the final norm, giving 8,030,261,248.
- Activation memory for a forward pass is separate from weights and is dominated by the FFN hidden state. At $n = 4096$ and bfloat16, one layer's $[1, 4096, 14336]$ gate tensor is 117 MiB, and there are two of them.
- Weight tying is controlled by a flag, `tie_word_embeddings`, in the Hugging Face config. It is true for GPT-2 and for Gemma, false for Llama 3.1 8B.
- The final logit tensor is genuinely large. At $n = 4,096$ and vocabulary 128,256 in float32, the logits alone are 2.1 GiB. Training frameworks compute the loss in chunks along the sequence axis to avoid materializing it.
- At inference only the last position's logits are needed, so implementations slice the hidden state to $[B, 1, d]$ before the `lm_head` multiply. Skipping this is a common and expensive beginner mistake.
- Some models scale the logits. Gemma multiplies the embedding output by the square root of `d_model`; some models divide final logits by a constant, called logit softcapping, which Gemma 2 uses with a cap of 30.0 on the final layer.
- Numerical note: logits are computed in float32 even in bfloat16 models, because cross-entropy over 128,256 classes is sensitive to precision in the log-sum-exp reduction.

■ 48.9.6 WORDS - remember these

- Logit — a raw score before it is turned into a probability — the pre-softmax output, one per vocabulary entry, unbounded in both directions.
- `lm_head` — the final matrix that scores every possible next token — the $[\text{vocab}, d_{\text{model}}]$ output projection, sometimes tied to the input embedding.
- Weight tying — reusing the input embedding table as the output matrix — sharing one $[\text{vocab}, d]$ tensor for both, halving vocabulary parameters.
- Residual stream — the constant-width pipe running the depth of the model — the $[B, n, d_{\text{model}}]$ activation that all blocks read and write.
- Cross-entropy loss — how wrong the guess was, in one number — the negative log probability assigned to the correct next token, averaged over positions.
- Forward pass — running the model once from input to output — one evaluation of the network with no gradient computation.
- Hidden state — the working vector for one token at one depth — a `d_model`-wide slice of the residual stream at a given layer and position.

48.10 The 2017 paper

■ 48.10.1 PLAIN – in simple words

1. The transformer was introduced in one paper: “Attention Is All You Need”.
2. It was posted publicly on 12 June 2017, and presented at the NeurIPS conference in December 2017 in Long Beach, California.
3. There were eight authors: Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan Gomez, Lukasz Kaiser and Illia Polosukhin.
4. Most were at Google Brain or Google Research. Aidan Gomez was at the University of Toronto, working with Google Brain.
5. The paper has an unusual footnote saying the authors contributed equally and the listing order is random. That was deliberate and unusual for the time.
6. It was a machine translation paper. Nothing in it is about chatbots, reasoning or general-purpose assistants.
7. The claim was narrow: you can throw away recurrence entirely and get better translation quality in a fraction of the training time.
8. The headline results were 28.4 BLEU on English-to-German and 41.8 BLEU on English-to-French, both on the WMT 2014 test sets.
9. BLEU is a machine translation score from 0 to 100 that measures overlap with human reference translations. Higher is better.
10. The 41.8 was a new single-model best, reached after 3.5 days on eight GPUs.
11. That last number is the point of the whole paper. Competing systems needed far more compute for worse results.
12. What was new was not attention. Attention had existed since 2014.
13. What was new was deleting the recurrent network and keeping only attention, which made the whole sequence processable in parallel.

■ 48.10.2 PLAIN – a picture in your head

1. Imagine a road crew resurfacing a long road, working one metre at a time, because each metre must set before the next can be laid.
2. Adding more workers does not help. The job is serial by construction.
3. Now imagine somebody invents a material that sets independently everywhere, so a thousand workers can lay a thousand metres simultaneously.
4. The material is not better. One metre of road is one metre of road. What changed is that the work is now divisible.
5. That is what removing recurrence did. It converted a serial job into a parallel one, and hardware that was already sitting idle could finally be used.

Where this comparison breaks: attention is not merely a parallel version of the same computation. It also gives every position direct access to every other position, which recurrence never did. So the change was both a speed change and a capability change, and the paper’s own results reflect both.

■ 48.10.3 PLAIN – a worked example

1. The two model sizes reported:

Setting	Base	Big
Parameters	65 million	213 million
d_model	512	1024
Heads	8	16

Setting	Base	Big
Training time	12 hours	3.5 days

- Both used 6 encoder layers and 6 decoder layers, and were trained on 8 NVIDIA P100 GPUs in one machine.
- The base model ran 100,000 steps at about 0.4 seconds each, which is 12 hours.
- The big model ran 300,000 steps at about 1.0 second each, which is 3.5 days.
- Now the parallelism argument in concrete terms. Take a 1,000-token training sequence.
- A recurrent model must execute 1,000 dependent steps. Each step is a small matrix multiplication that cannot fill a GPU, and between each pair of steps the chip waits.
- A transformer executes one matrix multiplication over all 1,000 positions. With $d_{\text{model}} = 512$, that single call is $2 \times 1000 \times 512 \times 512$, which is 524 MFLOP of dense work handed to the chip in one piece.
- The chip's utilization goes from a few per cent to near its limit, and the speed-up is far larger than the reduction in arithmetic.
- That is why the paper could report state-of-the-art translation from a single 8-GPU machine in under four days.

■ 48.10.4 PLAIN – what is really happening inside

- The architecture in the paper is an encoder-decoder, not the decoder-only design used by chat models today.
- The encoder has 6 layers, each with bidirectional self-attention and a feed-forward network. It reads the source sentence.
- The decoder has 6 layers, each with three sub-layers: causal self-attention over what has been produced so far, cross-attention into the encoder's output, and a feed-forward network.
- Post-norm was used throughout, with a warmup schedule that raised the learning rate for 4,000 steps then decayed it as the inverse square root of the step number.
- Dropout of 0.1 was applied to every sub-layer output and to the sum of the embedding and positional encoding.
- Label smoothing of 0.1 was used, which hurts perplexity and helps BLEU, and the paper says so plainly.
- Decoding used beam search with beam size 4 and length penalty 0.6.
- Every one of those settings is an ordinary engineering choice. None of them is the idea. The idea is one sentence: remove recurrence.

■ 48.10.5 TECHNICAL – the engineer's version

- Reported results, WMT 2014 test sets, from Table 2 of the paper:

System	EN-DE BLEU	EN-FR BLEU
Previous best single	25.16	41.29
Previous best ensemble	26.36	41.16
Transformer base	27.3	38.1
Transformer big	28.4	41.8

- Training cost in floating-point operations: $3.3e18$ for the base model and $2.3e19$ for the big model, both reported in the paper, and both one to two orders of magnitude below the competing systems in the same table.

3. The paper also reported English constituency parsing results, to argue the architecture generalizes beyond translation. Those results were good but not record-setting, and they are rarely cited.
4. The complexity table in the paper is worth knowing. Self-attention costs $O(n^2 d)$ per layer with $O(1)$ sequential operations and a maximum path length of $O(1)$ between any two positions.
5. A recurrent layer costs $O(n d^2)$ with $O(n)$ sequential operations and a maximum path length of $O(n)$.
6. So self-attention is cheaper than recurrence whenever n is less than d , which was true for translation sentences in 2017, where n was around 30 and d was 512.
7. Established fact: the architecture, the results and the training times are all in the published paper and have been reproduced many times.
8. **The honest version:** the paper did not predict what the transformer became. It contains no mention of language modelling at scale, no scaling laws, no in-context learning and no suggestion that this design would generalize to images, audio, protein structure or code.
9. The final sentence of the paper's conclusion is about applying attention to images, audio and video, and about making generation less sequential. Those were reasonable next steps, not a prediction of GPT-3.
10. The name "transformer" was chosen by Jakob Uszkoreit, and by several published accounts the team considered other names first. That is anecdote, not documented in the paper.
11. Reproduction is easy today. The `tensor2tensor` library released alongside the paper contained the original code, and Alexander Rush's annotated walkthrough from Harvard in 2018 reimplements the paper in a few hundred lines.

■ 48.10.6 WORDS - remember these

1. BLEU — a translation score based on word overlap — bilingual evaluation understudy, from Papineni and colleagues at IBM in 2002, scored 0 to 100.
2. WMT — the standard machine translation benchmark — the Workshop on Machine Translation shared task, with the 2014 English-German and English-French sets still used as reference points.
3. NeurIPS — the main machine learning conference — the Conference on Neural Information Processing Systems, called NIPS until 2018.
4. Encoder-decoder — one stack reads, another writes — the original transformer layout, with cross-attention linking the two.
5. Warmup — raising the learning rate slowly at the start — a schedule required by post-norm transformers, 4,000 steps in the 2017 paper.
6. Label smoothing — never demanding 100 per cent confidence — replacing the one-hot target with 0.9 on the correct class and the rest spread out.

48.11 The three architectures

■ 48.11.1 PLAIN – in simple words

1. The 2017 paper had two stacks: an encoder that reads and a decoder that writes. The field then split them apart and used them separately.
2. **Encoder-only**: keep the reading stack, throw away the writing stack. No causal mask, so every word sees every other word in both directions.
3. That is BERT, from Google in October 2018. It is superb at understanding a fixed piece of text and cannot generate text at all.
4. Use it for classification, for search, for named-entity extraction, and for producing embedding vectors of documents.
5. **Decoder-only**: keep the writing stack, throw away the reading stack. Causal mask, so every word sees only what came before.
6. That is the GPT line, and essentially every chat model you have ever used.
7. **Encoder-decoder**: keep both, as in the original. The encoder reads the input, the decoder writes the output, and cross-attention connects them.
8. That is T5, and most dedicated translation and speech-recognition models.
9. Decoder-only won for general-purpose models, and the reason is worth stating plainly.
10. A decoder-only model can do everything by writing. Ask it to classify, and it writes the class name. Ask it to translate, and it writes the translation.
11. Every task becomes text in, text out, so one model and one training objective covers all of them.
12. It also trains on more signal per token, needs no separate cross-attention machinery, and gets a prediction target at every position.

■ 48.11.2 PLAIN – a picture in your head

1. Think of three kinds of worker.
2. The encoder-only worker is a proofreader. Given a finished page, they read it forwards and backwards as many times as they like and tell you what it is about. They never write anything.
3. The decoder-only worker is a novelist writing live, one word at a time, never allowed to see the words they have not written yet.
4. The encoder-decoder worker is a translator. They read the whole source document freely, then write the translation one word at a time, glancing back at the source whenever they need to.
5. The novelist turned out to be the most useful, because you can ask a novelist to write a summary, a classification or a translation, and they will.

Where this comparison breaks: a proofreader could write if asked. BERT genuinely cannot, because it was never trained to predict a next token and its attention has no causal structure to build on. The limitation is architectural and training-based, not a matter of instruction.

■ 48.11.3 PLAIN – a worked example

1. Real models, sorted by architecture:

Model	Year	Architecture
BERT	2018	Encoder-only
RoBERTa	2019	Encoder-only
DeBERTa	2020	Encoder-only
T5	2019	Encoder-decoder
BART	2019	Encoder-decoder

Model	Year	Architecture
Whisper	2022	Encoder-decoder
GPT-2, GPT-3	2019, 2020	Decoder-only
Llama, Mistral	2023 on	Decoder-only
Qwen, Gemma	2023 on	Decoder-only
DeepSeek-V3	2024	Decoder-only

2. Sizes for the encoder models: BERT-base has 110 million parameters, 12 layers and d_model 768. BERT-large has 340 million, 24 layers and d_model 1024.
3. Notice that encoder-only models stopped getting bigger. Nobody built a 100-billion-parameter BERT, because the returns went to generative models.
4. Encoder-only models are still heavily used, and not as a legacy. Almost every semantic search system in production embeds text with a small encoder, because it is cheap, bidirectional and fast.
5. A concrete size comparison for a search system: an encoder producing a 384- dimensional embedding might have 22 million parameters and run in under a millisecond per short document on a CPU.

■ 48.11.4 PLAIN - what is really happening inside

1. The architectural differences reduce to two switches: is attention masked, and is there a second stack with cross-attention.
2. Encoder-only: no mask, one stack, no cross-attention.
3. Decoder-only: causal mask, one stack, no cross-attention.
4. Encoder-decoder: no mask in the encoder, causal mask in the decoder, plus cross-attention layers in the decoder.
5. Everything else, the blocks, the residuals, the normalization, the feed-forward networks, is identical in all three.
6. So they are not three architectures in any deep sense. They are one architecture with different masking and different wiring.
7. The training objective differs too, and that matters more than the wiring.
8. Encoder-only models are trained with masked language modelling, which corrupts about 15 per cent of tokens, giving a learning signal at 15 per cent of positions.
9. Decoder-only models are trained with next-token prediction, which gives a signal at 100 per cent of positions.
10. Encoder-decoder models like T5 use span corruption, dropping contiguous runs of tokens and asking the decoder to reproduce them.
11. That efficiency difference compounds over trillions of tokens, and it is a large part of why the decoder-only line scaled further.

■ 48.11.5 TECHNICAL - the engineer's version

1. T5, from Colin Raffel and colleagues at Google in October 2019, cast every task as text-to-text with a task prefix, and scaled from 60 million to 11 billion parameters.
2. T5 established the text-to-text framing that decoder-only models later adopted without needing the encoder.
3. The empirical comparison exists. The 2022 paper "What Language Model Architecture and Pre-training Objective Work Best for Zero-Shot Generalization?" by Thomas Wang and colleagues found that decoder-only with causal language modelling gave the best zero-shot performance af-

ter pretraining alone, while encoder-decoder with masked objectives was stronger after multitask fine-tuning.

4. So the decoder-only win is not absolute. It is a win under the specific regime that turned out to matter: pretrain once, then prompt.
5. Practical reasons decoder-only dominates in deployment:
 1. One stack means one set of key-value caches, so serving is simpler.
 2. The prompt and the generation share the same cache, so a long prompt costs one prefill pass and nothing more.
 3. Any input format can be expressed as text, so no schema changes are needed per task.
6. Encoder-decoder still wins where the input is genuinely a different modality or language from the output. Whisper, released by OpenAI in September 2022, is an audio encoder with a text decoder, and that split is natural.
7. Encoder-only remains standard for retrieval. The sentence-transformers library, from Nils Reimers and Iryna Gurevych in 2019, is the usual tool, and the MTEB leaderboard is the usual benchmark.
8. A hybrid worth naming: many modern retrieval systems now use decoder-only models with the causal mask removed for embedding, which works but requires adaptation training.

■ 48.11.6 WORDS – remember these

1. Encoder-only — reads text both ways, cannot write — a bidirectional transformer stack trained with masked language modelling.
2. Decoder-only — writes text one token at a time — a causally masked stack trained with next-token prediction.
3. Encoder-decoder — reads one sequence, writes another — two stacks joined by cross-attention, the original 2017 layout.
4. Bidirectional — able to use both left and right context — attention with no causal mask.
5. Span corruption — deleting runs of words and asking for them back — the T5 pretraining objective, replacing spans with sentinel tokens.
6. Text-to-text — every task expressed as text in and text out — the T5 framing, inherited by all current chat models.

48.12 Generation: producing text one token at a time

■ 48.12.1 PLAIN – in simple words

1. The model does not write a sentence. It writes one token, then starts again with that token appended.
2. The loop is simple. Feed in the prompt. Get a probability for every possible next token. Choose one. Append it. Feed in the whole thing again.
3. Stop when a special end-of-text token is chosen, or when a length limit is reached.
4. Every word you have ever seen a language model produce came out of this loop, one token at a time.
5. The interesting decision is step three: choose one. There are several ways.
6. **Greedy**: always take the highest-probability token. Fully deterministic and often dull and repetitive.
7. **Temperature**: divide all the logits by a number before softmax. Below 1 sharpens the distribution towards the top choice. Above 1 flattens it.

8. **Top-k**: keep only the k most likely tokens, throw the rest away, and sample from what remains after renormalizing.
9. **Top-p**, also called nucleus sampling: keep the smallest set of tokens whose probabilities add up to p, typically 0.9 or 0.95.
10. **Min-p**: keep tokens whose probability is at least some fraction of the top token's probability. It adapts to how confident the model is.
11. **Repetition and frequency penalties**: reduce the score of tokens that have already appeared, to stop loops.
12. **Beam search**: keep several candidate continuations alive at once and pick the best complete sequence at the end.
13. This is why the same prompt gives different answers. Unless you use greedy decoding, a random draw happens at every single token.

■ 48.12.2 PLAIN - a picture in your head

1. Picture a weather forecaster who can only forecast one hour ahead.
2. To forecast a whole day, they forecast hour one, write it down, then forecast hour two assuming hour one happened exactly as written.
3. Errors compound. If hour three is wrong, hours four to twenty-four are built on a mistake and cannot recover, because the forecaster treats their own written output as fact.
4. That is autoregressive generation, and it explains why long outputs drift.
5. Temperature is the forecaster's willingness to name an unlikely outcome. At temperature 0 they always say the single most likely thing. At high temperature they will occasionally forecast snow in July.

Where this comparison breaks: the forecaster knows their earlier forecast was a guess. The model has no such flag. Once a token is in the context, it is indistinguishable from text you wrote yourself, which is exactly why a model can confidently build on something it invented three sentences ago.

■ 48.12.3 PLAIN - a worked example

1. Take a real five-way choice. The model has produced "the cat sat on the" and these are the top five candidate logits:

token	logit
mat	4.0
floor	3.2
chair	2.5
roof	1.1
moon	0.4

2. Softmax at temperature 1.0 gives:

mat	0.5699	floor	0.2561	chair	0.1272
roof	0.0314	moon	0.0156		

3. Temperature divides every logit by T before the softmax. Here is the same distribution at several temperatures, computed exactly:

Token	T=0.5	T=1.0	T=2.0
mat	0.7965	0.5699	0.3933
floor	0.1608	0.2561	0.2636

Token	T=0.5	T=1.0	T=2.0
chair	0.0397	0.1272	0.1858
roof	0.0024	0.0314	0.0923
moon	0.0006	0.0156	0.0650

- At $T = 0.5$ the top token has 79.65 per cent, and “moon” has 0.06 per cent. At $T = 2.0$ the top token is down to 39.33 per cent and “moon” is up to 6.5 per cent, a hundredfold increase.
- At $T = 0.1$ the top token reaches 0.9997, which is greedy decoding in all but name. Temperature 0 is usually implemented as exactly greedy, since dividing by zero is undefined.
- Top-k with $k = 2$:** keep “mat” and “floor” only. Renormalize 0.5699 and 0.2561 by their sum 0.8260, giving 0.6900 and 0.3100. “chair” is now impossible.
- Top-p with $p = 0.9$:** accumulate from the top. 0.5699, then 0.8260, then 0.9531 which passes 0.9. So three tokens are kept, renormalized to 0.5979, 0.2687 and 0.1334.
- Note that top-p with 0.9 and 0.95 give the same set here, because the cumulative sum jumps from 0.826 to 0.953 in one step.
- Min-p with 0.1:** the threshold is 0.1 times the top probability, so $0.1 \times 0.5699 = 0.057$. Tokens above that are “mat”, “floor” and “chair”. Same set as top-p here.
- The methods differ on a peaked distribution. If the top token has 0.9947 and the rest share 0.0053, min-p keeps only “mat”, because the threshold is 0.0995 and nothing else clears it. Top-p at 0.9 also keeps only “mat”.
- On a flat distribution where the top is 0.2419, min-p’s threshold is 0.0242 and all five survive, which is the point: the cut adapts to confidence.
- Repetition penalty of 1.2** on a token already generated divides its logit by 1.2 when positive. “mat” goes from 4.0 to 3.3333, and its probability falls from 0.5699 to 0.4048.
- Frequency penalty of 0.5** subtracts 0.5 for each previous occurrence. If “mat” appeared twice, its logit drops from 4.0 to 3.0 and “floor” becomes the most likely token at 0.4002.

■ 48.12.4 PLAIN – what is really happening inside

- Generation has two phases, and they behave completely differently.
- Prefill:** the whole prompt goes through the model in one pass. All prompt positions are computed in parallel. This is compute-bound and fast per token.
- Decode:** each new token requires a full forward pass over one position. The model reads every weight in the file to produce one token.
- That is why decode is memory-bandwidth-bound. For an 8-billion model at 16-bit, each token requires reading 16 GB of weights from memory.
- On a card with 2,000 GB/s of memory bandwidth, that sets a hard ceiling of about 125 tokens per second for a single sequence, before any other cost.
- Beam search keeps b partial sequences alive, extends each by every candidate token, keeps the b best by total log probability, and repeats.
- It optimizes for the highest-probability complete sequence, which is exactly what you want in translation, where there is one correct output.
- It is wrong for chat for two reasons. The highest-probability continuation of open-ended text is bland and repetitive, a result documented by Ari Holtzman and colleagues in 2019. And beam search costs b times the compute and b times the key-value cache.
- So translation systems use beam 4 or 5. Chat systems use sampling with temperature and top-p, and never beam search.
- Why does the same prompt give different answers? Three separate reasons.

11. First, sampling. Every token is a random draw, so different random seeds give different text.
12. Second, even at temperature 0, floating-point addition is not associative. Changing the batch size changes the order of reductions inside the matrix multiplications, which changes the last bits of the logits, which can flip a near-tie between two tokens.
13. Third, providers change models, system prompts and serving configurations without notice. Determinism across dates is not offered by any major API.

■ 48.12.5 TECHNICAL – the engineer’s version

1. The autoregressive factorization: $P(x_1 \dots x_n) = \text{product over } t \text{ of } P(x_t \text{ given } x_1 \dots x_{\{t-1\}})$. Every generation method is a search or sampling strategy over this factorization.
2. Temperature applies as $\text{softmax}(z / T)$. It is a monotonic transform, so it never changes the ranking, only the gaps.
3. Method origins:

Method	Year	Source
Beam search	pre-1980s	speech recognition
Top-k	2018	Fan, Lewis, Dauphin
Top-p (nucleus)	2019	Holtzman and colleagues
Repetition penalty	2019	Keskar and colleagues, CTRL
Min-p	2024	Nguyen and colleagues

4. Typical production defaults as of 2026: temperature 0.7 to 1.0, top_p 0.9 to 1.0, top_k either 40 or disabled, repetition penalty 1.0 to 1.1. These are conventions, and every provider picks slightly different ones.
5. OpenAI’s API exposes `frequency_penalty` and `presence_penalty`, both in the range -2.0 to 2.0, applied additively to logits. Presence penalty applies once if a token has appeared at all; frequency penalty scales with the count.
6. The Hugging Face `generate` method applies processors in a fixed order: penalties first, then temperature, then top-k, then top-p, then min-p, then the sample. Order matters and is worth checking when results surprise you.
7. Greedy decoding is `do_sample=False`. Setting temperature to 0 in most APIs is mapped internally to greedy.
8. Speculative decoding, from Yaniv Leviathan, Matan Kalman and Yossi Matias at Google in 2022 and independently from DeepMind, uses a small draft model to propose several tokens and the large model to verify them in one pass. It is exact, meaning the output distribution is unchanged, and gives two to three times speedup in practice.
9. Constrained decoding masks logits to force valid JSON or a regular expression. Libraries such as `Outlines` and `llama.cpp`’s GBNF grammars do this by zeroing the probability of any token that would break the grammar.
10. Established fact: sampling parameters change output distribution in exactly the ways computed above. Active research: which defaults are best for which task. Marketing claim: any statement that a specific temperature makes a model “more creative” in a measurable, general sense.

■ 48.12.6 WORDS – remember these

1. Autoregressive generation — writing one token at a time, feeding each back in — sampling from $P(x_t \text{ given all previous tokens})$ in a loop.
2. Greedy decoding — always take the most likely token — `argmax` over logits, fully deterministic given identical arithmetic.

3. Temperature — a dial that flattens or sharpens the choice — the divisor T in $\text{softmax}(z / T)$, with T below 1 sharpening.
4. Top-k sampling — only consider the k best candidates — truncate to the k highest logits, renormalize, sample.
5. Top-p sampling — consider just enough candidates to cover p of the mass — nucleus sampling, keeping the smallest prefix of the sorted distribution summing to at least p .
6. Min-p sampling — keep anything within a fraction of the top choice — a threshold of p times max probability, adapting to model confidence.
7. Beam search — keep several drafts alive and pick the best — breadth-limited search over sequences, standard for translation, unsuitable for chat.
8. Prefill — processing the whole prompt at once — the parallel phase of inference, compute-bound.
9. Decode — producing one token per forward pass — the sequential phase, memory-bandwidth-bound.
10. Speculative decoding — a small model guesses, a big model checks — a draft-and-verify scheme that preserves the output distribution exactly.

48.13 The cost of attention: the quadratic problem

■ 48.13.1 PLAIN – in simple words

1. Attention compares every token with every token. For n tokens that is n times n comparisons.
2. Double the length and the work quadruples. Ten times the length and the work goes up a hundredfold.
3. That is what “quadratic” means, and it is the single biggest engineering problem in the architecture.
4. At 1,000 tokens it is a million comparisons, which is nothing.
5. At 1,000,000 tokens it is a trillion comparisons, which is an enormous amount for a single layer of a single head.
6. There is a second cost, separate and often larger in practice: the **key-value cache**.
7. When generating, the model has already computed the keys and values for every earlier token. Recomputing them every step would be wasteful, so they are stored.
8. That store grows with every token generated, and it must live in fast memory next to the chip.
9. For a 70-billion-parameter model, every token of context costs 320 kilobytes of cache. A 128,000-token conversation costs 40 gigabytes.
10. That is more than the model itself in some configurations, and it is why long-context serving is expensive.
11. There are four families of fix, and three of them are in production today.
12. Make the exact computation memory-smarter (FlashAttention). Share keys and values across heads (grouped-query attention). Restrict which tokens can be looked at (sliding window, sparse). Or replace attention with something cheaper (linear attention, state-space models).

■ 48.13.2 PLAIN – a picture in your head

1. Picture a party where every guest must shake hands with every other guest.
2. With 10 guests that is 45 handshakes. Manageable.
3. With 1,000 guests it is 499,500. With 100,000 guests it is about 5 billion.
4. The number of guests grew by a factor of 100 and the handshakes grew by a factor of 10,000.
5. Nobody solves this by shaking hands faster. You solve it by changing the rules: only shake hands with the nearest twenty people, or shake hands in small groups, or appoint representatives.

- Every one of those is a real technique in the list of fixes.

Where this comparison breaks: a handshake is symmetric and attention is not. Token 5 attending to token 2 is a different computation from token 2 attending to token 5, and in a causal model only one of the two happens at all. So the true count is $n(n+1)/2$, not $n(n-1)/2$, and the arithmetic differs by a factor of about two rather than the shape being different.

■ 48.13.3 PLAIN – a worked example

- The score matrix has n squared entries. Here is what that means, with the memory it would take at 16 bits per number, for a single head in a single layer:

Tokens	Entries	Memory, one head
1,024	1,048,576	2 MB
32,768	1,073,741,824	2,048 MB
131,072	17,179,869,184	32,768 MB
1,048,576	1,099,511,627,776	2,097,152 MB

- Read the last row again. Two terabytes, for one head, in one layer, for one sequence.
- A model like Llama 3.1 8B has 32 heads in each of 32 layers. If you tried to store the score matrices for a 131,072-token sequence in one layer, that is 32 times 32,768 MB, which is 1,024 GiB.
- No accelerator has that memory. This is why the naive implementation simply cannot run at long context, and why FlashAttention exists.
- Now the cost of attention relative to the rest of the model. The score computation costs about $4n$ squared d operations per layer, and the feed-forward network costs about $16nd$ squared.
- Setting those equal gives a crossover at $n = 4d$. For Llama 3.1 8B with $d = 4096$, that is 16,384 tokens.
- Below 16,384 tokens the feed-forward network dominates the compute. Above it, attention does. Measured for that configuration:

Sequence length	Attention share
512	2 per cent of FFN
4,096	19 per cent of FFN
24,576	114 per cent of FFN
131,072	609 per cent of FFN

- Now the key-value cache, computed exactly. The formula is $2 \times \text{layers} \times \text{kv\_heads} \times \text{head\_dim} \times \text{bytes}$, per token.
- For Llama 3.1 8B: $2 \times 32 \times 8 \times 128 \times 2 = 131,072$ bytes per token, which is exactly 128 KiB.

Context	Llama 3.1 8B	Llama 3.1 70B
2,048 tokens	0.25 GiB	0.62 GiB
32,768 tokens	4.00 GiB	10.00 GiB
131,072 tokens	16.00 GiB	40.00 GiB

- Without grouped-query attention, using 32 key-value heads instead of 8, the 8B model's cache would be 512 KiB per token, so 64 GiB at full context instead of 16 GiB. That single design choice saves four times the memory.

11. Batching multiplies this. Serving 32 concurrent conversations of 8,192 tokens each on Llama 3.1 8B costs 32 GiB of cache, on top of 16 GB of weights.
12. That is the actual reason long context costs more per token in commercial APIs. It is memory, not arithmetic.

■ 48.13.4 PLAIN – what is really happening inside

1. **FlashAttention** is the most important fix, and it is important to understand what it is not. It is not an approximation. Its output is exactly the same as the naive implementation, bit-for-bit equivalent up to floating-point reassociation.
2. The insight is that the n -by- n score matrix never needs to exist all at once.
3. The naive implementation writes the whole score matrix to the chip's main memory, reads it back for the softmax, writes it again, and reads it again for the multiply by V .
4. Those trips to main memory are the bottleneck, not the arithmetic. Chapter 22 covered why: a GPU's arithmetic units are far faster than its memory.
5. FlashAttention splits Q , K and V into tiles that fit in the chip's small fast on-chip memory, and computes the output tile by tile, keeping a running softmax normalizer as it goes.
6. The running softmax is possible because of an algebraic trick published by Maxim Milakov and Natalia Gimelshein in 2018, called online softmax, which lets you update a softmax result as new terms arrive.
7. The score matrix is therefore never stored. Memory use drops from order n squared to order n , and the speed rises because far fewer bytes move.
8. **Grouped-query attention** attacks the cache instead. Instead of 32 separate key and value heads, use 8, and let four query heads share each one.
9. The cache shrinks by exactly that ratio, and the quality loss is small.
10. **Sliding window attention** simply forbids looking back more than w tokens. Cost becomes linear in n , and information travels further only by hopping layer by layer, at w tokens per layer.
11. **Linear attention** rewrites the formula so the value sum can be accumulated without ever forming the score matrix, giving linear cost at the price of a weaker, approximate similarity function.

■ 48.13.5 TECHNICAL – the engineer's version

1. FlashAttention, from Tri Dao, Daniel Fu, Stefano Ermon, Atri Rudra and Christopher Re, May 2022. FlashAttention-2 from Tri Dao, July 2023, improved work partitioning. FlashAttention-3, July 2024, targets Hopper-class hardware with asynchrony and low-precision paths.
2. Complexity: standard attention uses $O(n^2)$ memory. FlashAttention uses $O(n)$ and reduces high-bandwidth-memory accesses from $O(n^2 d)$ to about $O(n^2 d^2 / M)$, where M is on-chip SRAM size. Reported speedups on training are roughly 2x to 4x end to end.
3. It recomputes the attention matrix during the backward pass rather than storing it, trading a little arithmetic for a large memory saving. This is gradient checkpointing applied at kernel level.
4. Multi-query attention, from Noam Shazeer in November 2019, uses one key-value head for all query heads. Grouped-query attention, from Joshua Ainslie and colleagues at Google in May 2023, interpolates between the two.

Scheme	KV heads (32 q heads)	Cache size
Multi-head (MHA)	32	1.0x
Grouped-query (GQA)	8	0.25x
Multi-query (MQA)	1	0.031x

5. Multi-head latent attention, introduced in DeepSeek-V2 in 2024 and used in DeepSeek-V3, compresses keys and values into a low-rank latent vector and reconstructs them, cutting cache further than GQA at comparable quality.
6. Sliding window attention: Longformer, from Iz Beltagy, Matthew Peters and Arman Cohan in April 2020, combined a local window with a few global tokens. Mistral 7B, October 2023, used a 4,096-token window with 32 layers, giving a theoretical receptive field of 131,072 tokens.
7. Sparse attention: the Sparse Transformer, from Rewon Child, Scott Gray, Alec Radford and Ilya Sutskever in April 2019, used strided and fixed patterns to reach $O(n \sqrt{n})$. BigBird, from Manzil Zaheer and colleagues in July 2020, combined window, global and random attention and proved it is a universal approximator.
8. Linear attention: Angelos Katharopoulos and colleagues in June 2020 showed that replacing softmax with a kernel feature map lets you reorder the multiplications to get $O(n)$ cost, and that the resulting model is a recurrent network at inference. The paper is titled “Transformers are RNNs”.
9. Performer, from Krzysztof Choromanski and colleagues in September 2020, approximates the softmax kernel with random features. Reformer, from Nikita Kitaev, Lukasz Kaiser and Anselm Levskaya in January 2020, used locality-sensitive hashing.
10. **The honest version:** the approximate methods have largely not displaced exact attention in frontier models. Exact attention plus FlashAttention plus GQA is what production systems actually run as of 2026. The approximations remain valuable in specific niches.
11. State-space models are the serious current alternative. Mamba, from Albert Gu and Tri Dao in December 2023, uses a selective state-space layer with linear cost in sequence length and a constant-size state, so there is no growing key-value cache at all.
12. Mamba-2 followed in 2024 and showed a formal duality between state-space models and a restricted form of attention. Jamba, from AI21 Labs in March 2024, is a production hybrid interleaving Mamba and attention layers.
13. Active research, stated plainly: hybrid models are being trained at scale and show competitive quality with much better long-context economics. Whether they displace attention entirely is not known. Pure state-space models have measured weaknesses on tasks requiring precise recall of arbitrary earlier tokens, which is exactly what attention is good at.
14. Observation tooling: `nvidia-smi` shows memory use, but for cache accounting the useful figures come from the serving framework. vLLM, which implements PagedAttention from Woosuk Kwon and colleagues in 2023, reports cache block usage directly and manages the cache in fixed-size pages like an operating system manages virtual memory.

■ 48.13.6 WORDS - remember these

1. Quadratic complexity — work grows with the square of the length — $O(n^2)$ in sequence length for the score matrix and the weighted sum.
2. KV cache — stored keys and values so earlier tokens are not recomputed — $2 \times \text{layers} \times \text{kv\_heads} \times \text{head\_dim} \times \text{bytes per token of context}$.
3. FlashAttention — computing attention without ever storing the score matrix — an exact, IO-aware tiled kernel with online softmax and backward recomputation.
4. Online softmax — updating a softmax as new terms arrive — a rescaling recurrence that avoids a second pass over the scores.
5. Multi-query attention — one key-value head shared by all query heads — MQA, cutting cache by the number of heads at some quality cost.
6. Grouped-query attention — a few key-value heads shared in groups — GQA, the standard compromise, 8 KV heads for 32 query heads in Llama 3.

7. Sliding window attention — only look back a fixed distance — local attention with window w , giving linear cost and a layer-multiplied receptive field.
8. State-space model — a linear recurrence with a fixed-size state — the Mamba family, linear in sequence length with no growing cache.
9. PagedAttention — managing the cache in fixed-size blocks — the vLLM technique that removes cache fragmentation, modelled on virtual memory paging.

48.14 Mixture of experts: paying for a big model, running a small one

■ 48.14.1 PLAIN – in simple words

1. Every transformer block has two halves. Attention mixes information between tokens. The feed-forward network then thinks about each token on its own.
2. In an ordinary model there is exactly one feed-forward network per block, and every token goes through it. That block holds most of the model's numbers.
3. A **mixture of experts** model replaces that one feed-forward network with many copies of it, called **experts**, sitting side by side.
4. A tiny extra network called the **router** looks at each token and picks a small number of experts for it. Usually two, four or eight out of dozens.
5. Only the chosen experts run. The rest sit still and cost nothing in arithmetic for that token.
6. So the model is enormous when you count its numbers, and small when you count the work done for any one token.
7. This gives two different parameter counts, and the difference between them is the whole point of the idea.
8. **Total parameters** is every number in the file. It sets how much memory you need and how much a copy costs to store and to move.
9. **Active parameters** is how many numbers actually take part in producing one token. It sets the speed, the electricity bill and the training cost.
10. A real example: DeepSeek-V3, released December 2024, holds 671 billion parameters in total and uses 37 billion for each token. That is 5.5 per cent.
11. So it costs as much arithmetic per token as a 37-billion model, while holding the knowledge capacity of something eighteen times larger.
12. Nothing is free. You must keep all 671 billion numbers loaded, because you cannot know in advance which experts the next token will want.
13. That is the trade in one line: you pay in memory, and you are paid back in speed.
14. There is one nasty problem. Left alone, the router learns to send almost every token to the same two or three favourite experts, and the rest die.
15. Fixing that needs a deliberate extra pressure during training, and most of the engineering in this area is about exactly that.

■ 48.14.2 PLAIN – a picture in your head

1. Picture a large hospital with 64 specialist doctors and one triage nurse at the door.
2. You walk in. The nurse spends two seconds on you and sends you to two of the 64 doctors. You see those two. The other 62 never learn you existed.
3. The hospital still has to employ all 64, pay all 64, and keep 64 consulting rooms heated. That is the memory cost.
4. But your visit only occupies two doctors' time. That is the compute cost.
5. Now the failure. Suppose the nurse sends nearly everyone to Doctor Smith because Doctor Smith seemed good on the first day.
6. Doctor Smith sees thousands of patients and gets very good. The other 63 see nobody and stay bad. So the nurse sends even more people to Smith.
7. You have not built a hospital with 64 specialists. You have built an expensive building containing one overworked doctor.
8. To stop this the hospital management adds a rule: the nurse is penalized when the workload is uneven, whatever the nurse thinks is best.
9. That rule is real, and it has a name in the papers: a load balancing loss.

Where this comparison breaks: real experts are not topic specialists, and this is the single most over-sold part of the idea. The Mixtral paper of January 2024 looked for topic-based routing and did not find it. Tokens about medicine do not reliably go to a “medicine expert”. What routing does correlate with is much duller: token identity, position, and surface patterns such as indentation in code. Also, a patient sees one doctor and gets one opinion. A token gets a weighted blend of two experts' outputs, added together in proportion to the router's confidence.

■ 48.14.3 PLAIN – a worked example

1. We will build a mixture-of-experts model from a normal one, count the parameters exactly, and land on a real published model.
2. Start with the dense shape used earlier in this chapter: d_{model} 4096, feed forward width 14,336, SwiGLU with three matrices, 32 layers, 8 key-value heads, vocabulary 32,000.
3. One feed-forward network is $3 \times 4096 \times 14336 = 176,160,768$ parameters.
4. Attention in one layer is 41,943,040 parameters. Two norms are 8,192.
5. Now replace the single feed-forward network with 8 copies, and add a router that maps 4096 numbers to 8 scores: $4096 \times 8 = 32,768$ parameters.
6. The expert bank in one layer is $8 \times 176,160,768 = 1,409,286,144$.
7. One layer therefore holds:

attention	41,943,040
experts (8)	1,409,286,144
router	32,768
two norms	8,192
layer total	1,451,270,144
8. Thirty-two layers give 46,440,644,608. Add the two vocabulary tables, $2 \times 32,000 \times 4,096 = 262,144,000$, and a final norm of 4,096.
9. Total: 46,702,792,704 parameters, which is 46.7 billion.
10. That is Mixtral 8x7B, published by Mistral AI in December 2023, whose stated size is 46.7B. The arithmetic reproduces the published number exactly.
11. Now count what one token actually touches, with top-2 routing.

attention	41,943,040
2 experts of 8	352,321,536
router	32,768
two norms	8,192
per layer active	394,305,536
x 32 layers	12,617,777,152
+ two vocab tables	262,144,000
active total	12,879,921,152

12. That is 12.9 billion active parameters, which is also exactly the published figure. The ratio is 46.7 to 12.9, or 3.6 to 1.
13. Note what did not change: attention is dense. Every token runs the full attention block. Only the feed-forward half is sparse.
14. Now push the idea harder. Cut each expert into 8 narrower ones, so 64 experts, and route to 8 instead of 2.
15. The arithmetic per token is identical, because 8 narrow experts equal 2 wide ones. But the number of possible expert combinations explodes.
16. Choosing 2 from 8 gives 28 combinations. Choosing 8 from 64 gives 4,426,165,368. That is the argument for fine-grained experts.
17. Here is what real models do, as published by each laboratory:

Model	Total params	Active per token
Mixtral 8x7B, 2023	46.7B	12.9B
DBRX, 2024	132B	36B
Mixtral 8x22B, 2024	141B	39B
Grok-1, 2024	314B	about 79B
DeepSeek-V3, 2024	671B	37B
Llama 4 Scout, 2025	109B	17B
Qwen3-235B-A22B, 2025	235B	22B
Llama 4 Maverick, 2025	400B	17B
gpt-oss-120b, 2025	117B	5.1B
Kimi K2, 2025	1,000B	32B

18. Now the same table as one number, the fraction of the model used per token, sorted by date. The trend is the story:

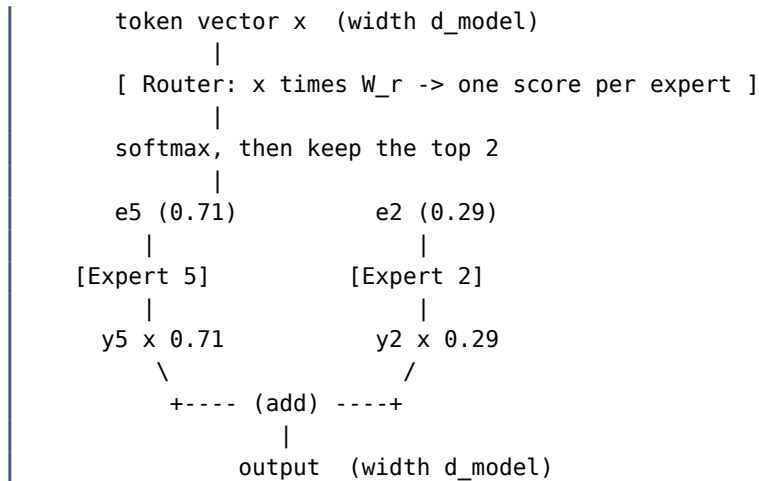
Model	Year	Active fraction
Mixtral 8x7B	2023	27.6 per cent
Grok-1	2024	25.0 per cent
DeepSeek-V3	2024	5.5 per cent
Llama 4 Scout	2025	15.6 per cent
Qwen3-235B-A22B	2025	9.4 per cent
gpt-oss-120b	2025	4.4 per cent
Kimi K2	2025	3.2 per cent

19. Models are getting sparser. In 2023 a mixture-of-experts model used about a quarter of itself per token. By 2025, three per cent was normal.
20. Finally the memory bill. Mixtral 8x7B at 16 bits is 93.4 gigabytes of weights, which does not fit on one 80-gigabyte accelerator.

21. A dense model of the same speed, 12.9 billion parameters, would be 25.8 gigabytes and would fit on one card with room for a cache.
22. So you pay roughly 3.6 times the memory for roughly 1 times the arithmetic. Whether that is a good deal depends entirely on how many users you serve.

■ 48.14.4 PLAIN – what is really happening inside

1. Here is the whole mechanism for one token at one layer.



2. The router is one small matrix, of shape d_{model} by number of experts. For Mixtral that is 4096 by 8. It is trained by ordinary gradient descent.
3. It produces one score per expert. A softmax turns those into probabilities.
4. **Top-k gating** keeps the k highest, throws the rest away, and renormalizes the survivors so they add to 1.
5. Only the surviving experts are run. Their outputs are multiplied by their renormalized weights and added together.
6. The gate weight is what makes this trainable. Without it, the choice of expert would be a hard, non-differentiable decision with no gradient.
7. With it, the gradient can flow back through the weight and tell the router “this expert helped, raise its score” or “this one hurt, lower it”.
8. Now the collapse. The router and the experts learn at the same time, which creates a feedback loop.
9. An expert that gets slightly more tokens early gets more gradient updates, improves faster, produces better outputs, and so earns higher router scores.
10. That earns it more tokens still. Within a few thousand steps a handful of experts absorb nearly everything and the others are effectively dead weight.
11. The standard fix is a **load balancing loss**: an extra number added to the training loss that is smallest when every expert gets an equal share.
12. It pushes against the router’s preference. The model is being told, in effect, “spread the work, even if you would rather not”.
13. There is a second, purely practical constraint. On real hardware every expert is given a fixed-size buffer, called its **capacity**.
14. If more tokens are routed to an expert than its capacity allows, the extra tokens are **dropped**: they skip the feed-forward network entirely and pass through on the residual connection only.
15. Dropped tokens are not an error. They are a deliberate trade that keeps the tensor shapes fixed so the hardware stays efficient.

16. Finally, where the experts physically live. A 671-billion-parameter model does not fit on one accelerator, so the experts are split across machines.
17. That is **expert parallelism**. Each device holds a few experts. Every token must be shipped to whichever devices hold its chosen experts, and the results shipped back.
18. So each mixture-of-experts layer contains two network exchanges: one to dispatch tokens, one to collect answers. On a slow network this, and not the arithmetic, becomes the limit.

■ 48.14.5 TECHNICAL – the engineer’s version

1. History. The idea is from Robert Jacobs, Michael Jordan, Steven Nowlan and Geoffrey Hinton, “Adaptive Mixtures of Local Experts”, Neural Computation, 1991. It predates the transformer by twenty-six years.
2. Noam Shazeer and colleagues, including Hinton and Jeff Dean, revived it for deep learning in January 2017 with “Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer”, reaching 137 billion parameters in an LSTM, with noisy top-k gating.
3. GShard, from Dmitry Lepikhin and colleagues at Google in June 2020, put MoE layers inside a transformer and trained a 600-billion-parameter translation model with automatic sharding.
4. Switch Transformer, from William Fedus, Barret Zoph and Noam Shazeer, January 2021, simplified routing to top-1 and scaled to 1.6 trillion parameters. Switch-C matched T5-Large’s compute per sequence at 2,000 times the size.
5. GLaM, from Nan Du and colleagues at Google, December 2021: 1.2 trillion total parameters, about 97 billion active per token, 64 experts per MoE layer, top-2. It reported better zero-shot quality than GPT-3 at about one third of the training energy.
6. ST-MoE, from Barret Zoph and colleagues, February 2022, introduced the router z-loss and produced ST-MoE-32B at 269 billion parameters.
7. Routing mathematics. With router matrix W_r , gate $g = \text{softmax}(x W_r)$, top-k index set T , and renormalized weights $g_i / \sum_{j \in T} g_j$:

```

scores  = x @ W_r                # [n_tok, n_exp]
probs   = softmax(scores, dim=-1)
val, idx = topk(probs, k)         # weights, expert ids
val     = val / val.sum(-1, keepdim=True)
y = zeros_like(x)
for slot in range(k):
    for e in range(n_exp):
        m = idx[:, slot] == e
        y[m] += val[m, slot, None] * expert[e](x[m])

```

8. The Switch Transformer auxiliary loss, which is still the default in most frameworks:

```

f_i = fraction of tokens in the batch routed to expert i
P_i = mean router probability assigned to expert i
L_aux = alpha * N * sum over i of ( f_i * P_i )
alpha = 0.01, N = number of experts
minimum value 1.0, reached when all f_i = P_i = 1/N

```

9. The product $f_i P_i$ is the trick. f_i is a count and has no gradient. P_i is differentiable. Multiplying them gives the router a gradient proportional to how overloaded each expert already is.
10. Router z-loss, from ST-MoE: the mean of the squared log-sum-exp of the router logits, with coefficient 0.001. It keeps logits small, which improves numerical stability in bfloat16 and reduces round-off in the softmax.
11. **Expert capacity** = (tokens per batch / number of experts) $\times$ capacity factor. Typical capacity factors are 1.0 to 1.25 in training and higher at evaluation. Overflow tokens are dropped to the residual.

12. Expert Choice routing, from Yanqi Zhou and colleagues at Google in 2022, inverts the problem: each expert picks its top tokens instead of each token picking its experts. Balance is then perfect by construction, but the method is not causal-safe for autoregressive decoding without care.
13. MegaBlocks, from Trevor Gale and colleagues in 2022, reformulates MoE as block-sparse matrix multiplication so no token is ever dropped and no padding is needed. It is used in several production stacks.
14. DeepSeek-V3, December 2024, made two notable changes. **Fine-grained experts**: 256 routed experts of intermediate width 2048, top-8, plus one **shared expert** that every token always uses, to hold common knowledge.
15. And **auxiliary-loss-free load balancing**: instead of an extra loss term, a per-expert bias is added to the routing scores and adjusted between steps, raised for underloaded experts and lowered for overloaded ones. The bias affects selection only, never the gate weight, so it adds no gradient interference.
16. Its reported training cost was about 2.788 million H800 GPU-hours on 14.8 trillion tokens. The compute rule C is about 6ND, covered in Chapter 49, uses the ACTIVE parameter count. That substitution is the entire economic case for MoE.
17. Expert parallelism. Experts are sharded across devices and each MoE layer performs two all-to-all collectives, dispatch and combine. Communication per token per layer is about $2 \times k \times d_{\text{model}} \times \text{bytes across the fabric}$.
18. DeepSeek-V3 bounded this with node-limited routing, capping each token at 4 nodes, and overlapped communication with computation using a scheme it calls DualPipe. Frameworks that implement expert parallelism include DeepSpeed-MoE from Samyam Rajbhandari and colleagues in 2022, Tutel from Microsoft, and Megatron-Core.
19. Serving arithmetic intensity. At batch size 1 you read only k experts. At batch size 256 the tokens collectively select nearly every expert, so you read the whole expert bank anyway. MoE therefore looks cheap at small batch and behaves closer to dense at large batch.

Setting	Weights read per step
Dense 13B, any batch	26 GB at bf16
MoE 47B, batch 1	about 27 GB
MoE 47B, batch 256	up to 93 GB

20. **Established fact.** Sparse models reach a lower pretraining loss than dense models given equal training compute; this was measured by GShard, Switch and GLaM and has been reproduced many times. Active parameters set FLOPs and training cost. Total parameters set memory. All experts must be resident. Unbalanced routing occurs reliably without a balancing mechanism.
21. **Active research.** How to compare an MoE against a dense model fairly, and what the right ratio of total to active parameters is. Whether experts specialize in any human-meaningful way; the current evidence is weak. Routing without auxiliary losses. Expert merging and pruning after training. Whether the shared-expert design is genuinely better or merely convenient.
22. **Vendor claim.** Headline parameter counts. A “1 trillion parameter model” that activates 32 billion is not equivalent to a dense trillion-parameter model, and no laboratory has demonstrated that it is. A common informal rule of thumb places MoE quality near the geometric mean of active and total parameters, which for Mixtral 8x7B would be about 24.5 billion; this is a heuristic, not a measured law, and should not be quoted as one.
23. Also vendor claim, by omission: GPT-4 has been widely reported to use a mixture-of-experts design since 2023, but OpenAI has never confirmed its architecture or parameter count. Treat any

specific number you see for it as unverified.

24. Observation. For an open-weights model, safetensors index files list every expert tensor by name, so total parameters are directly countable. Active parameters must be derived from the config: `num_local_experts` and `num_experts_per_tok` in a Hugging Face config.json give `k` and `N`. Serving frameworks such as vLLM and SGLang report expert-parallel placement and per-expert token counts, which is how you detect a collapsed router in production.

■ 48.14.6 WORDS – remember these

1. Mixture of experts — many small networks with a chooser in front — MoE, replacing a dense feed-forward layer with `N` experts and a learned router.
2. Expert — one of the interchangeable feed-forward networks — a standard FFN or SwiGLU block, one of `N` per MoE layer.
3. Router, or gate — the small network that picks experts — a `d_model` by `N` matrix followed by softmax and top-`k` selection.
4. Top-`k` gating — take the `k` best experts and ignore the rest — hard sparse selection with renormalized gate weights, `k` typically 1, 2 or 8.
5. Total parameters — every number in the model file — the figure that sets memory, storage and download size.
6. Active parameters — the numbers used for one token — the figure that sets FLOPs per token, latency and training compute in `C` is about $6ND$.
7. Load balancing loss — an extra penalty for using experts unevenly — the Switch auxiliary loss $\alpha N \sum f_i P_i$, α typically 0.01.
8. Router z-loss — a penalty that keeps router scores small — the mean squared log-sum-exp of router logits, coefficient about 0.001, for stability.
9. Expert capacity — the fixed number of tokens an expert will accept — buffer size = tokens/experts x capacity factor; overflow tokens are dropped.
10. Token dropping — skipping the feed-forward step for overflow tokens — those tokens pass through on the residual connection unchanged.
11. Shared expert — an expert every token always uses — the DeepSeek design for holding knowledge common to all inputs, alongside the routed experts.
12. Fine-grained experts — many narrow experts instead of few wide ones — same FLOPs, far more possible combinations, 64-choose-8 versus 8-choose-2.
13. Expert parallelism — experts split across devices — sharding along the expert axis, requiring two all-to-all collectives per MoE layer.
14. Sparse upcycling — turning a trained dense model into an MoE — copying the trained FFN into `N` experts and continuing training from there.

48.15 Why one architecture does so many tasks

■ 48.15.1 PLAIN – in simple words

1. This is the question most people actually want answered. One machine writes code, translates Hindi, summarizes a contract and holds a conversation. Why?
2. The short answer is that there is only one training task, and doing it well happens to require almost everything else.
3. The task is **next-token prediction**. Show the model some text, hide what comes next, make it guess, and correct it. Repeat trillions of times.
4. There is no list of skills. There is no separate summarizing module and no translation module. There is one number being pushed down: the prediction error.
5. Now the important argument. To predict text well, you must model whatever produced that text.
6. To finish “The capital of France is” you need a fact. Guessing needs knowledge, so knowledge lowers the error, so training installs knowledge.
7. To finish “for i in range(10):” you need to know what an indented block looks like. So the training installs the shape of Python.
8. To finish the last page of a detective novel you must have worked out who did it from the clues. So training rewards keeping track of a plot.
9. To finish a polite email you need to know how polite emails go. So training installs style and register.
10. Grammar, facts, arithmetic, tone, the structure of arguments, the layout of a table: every one of them lowers the prediction error, so every one of them gets learned.
11. Then the second half of the answer, which is easy to miss. Text on the internet already contains every task, written out in full.
12. An article is followed by its summary. An English sentence is followed by its French translation. A question is followed by an answer. A bug report is followed by a fix.
13. So when the text is arranged that way, predicting the next token IS doing the task. The task was never separate. It was always just more text.
14. That covers translation, summarizing and code. Then there is one behaviour that genuinely surprised the people who built it.
15. **In-context learning**. Put three worked examples in the prompt and the model does the fourth, for a task nobody trained it on and that may not exist anywhere in its training data.
16. Nothing inside the model changes. No number is updated. The examples sit in the input and the model behaves as if it had been taught. That was not designed. It was noticed.
17. Then **instruction following**: you write “Summarize this in three bullet points” and it obeys. Here we must be honest.
18. That is mostly not from pretraining. A raw pretrained model given that instruction is quite likely to continue with another instruction, because that is what usually follows an instruction on a web page.
19. Obedience is added afterwards, in a separate stage, using human-written examples and human preference data. Chapter 49 covers exactly how.
20. Finally, **emergence**: the claim that some abilities are absent in small models and appear suddenly in large ones. This one is genuinely disputed, and we will give both sides.

■ 48.15.2 PLAIN – a picture in your head

1. Imagine a person locked in the world’s largest library, playing one game, forever.
2. The game: open any book at any page, cover the next word, guess it, then uncover it and see if you were right. Score kept. No other instructions.

3. On day one they guess badly. Soon they learn that “the” is common, that a full stop is usually followed by a capital letter.
4. Keep going. Now they are playing on a chemistry textbook. To guess the word after “sodium reacts violently with” they have to learn chemistry.
5. Keep going. Now a French-English phrasebook, where every line is the same sentence twice. To win, they must learn to translate.
6. Keep going. Now the last page of a murder mystery. To guess the name after “the killer was” they must have followed the clues and solved it.
7. Nobody ever told them to learn chemistry, French or detective work. They learned all three because each one improved their score at one game.
8. That is the whole explanation of why one architecture does so many things. The tasks were never taught. They were the only way to win.

Where this comparison breaks: the person in the library knows they are playing a game, wants to win, and could stop. The model has none of that. It has no goal, no awareness that it is guessing, and no sense of being right or wrong. A person who does not know says so; the model always produces a full probability distribution over every possible next token, and something is always on top. Also, the person learns by understanding. The model learns by a mechanical adjustment of billions of numbers in the direction that lowers one error term.

■ 48.15.3 PLAIN – a worked example

1. Here is how four different-looking jobs become the same job. Each block is just text, and the model only ever continues it.

```
[ translation ]
English: the sea otter is asleep
French:

[ summarizing ]
<a 900-word article>
TL;DR:

[ code ]
# return the nth Fibonacci number
def fib(n):

[ conversation ]
User: what is the boiling point of water?
Assistant:
```

2. In every case the model does the same thing: predict what comes next. The task lives entirely in how the text was arranged.
3. The “TL;DR:” trick is not made up. The GPT-2 paper of February 2019 used exactly that string to get summaries out of a model that had no summarizing training at all.
4. Now in-context learning. This is the format from the GPT-3 paper of May 2020, which is where the phrase “few-shot” entered common use:

```
Translate English to French.
sea otter => loutre de mer
peppermint => menthe poivree
plush giraffe => girafe peluche
cheese =>
```

5. The three examples are not training data. They are input. They are thrown away when the request ends. The model's weights are identical before and after.
6. It works anyway, and it works better with more examples. Measured on the TriviaQA question-answering set with the 175-billion-parameter GPT-3:

Setting	Examples in prompt	Accuracy
Zero-shot	0	64.3 per cent
One-shot	1	68.0 per cent
Few-shot	64	71.2 per cent

7. Seven points of accuracy, bought with nothing but text in the prompt. No training, no weight change, no gradient.
8. Now the honest correction about instructions. Take a base model, before any fine-tuning, and give it this:

| Write a haiku about rain.

9. A base model may well continue like this, because that is what actually follows such a line on a real web page:

| Write a haiku about rain.
 | Write a limerick about a cat.
 | Write a short story about a lighthouse.
 | Submit your entries by Friday.

10. It is not being unhelpful. It is predicting correctly. It has recognized a list of writing prompts and is continuing the list.
11. Turning that into a model that answers is a separate stage of work, with its own data and its own cost, described in Chapter 49.

■ 48.15.4 PLAIN - what is really happening inside

1. One loss function drives everything. For each position, the model outputs a probability for every token in the vocabulary, and the loss is the negative logarithm of the probability it gave to the token that actually came next.
2. Averaged over the sequence, that single number is the whole training signal. There is nothing else. No task labels, no reward, no human in the loop.
3. Because the loss is per-position, one 8,192-token document gives 8,192 separate lessons in one pass. That is why the objective is so efficient.
4. **Implicit multitask learning** is the mechanism behind the range. The corpus contains question-answer pairs, parallel translations, code with comments, articles with headlines, arguments with rebuttals.
5. Each of those is a task demonstration that happens to be written down as ordinary text. The model never learns "this is translation". It learns that after a French colon, French words are likely.
6. The honest version: it is often said that models were never trained on these tasks. That is too strong. The tasks are present in the data in enormous quantity, just unlabelled. And since about 2023 laboratories deliberately mix instruction-shaped and textbook-shaped data into pretraining itself, so the clean story of pure raw web text is out of date.
7. In-context learning works differently, and we do have a partial mechanical account of it.
8. Earlier in this chapter we met **induction heads**: a pair of attention heads that between them find an earlier occurrence of the current token and copy what followed it.

9. That is a complete, if crude, pattern-completion machine built out of attention. Given “A B ... A” it predicts “B”. Given three translation pairs it can find the pattern and continue it.
10. So few-shot prompting is not the model learning. It is the model reading. The examples are in the context, and attention can reach them.
11. That is why in-context learning vanishes the moment the examples fall out of the context window. Nothing was stored. Nothing was learned in the sense of a weight changing.
12. Instruction following is the part that is genuinely bolted on. The base model has all the capability and none of the disposition to use it on request.
13. Fine-tuning on instruction-and-answer pairs changes which continuation is likely, not what the model knows. It is a change of habit, not of knowledge.
14. That distinction matters, because it explains a common observation: fine tuning for helpfulness can make a model slightly worse at raw prediction while making it far more useful.

■ 48.15.5 TECHNICAL – the engineer’s version

1. The objective, exactly. Maximize the log-likelihood of the corpus under the chain rule of probability, one factor per token:

$$\text{loss} = - (1/T) * \sum \text{over } t \text{ of } \log P(x_t | x_1 \dots x_{t-1})$$

2. This is cross-entropy between the model’s distribution and a one-hot target, in nats per token. Divide by $\ln 2$ for bits per token.
3. Shannon’s source coding theorem makes the connection to compression exact: the cross-entropy in bits per token is the size of the optimally compressed text under that model. Better prediction is literally better compression.

Method	Approximate bits per char
gzip on English prose	about 2.4
Shannon’s 1951 human study	0.6 to 1.3
Large model, held-out text	under 1.0

4. Claude Shannon’s “Prediction and Entropy of Printed English”, 1951, estimated English at roughly 0.6 to 1.3 bits per character by having humans guess next letters. Current models are inside that band on ordinary prose. Treat the figures above as approximate; they depend heavily on the text.
5. Zero-shot task transfer was demonstrated by GPT-2, “Language Models are Unsupervised Multi-task Learners”, from Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei and Ilya Sutskever, February 2019, at 1.5 billion parameters.
6. Few-shot in-context learning was the headline result of GPT-3, “Language Models are Few-Shot Learners”, from Tom Brown and colleagues, May 2020, at 175 billion parameters.
7. Instruction tuning as a named technique: FLAN, “Finetuned Language Models Are Zero-Shot Learners”, Jason Wei and colleagues at Google, September 2021; and T0, “Multitask Prompted Training Enables Zero-Shot Task Generalization”, Victor Sanh and colleagues, October 2021.
8. Then InstructGPT, “Training language models to follow instructions with human feedback”, Long Ouyang and colleagues at OpenAI, March 2022. Its key measured result: human labellers preferred the outputs of the 1.3-billion-parameter InstructGPT model over the 175-billion-parameter GPT-3, a hundredfold size difference overturned by post-training alone.
9. That single result is the strongest available evidence that instruction following is added, not emergent. Full detail is in Chapter 49.

10. Now emergence, stated fairly, because this is contested.
11. **The claim.** “Emergent Abilities of Large Language Models”, Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph and others, June 2022, published in TMLR. Their definition: an ability is emergent if it is not present in smaller models but is present in larger models, and cannot be predicted by extrapolating from smaller models.
12. Reported examples include multi-digit arithmetic, word unscrambling and some of the harder tasks in BIG-bench, the 204-task benchmark released by Aarohi Srivastava and several hundred co-authors in June 2022, which itself reported “breakthrough” behaviour on some tasks.
13. **The critique.** “Are Emergent Abilities of Large Language Models a Mirage?”, Rylan Schaeffer, Brando Miranda and Sanmi Koyejo of Stanford, April 2023. Their argument: nonlinear or discontinuous metrics produce apparent emergence, while linear or continuous metrics on the same models produce smooth, predictable curves.
14. Their strongest evidence is constructive: they induced apparent emergence in simple vision models, where nobody claims it exists, purely by choosing a harsh metric. And they showed most BIG-bench emergence disappears when exact-match accuracy is replaced by token edit distance.
15. Here is the artefact in arithmetic you can check yourself. Suppose per-token accuracy improves smoothly with scale, and the answer is 10 tokens long, and the metric is exact string match, so all 10 must be right:

Per-token accuracy	10-token exact match
0.50	0.1 per cent
0.70	2.8 per cent
0.80	10.7 per cent
0.90	34.9 per cent
0.95	59.9 per cent
0.99	90.4 per cent

16. The left column rises steadily. The right column sits near zero, then leaps. Plot the right column against compute on a log axis and you get a cliff, and nothing whatsoever happened inside the model.
17. **The reply to the critique**, given fairly. Three points. First, the critique does not show that scaling is safe or fully predictable; it shows that one class of measurement exaggerates sharpness.
18. Second, many things people actually care about really are all-or-nothing. A program compiles or it does not. A proof is valid or it is not. If the metric that matters to the user is discontinuous, the user’s experience of a sudden capability jump is real, whatever the smooth underlying curve says.
19. Third, at least one genuine discontinuity has been found mechanistically, not metrically. “In-context Learning and Induction Heads”, Catherine Olsson and colleagues at Anthropic, March 2022, identified a visible bump in the training loss curve where induction heads form, accompanied by a jump in in-context learning ability. That is a phase change in the model, not in the scoring.
20. Relatedly, “Grokking”, from Alethea Power and colleagues at OpenAI in January 2022, showed genuine sudden generalization long after memorization on small algorithmic tasks. Sharp transitions do exist.
21. A later position, from Sheng Lu and colleagues at ACL 2024, argued that many claimed emergent abilities reduce to in-context learning plus instruction tuning rather than any new latent reasoning capacity.

22. **Established fact.** Next-token prediction is the sole pretraining objective. Pretraining loss falls as a smooth power law in compute, data and parameters, per Jared Kaplan and colleagues in January 2020 and Jordan Hoffmann and colleagues in March 2022. In-context learning is real and measurable. Instruction following comes overwhelmingly from post-training.
23. **Active research.** Why in-context learning works at all. One line of work, including Johannes von Oswald and colleagues in 2022, argues the forward pass implements something equivalent to gradient descent on the in-context examples; this is suggestive and not settled. Whether emergent abilities are real, artefactual, or both depending on the ability, is open, and reasonable researchers disagree in print.
24. **Marketing claim.** That emergence shows a model has begun to understand, reason or think. That is a philosophical claim dressed as a measurement. The “Sparks of AGI” paper from Microsoft in March 2023 was widely criticized on exactly this ground, including for evaluating an unreleased model version that others could not reproduce.
25. Also marketing: presenting a capability as spontaneous when it was in the fine-tuning mix. Without disclosure of the post-training data, which is almost never published, that distinction cannot be checked from outside.
26. Observation. Per-token loss and log-probabilities are the measurable quantities here, exposed as `logprobs` by most inference APIs and by `output_scores` in Hugging Face `generate`. Task-level claims are measured with harnesses such as EleutherAI’s `lm-evaluation-harness` or Stanford’s HELM, and the first thing to check in any emergence plot is which metric the y-axis uses.

■ 48.15.6 WORDS - remember these

1. Next-token prediction — guess the next piece of text — autoregressive language modelling with a cross-entropy loss over the vocabulary.
2. Pretraining objective — the one thing the model is scored on — maximizing log-likelihood of the corpus under the chain rule of probability.
3. Cross-entropy loss — how surprised the model was by the true next token — minus the log probability assigned to the target, in nats per token.
4. Implicit multitask learning — tasks learned without being labelled as tasks — task demonstrations occurring naturally in the pretraining corpus.
5. In-context learning — learning from examples in the prompt, with no weight change — conditioning on demonstrations within a single forward pass.
6. Few-shot prompting — putting a handful of examples in the prompt — k-shot conditioning, popularized by the GPT-3 paper of May 2020.
7. Zero-shot — asking with no examples at all — task specification by instruction or format alone, demonstrated by GPT-2 in February 2019.
8. Base model — a model that has only been pretrained — a pure next-token predictor with no instruction tuning and no preference optimization.
9. Instruction tuning — teaching the model to do what it is asked — supervised fine-tuning on instruction-and-response pairs, FLAN 2021, InstructGPT 2022.
10. Emergent ability — a skill said to appear only above some size — an ability absent in smaller models and present in larger ones, per Wei et al. 2022, with the sharpness disputed by Schaeffer et al. 2023.
11. Metric artefact — a jump caused by the ruler, not the thing measured — apparent discontinuity produced by exact-match or thresholded scoring.
12. Scaling law — the predictable fall in loss as you spend more — a power law in parameters, data and compute, per Kaplan 2020 and Hoffmann 2022.

48.98 Common wrong ideas

1. Wrong: attention means the model is focusing, the way a person concentrates. Right: attention is a weighted average. Every token computes a similarity score against every other token, those scores go through a softmax, and the result is used to mix value vectors. Nothing is ignored: after a softmax every weight is strictly greater than zero, so a “0.001” token still contributes. There is no spotlight, no effort and no choosing. The name is a 1990s metaphor from Bahdanau, Cho and Bengio’s 2014 alignment work, and it has misled more readers than it has helped.
2. Wrong: the transformer understands grammar, because everything it writes is grammatical. Right: grammatical output is evidence that grammatical continuations had low prediction loss, nothing more. The model holds no parse tree, no rule for subject-verb agreement and no category called “verb”. Probing studies do find syntax-correlated directions in the activations, and that is a genuine finding, but a correlated direction is not a rule. The honest test is behaviour on rare constructions, where models still fail in ways no speaker of the language would.
3. Wrong: a bigger context window means better memory. Right: the context window is the maximum input size for one request. Nothing persists between requests, and any apparent memory is your application resending the history. Accuracy also dips for material in the middle of a long input, and NVIDIA’s RULER benchmark in 2024 found many models advertising 32,000 tokens performed well only at much shorter lengths. Longer also costs more: at 131,072 tokens the key-value cache for Llama 3.1 70B is 40 GiB, on top of the weights.
4. Wrong: it predicts words. Right: it predicts tokens, which are subword pieces from a fixed vocabulary, typically 32,000 to 256,000 entries. “Unbelievable” may be three tokens; a leading space is usually part of the token; a number may split in places no human would choose. And it does not predict one token, it produces a probability for every token in the vocabulary at once, and then a separate sampling step picks one.
5. Wrong: more heads means more intelligence. Right: heads split the same d_{model} between them. With d_{model} 512, going from 8 heads to 16 halves the head dimension from 64 to 32. Total parameters and total FLOPs are unchanged. The 2017 paper measured this directly and found 1 head was worse than 8, and 32 heads were also worse than 8. Head count is a shape choice with an interior optimum, not a quality dial.
6. Wrong: emergent abilities prove something new appeared inside the model. Right: this is disputed in print. Wei and colleagues reported sharp jumps in 2022; Schaeffer, Miranda and Koyejo at Stanford argued in 2023 that the sharpness is often produced by all-or-nothing metrics, and demonstrated the effect by manufacturing fake emergence in vision models. Some genuine phase changes do exist, notably the formation of induction heads found by Anthropic in 2022. The correct position is that the metric must be checked before any claim about the model is made.
7. Wrong: a 671-billion-parameter mixture-of-experts model is equivalent to a dense 671-billion model. Right: it does the arithmetic of a 37-billion model per token. You get the memory bill of the large model and the speed of the small one. No laboratory has shown the quality matches a dense model of the same total size, and the informal geometric-mean rule of thumb is a heuristic, not a measured law.
8. Wrong: attention is where the model’s knowledge is stored, since attention is what makes it a transformer. Right: attention holds about 17 per cent of the parameters in Llama 3.1 8B; the feed-forward blocks hold about 70 per cent, rising to 81 per cent at 405B. Attention moves information between positions. The feed-forward blocks are where the storage happens.
9. Wrong: the transformer processes text left to right, like a person reading. Right: during training every position is computed at once in one matrix multiplication, which is the entire reason the architecture won. The causal mask enforces the ordering of information without enforcing an

ordering of computation. Only generation is sequential, and that is a property of writing text one token at a time, not of the architecture.

10. Wrong: setting temperature to 0 makes the model correct. Right: it makes the model deterministic, which is a different thing. Greedy decoding picks the highest-probability token every time, which removes variation and often produces repetitive text. If the highest-probability continuation is wrong, temperature 0 guarantees you get the wrong answer every time instead of some of the time.

48.99 Chapter summary in 20 lines

1. Before 2017 sequences were handled by recurrent networks, which kept one hidden state and updated it once per word, so information from early words faded and gradients vanished or exploded across hundreds of steps.
2. The fatal flaw was not memory but order: word ten could not be computed until word nine finished, so a chip with tens of thousands of parallel units sat idle, and no amount of hardware fixed it.
3. Attention removes the order constraint. Every position looks at every other position at once, scores them for relevance, and takes a weighted average of what it finds. The whole sequence becomes one matrix multiplication.
4. Each token is projected into three roles: a query saying what it is looking for, a key advertising what it offers, and a value carrying what it passes on if selected. Three learned matrices, three different jobs.
5. The formula is $\text{softmax}(\mathbf{QK}^T / \sqrt{d_k}) \mathbf{V}$. Dot products give similarity, division by $\sqrt{d_k}$ keeps the variance of the scores near 1 so the softmax does not saturate, softmax turns scores into weights that sum to 1, and the multiply by $\mathbf{V}$ collects the answer.
6. Without the $\sqrt{d_k}$ scaling, scores at $d_k = 64$ have standard deviation around 8, the softmax becomes nearly one-hot, and gradients through it fall towards zero. The scaling is not cosmetic.
7. A causal mask sets every score above the diagonal to minus infinity before the softmax, which makes those weights exactly zero. That is how a model is stopped from reading its own answer during training.
8. Multi-head attention splits d_{model} into h independent subspaces, runs attention in each, concatenates and projects. Parameters and FLOPs are unchanged; only the shapes differ. The 2017 paper measured 8 heads as better than 1 and better than 32.
9. Heads are not clean specialists. Some interpretable ones are real, notably induction heads that complete a repeated pattern and underlie in-context learning, but most heads resist a tidy description and many can be pruned.
10. Attention is permutation equivariant: shuffle the input and the output shuffles identically. It has no idea what order anything is in, so position must be injected deliberately.
11. Position schemes went sinusoidal in 2017, learned absolute in BERT and GPT, T5 relative bias in 2019, ALiBi in 2021, and rotary embedding in 2021. RoPE rotates queries and keys by an angle set by position, so scores depend on relative distance exactly, and it is the dominant choice as of 2026.
12. RoPE's base controls the slowest wavelength. Raising it from 10,000 to 500,000, as Llama 3.1 did in July 2024, is part of how a 131,072-token context is reached without the slowest frequency wrapping around.
13. After attention, each position passes alone through a feed-forward network that widens by roughly four times, applies a non-linearity such as SwiGLU, and narrows back. This block holds 70 to 81 per cent of a model's parameters and is where facts appear to be stored.
14. Residual connections add the input back to every sub-layer's output, giving gradients a clean

path through depth and creating a residual stream that every block reads from and writes to. Normalization, LayerNorm in 2017 and RMSNorm today, keeps activation sizes stable; pre-norm replaced post-norm around 2019 and removed the need for learning-rate warmup.

15. The assembled model is: embed token ids, add or rotate in position, run N identical blocks, normalize once more, multiply by the unembedding matrix, and read a probability over the whole vocabulary at every position.
16. The 2017 paper, "Attention Is All You Need" by Ashish Vaswani and seven co-authors, reached 28.4 BLEU on WMT 2014 English-German with the big model, beating every previous system at one to two orders of magnitude less training compute. It never mentioned scaling laws or in-context learning.
17. Three architectures came out of it: encoder-only for reading, such as BERT; decoder-only for writing, which is every chat model today; and the original encoder-decoder, joined by cross-attention, which survives in translation and in T5-style text-to-text framing.
18. Generation is a loop: feed everything, get a distribution, choose one token, append, repeat. The choice is greedy, or sampled with temperature, top-k, top-p or min-p. That sampling step, not the model, is why the same prompt gives different answers.
19. Attention costs $O(n^2)$. At 131,072 tokens the score matrix for one head in one layer would be 32 GiB, so nobody stores it: FlashAttention computes the same result tile by tile without ever writing it out, grouped-query attention shrinks the key-value cache fourfold, and sliding windows, sparse patterns and state-space models attack the problem from other directions.
20. Mixture of experts splits total from active parameters, letting DeepSeek-V3 hold 671 billion numbers while using 37 billion per token; and the reason one architecture does so many jobs is that it is trained on one objective, next-token prediction, which rewards grammar, facts, style and reasoning alike, with in-context learning arriving unplanned, instruction following added afterwards by fine-tuning, and emergence still argued over.

How a Model Is Actually Made — Pretraining to Release

49.0 What this chapter gives you

1. You will be able to name every stage that turns raw web text into a shipped chat model, in order, and say what each stage costs and what it changes.
2. You will be able to explain where training data comes from, how it is cleaned, and why the legal position around it is genuinely unsettled.
3. You will be able to describe one training step exactly: forward pass, loss, backward pass, optimizer update, and say what a good loss curve looks like.
4. You will be able to use the formula C is about $6ND$ to estimate the compute of any announced model, and check it against the figures the lab published.
5. You will be able to explain scaling laws, what Chinchilla changed in 2022, and why labs deliberately ignore Chinchilla for small models today.
6. You will be able to say why a base model cannot be used as a chatbot, and what supervised finetuning and preference optimization each add.
7. You will be able to explain RLHF, reward hacking, DPO, constitutional methods and verifiable-reward training, and when each is used.
8. You will be able to open a model directory and say what every file is, what format it is in, and which formats can execute code when loaded.
9. You will be able to state precisely what “open weights”, “open source”, “open data” and “permissive licence” each mean, and tell them apart.
10. You will be able to read a model card and a licence and say what the publisher withheld, and what that omission tells you.

49.1 The pipeline in one view

■ 49.1.1 PLAIN - in simple words

1. A chat model is not made in one step. It is made in a chain of stages.
2. Each stage takes the output of the one before and changes it.
3. Stage one is collecting text. Enormous amounts of text, mostly from the web.
4. Stage two is cleaning that text. Throwing away most of it, in fact.
5. Stage three is building the **tokenizer**, the fixed list of text pieces the model is allowed to see. Chapter 47 covered this in detail.
6. Stage four is **pretraining**. This is the long, expensive one. Weeks or months on thousands of machines, learning to predict the next piece of text.

7. What comes out of stage four is called a **base model**. It is not a chatbot. It continues text. It does not answer questions.
8. Stage five is **post-training**. This is where it learns to behave like an assistant: to answer, to follow instructions, to refuse harmful requests.
9. Post-training has parts of its own: showing it good examples, then teaching it which of two answers people prefer, then safety work.
10. Stage six is **evaluation**: measuring what the thing can and cannot do.
11. Stage seven is release: publishing the numbers, or putting them behind an interface, along with documents describing what was made.
12. The costs are wildly unequal. Pretraining is almost all of the money.
13. Post-training is a rounding error in compute, and yet it is what makes the difference between a strange text continuer and something you can talk to.
14. That imbalance is the single most useful fact in this chapter.

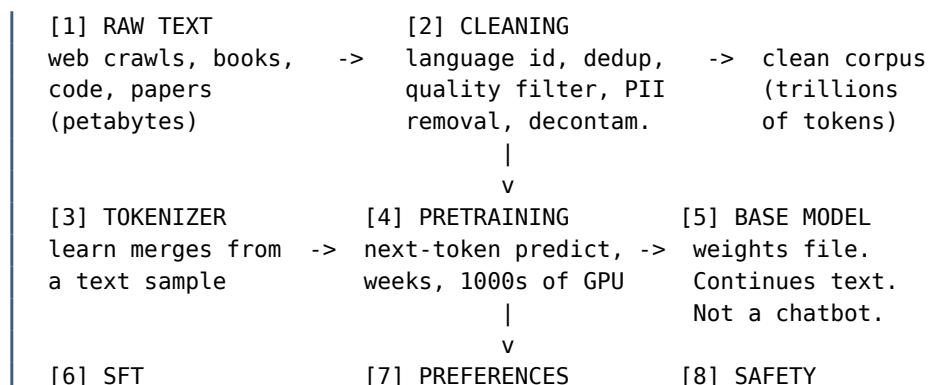
■ 49.1.2 PLAIN – a picture in your head

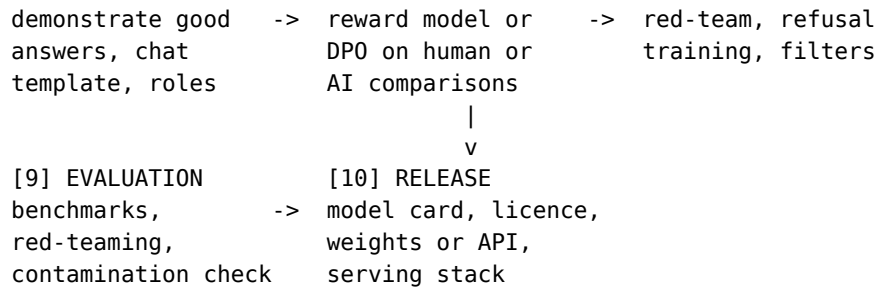
1. Think of how a doctor is made.
2. First, years of general education. School, then a science degree. Broad, slow, expensive, and not aimed at any one job.
3. That is pretraining. Enormous input, no particular task, no bedside manner.
4. A person at the end of that stage knows a great deal of biology and can still be useless in a consulting room.
5. Then comes clinical training. Watching a senior doctor, copying how they greet a patient, how they explain bad news, when they say “I do not know”.
6. That is supervised fine-tuning: learning the form of the job by imitation.
7. Then comes years of feedback. A supervisor says “that explanation was better than the one you gave yesterday”. Slowly the style settles.
8. That is preference optimization: learning from comparisons, not from corrections.
9. Then exams and a licence before anyone is allowed to practise unsupervised.
10. That is evaluation and release.

Where this comparison breaks: a medical student keeps learning after graduation. A shipped model does not. Its numbers are frozen at release and never change again unless the lab trains a new version and publishes a new file. Also, a doctor’s general education takes years and their clinical training takes years. In a model, the general education is 99 per cent of the cost and the clinical training is about 1 per cent. The proportions are nothing like a human career.

■ 49.1.3 PLAIN – a worked example

1. Here is the whole chain as a diagram. Follow the arrows left to right.





- Now the same chain as a table, with rough costs. These are typical orders of magnitude for a large 2024 to 2026 model, not exact figures for any one lab.

Stage	Share of GPU compute	Typical duration
Data collection, cleaning	Under 1% (CPU-heavy)	Weeks to months
Tokenizer training	Under 0.001%	Hours
Pretraining	95 to 99%	Weeks to months
Supervised fine-tuning	0.1 to 1%	Hours to days
Preference optimization	0.1 to 2%	Days
Evaluation, red-teaming	Under 1%	Weeks (human time)

- Note what that table says. Data cleaning is cheap in GPU terms and enormous in CPU and human terms. It runs on ordinary processors, not accelerators.
- Note also that evaluation and red-teaming take weeks of calendar time while using almost no compute, because the bottleneck is people.

■ 49.1.4 PLAIN – what is really happening inside

- What changes at each stage is worth being precise about.
- Data collection changes nothing in the model. There is no model yet.
- Tokenizer training produces a small file, a few megabytes, that fixes the vocabulary for the life of the model. Change it later and everything breaks.
- Pretraining sets every weight in the model, starting from random numbers. This is where essentially all of the model's knowledge is installed.
- Supervised fine-tuning changes every weight again, but only very slightly, using a dataset perhaps a millionth the size.
- Preference optimization changes every weight again, again slightly.
- Safety work is not a separate mechanism. It is more of the same fine-tuning, using data chosen to produce refusals and careful answers.
- Evaluation changes nothing. It only measures.
- So there is exactly one moment where the model gains its knowledge, and it is pretraining. Everything after that shapes how the knowledge comes out.
- That sentence needs a caution, which section 49.8 gives properly: the boundary is real but blurrier than it sounds.

■ 49.1.5 TECHNICAL – the engineer's version

- The canonical pipeline, with the standard names used in papers: corpus construction, tokenizer fitting, pretraining, mid-training or annealing, supervised fine-tuning, preference optimization, evaluation.

2. **Mid-training** is the newest of these names and dates from roughly 2024. It is a final pretraining phase on a much higher-quality mixture, with a decaying learning rate. Ai2’s Dolmino mixes and OLMo 2, published in January 2025, made the term common.
3. Compute is measured in floating-point operations, FLOPs, and in accelerator-hours. Both are reported. Neither alone tells you the cost.
4. Real reported compute for the pretraining stage of Llama 3.1, from Meta’s published model card of July 2024:

Model	GPU hours (H100-80GB)	Reported CO2eq
Llama 3.1 8B	1.46 million	420 tonnes
Llama 3.1 70B	7.0 million	2,040 tonnes
Llama 3.1 405B	30.84 million	8,930 tonnes

5. Those CO2 figures are location-based. Meta reported market-based emissions as zero, because it matches its electricity use with renewable purchases. Treat “zero” as an accounting statement, not a physical one.
6. Post-training compute is rarely published separately. Where it has been, it is well under 1 per cent of pretraining. InstructGPT, published by OpenAI in March 2022, reported its whole RLHF stage at about 60 petaflop-days against 3,640 petaflop-days for GPT-3 pretraining, roughly 1.6 per cent.
7. Established fact: pretraining dominates training cost. Active research: how much of final capability comes from post-training, especially for reasoning models where reinforcement learning budgets have grown sharply since late 2024. Marketing claim: any statement that a model was “built from scratch” when the lab in fact fine-tuned somebody else’s open-weight base model.

■ 49.1.6 WORDS – remember these

1. Pretraining — the long stage where the model learns from raw text — training a randomly initialized network on a large corpus with a next-token objective.
2. Post-training — everything done after pretraining to make it usable — supervised fine-tuning, preference optimization and safety tuning.
3. Base model — the raw output of pretraining — a text continuer with no chat template, no role notion and no refusal behaviour.
4. Corpus — the pile of text used for training — the deduplicated, filtered token stream fed to the pretraining loop.
5. Mid-training — a high-quality final stretch of pretraining — an annealing phase on a curated mixture with a decaying learning rate.
6. FLOP — one arithmetic operation on numbers with a decimal point — one floating-point operation, the standard unit of training compute.

49.2 The data: where it comes from and how it is cleaned

■ 49.2.1 PLAIN – in simple words

1. The model needs text. An almost unimaginable amount of text.
2. The biggest source is the open web, scraped by crawlers that follow links and save pages.
3. The best-known scrape is **Common Crawl**, run by a non-profit of the same name, which has been crawling since 2008 and gives its archives away free.
4. Common Crawl alone is not enough and is not good enough. Most of it is navigation menus, adverts, spam and duplicated boilerplate.

5. So labs add other sources: curated web pages, books, code from public repositories, scientific papers, forum posts, subtitles, reference works.
6. Some labs pay for data. News archives, stock photo libraries, forum archives and publisher catalogues have all been licensed for money.
7. And increasingly, labs make text with other models. That is called **synthetic data**.
8. Then comes the part nobody outside the field expects: most of it is thrown away.
9. A realistic pipeline keeps somewhere between 5 and 20 per cent of what it started with, and sometimes far less.
10. Removing near-duplicate pages alone can cut a web crawl in half.
11. What survives is a stream of trillions of tokens, where a token is roughly three-quarters of an English word.
12. The last twenty years of computing said “more data is better”. This field learned, roughly between 2022 and 2024, that better data is better.

■ 49.2.2 PLAIN – a picture in your head

1. Imagine you want to teach someone English by handing them paper.
2. You send a truck round a city and collect every piece of printed paper it can find. Newspapers, junk mail, receipts, packaging, books, love letters.
3. You get a warehouse full. Most of it is worthless for your purpose.
4. So you sort it. First you throw out everything not in English.
5. Then you notice that you have forty thousand copies of the same supermarket leaflet, so you keep one and burn the rest.
6. Then you throw out anything that is mostly numbers, menus or telephone listings, because reading it teaches nothing about language.
7. Then you black out every name, address and card number you find.
8. Then, before the exam, you check that the exam paper itself did not end up in the pile, because that would make the results meaningless.
9. What is left is a much smaller, much better warehouse.

Where this comparison breaks: paper sorting is done by people who understand what they are reading. Data cleaning is done by programs that mostly cannot. A quality filter is itself a small trained model making a guess, and it makes systematic mistakes: it tends to rate text that looks like a textbook highly and text written in dialect or by non-native speakers poorly. That bias goes straight into the finished model and nobody notices until much later.

■ 49.2.3 PLAIN – a worked example

1. Here is a real published data mixture. It is from the GPT-3 paper by Brown and colleagues in 2020, and it is one of the last fully published mixtures from a frontier lab.

Source	Tokens available	Share of training
Common Crawl, filtered	410 billion	60%
WebText2	19 billion	22%
Books1	12 billion	8%
Books2	55 billion	8%
Wikipedia	3 billion	3%

2. Read the two columns against each other. Wikipedia is 3 billion tokens out of 499 billion available, which is 0.6 per cent of the pile.

3. But it was given 3 per cent of the training weight. So during training, the model saw Wikipedia about 3.4 times over while seeing Common Crawl less than half way through.
4. That is what a **mixing ratio** does. It is a deliberate decision to over-sample good text and under-sample mediocre text.
5. Now a real published mixture from four years later. Meta’s Llama 3 paper of July 2024 gave the final proportions of its 15 trillion token mix:

Category	Share of the mix
General knowledge	50%
Maths and reasoning	25%
Code	17%
Multilingual	8%

6. Note that a quarter of the diet is maths and reasoning and a sixth is code, for a model most people use to write English prose.
7. That is not an accident. Code and maths are believed to improve general reasoning, and they are the two areas where correctness is checkable.
8. Note also what Meta did not publish: which datasets, which websites, which books, in what proportion, under what licence.

■ 49.2.4 PLAIN – what is really happening inside

1. Here is the cleaning pipeline in the order it actually runs.
2. **Extract.** A crawl file holds raw HTML. A tool pulls out the article text and drops the menus, headers, adverts and cookie banners. The library most commonly named for this in open pipelines is Trafilatura.
3. **URL filtering.** Drop whole domains from block lists: adult sites, known spam farms, malware hosts. Cheap and effective, done before anything else.
4. **Language identification.** A small classifier guesses the language of each document and scores its confidence. Keep only what you want, above a threshold. This is where most of a global crawl disappears.
5. **Exact deduplication.** Hash each document. Identical hash, identical document, keep one.
6. **Fuzzy deduplication.** Much harder and much more important. Two pages can differ by a date stamp and be otherwise identical. Section 49.2.5 gives the method, called MinHash.
7. **Quality filtering.** Two kinds. Heuristics: rules like “drop documents with fewer than 50 words”, “drop documents where more than 30 per cent of lines are duplicates”, “drop documents with no full stops”. And classifiers: a small model trained to score whether text looks like the good stuff.
8. **Toxicity and safety filtering.** Classifiers remove the most extreme material. This is a judgement call with real costs both ways, and labs differ on where the line goes.
9. **PII removal.** Personally identifiable information. Email addresses, phone numbers, public IP addresses are detected by pattern and masked.
10. **Decontamination.** Search the corpus for text that appears in the evaluation benchmarks and remove it. Otherwise the model has seen the exam.
11. **Mixing.** Assign weights to each remaining source and sample accordingly.
12. Every one of those steps discards data. The compounding is severe. Ten steps each keeping 80 per cent leave you with 11 per cent.

■ 49.2.5 TECHNICAL – the engineer’s version

1. Scale of the raw material. Common Crawl’s archive CC-MAIN-2025-30, crawled between 7 and 21 July 2025, holds 2.42 billion web pages, 419 TiB of uncompressed content, 114.15 TiB compressed. There are several such crawls per year and the organization has been publishing since 2008.
2. Formats: WARC holds the raw request and response, WAT holds metadata, WET holds extracted plain text. Serious pipelines re-extract from WARC rather than trusting WET.
3. Named public corpora, with real figures:

Corpus	Size	Published
The Pile (EleutherAI)	825 GiB, 22 sources	Dec 2020
C4 (from the T5 paper)	about 750 GB English	2019
Dolma (Ai2)	3 trillion tokens	2024
FineWeb (Hugging Face)	15T at v1.0, now 18.5T	Apr 2024
Dolma 3 (Ai2)	9.3 trillion tokens	Nov 2025

4. FineWeb’s published pipeline is the best documented open example. In order: URL filtering, Trafilatura extraction, a fastText language filter keeping English scores above 0.65, the Gopher repetition and quality filters, the C4 quality filters, custom heuristics, MinHash deduplication per crawl, and PII anonymization of emails and public IP addresses. Licence: ODC-By 1.0.
5. **MinHash deduplication**, properly. Cut each document into overlapping n-grams, called shingles. FineWeb uses 5-grams. Hash every shingle. For each of many hash permutations, keep only the minimum hash value. The resulting vector of minimums is the document’s signature.
6. The mathematical property that makes this work: the probability that two documents share a given minimum equals their Jaccard similarity, which is the size of the intersection of their shingle sets divided by the size of the union.
7. Signatures are then split into bands. FineWeb uses 14 bands of 8 hashes, so 112 hashes per document. Two documents are candidate duplicates if any one band matches exactly. This turns a quadratic all-pairs comparison into a hash-table lookup.
8. The detection curve for 14 bands of 8, computed from 1 minus (1 minus s to the eighth) to the fourteenth:

Jaccard similarity	Chance of being flagged
0.50	about 5%
0.75	about 77%
0.90	about 99.96%

9. So this configuration is a deliberate threshold at roughly 0.75. Change the bands and rows and you move the knee of that curve.
10. **Decontamination** is done by n-gram overlap against benchmark sets, often 13-gram or 8-gram matching. Ai2 published a dedicated tool called decon with OLMo 3 in November 2025. GPT-4’s technical report in March 2023 described substring-match contamination checks and reported that its results moved little when contaminated examples were removed.
11. Established fact: deduplication improves models. Lee and colleagues in 2021 showed deduplicated training data reduces memorized regurgitation by an order of magnitude and does not hurt, and often helps, perplexity.

12. Established fact: quality filtering beats raw volume at fixed compute. FineWeb-Edu, filtered by an educational-quality classifier, beat much larger unfiltered sets on knowledge benchmarks at the same token budget.
13. Now the honest part, on copyright. As of mid-2026 the legal position on training on copyrighted text is unsettled and actively litigated. Named cases you should know:

Case	Filed	Position
NYT v. OpenAI, Microsoft	Dec 2023	Live, dismissal denied
Kadrey v. Meta	Jul 2023	Live, mixed rulings
Bartz v. Anthropic	Aug 2024	Settled, 1.5bn USD
Thomson Reuters v. Ross	2020	Fair use rejected 2025

14. In Bartz v. Anthropic, Judge William Alsup ruled in June 2025 that training on lawfully bought and scanned books was fair use, but that keeping a library of pirated copies was not. Anthropic then agreed a settlement reported at 1.5 billion US dollars, roughly 3,000 dollars per work across about half a million works. Reported as the largest copyright settlement in United States history.
15. Read those two halves carefully, because they say different things. The training was found transformative. The acquisition was the problem.
16. In Thomson Reuters v. Ross Intelligence, Judge Stephanos Bibas held in February 2025 that the use was not fair. That case concerned a search tool rather than a generative model, so its reach is disputed.
17. The honest version: nothing here is settled. Different courts in different countries have reached different conclusions, appeals are pending, and the European Union, the United Kingdom and Japan have taken materially different statutory approaches to text and data mining.
18. Second honest point: **most labs no longer disclose their data mixture**. GPT-3 in 2020 published a table. GPT-4 in 2023 explicitly declined, citing competition and safety. Llama 3 in 2024 published category percentages but no source list. Gemini and Claude publish neither.
19. This is a genuine change in scientific norms and you should treat any claim about a closed model's training data, including claims about what it was not trained on, as unverifiable.
20. Third honest point: **data quality now matters more than quantity**. The reason is that the supply of high-quality human text is finite. Villalobos and colleagues estimated the stock of public human text at roughly 300 trillion tokens and projected full use somewhere between 2026 and 2032. That projection is contested, but the direction is not.
21. Tools you will actually use: datatrove and dolma for pipelines, fastText for language identification, datasketch or a custom Spark job for MinHash, and the Common Crawl index for selecting URLs without downloading the lot.

■ 49.2.6 WORDS – remember these

1. Common Crawl — a free archive of crawled web pages — a non-profit web corpus published since 2008 in WARC, WAT and WET formats.
2. Deduplication — removing repeated text — exact hashing plus fuzzy matching, usually MinHash with locality-sensitive hashing over shingles.
3. MinHash — a trick for spotting near-identical documents cheaply — a signature of minimum hash values whose collision rate equals Jaccard similarity.
4. Decontamination — checking the exam is not in the textbook — n-gram overlap removal of benchmark text from the pretraining corpus.

5. Mixing ratio — how much of each source the model sees — per-source sampling weights, expressed as epochs or proportions of the token budget.
6. Synthetic data — text written by another model — model-generated corpora used for pretraining, mid-training or instruction tuning.
7. PII — personal details about real people — personally identifiable information, detected by pattern and masked before training.

49.3 Pretraining itself

■ 49.3.1 PLAIN - in simple words

1. Pretraining has exactly one goal, and it is smaller than people expect.
2. Given some text, predict the next piece of text.
3. That is all. There is no second objective. Nobody labels anything. Nobody tells the model what is true.
4. The text itself is the answer key, because the next token is right there in the data.
5. This is why it is called **self-supervised**: the supervision comes free from the data, not from a human.
6. So the model reads “The capital of France is” and tries to guess the next token. It compares its guess to the real one, which is “Paris”.
7. If it guessed badly, every one of its billions of numbers gets nudged a tiny amount in the direction that would have made the guess better.
8. Then it does that again. And again. Trillions of times.
9. Nothing else happens. No rules are written down. No facts are stored as facts. Just this one loop, repeated at enormous scale.
10. And out of that loop comes something that can write working code and explain quantum mechanics. That is the surprising part, and it remains genuinely surprising to the people who build these things.

■ 49.3.2 PLAIN - a picture in your head

1. Imagine a person locked in a room with a printing press feeding them paper.
2. The paper always arrives with the last word covered by a sticker.
3. Their only job is to guess the covered word, then peel the sticker and see.
4. They are not told the meaning of anything. They get no explanations. They are never asked a question. They just guess and check, forever.
5. At first they guess randomly. After a while they learn that “the” is common.
6. Later they learn that after “the capital of France is” the covered word is almost always “Paris”. They have not been taught geography. They have been taught which word comes next.
7. After enough of this, to get the guesses right, they are forced to build something inside their head that behaves a great deal like understanding.

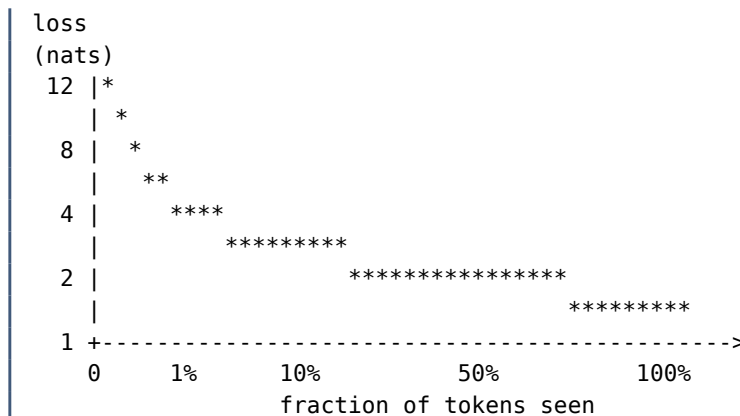
Where this comparison breaks: a person would get bored, would sleep, and would form intentions. The training loop has none of that. Also, a person reads in order and remembers yesterday. The model sees documents in shuffled order, in fixed-length chunks, and carries nothing between chunks except the weights it has changed. There is no continuous experience of any kind.

■ 49.3.3 PLAIN – a worked example

1. One training step, all the way through, with small round numbers.
2. Take a batch of 2,048 sequences, each 8,192 tokens long. That is 16,777,216 tokens in one step. Roughly 16.8 million.
3. **Forward pass.** Push all those tokens through the network. At every position the model produces a probability for every token in the vocabulary.
4. For a 128,256-token vocabulary that is 128,256 probabilities per position, which for this batch is about 2.15 trillion numbers produced and consumed.
5. **Loss.** At each position, look up the probability the model gave to the token that actually came next. Take the negative logarithm of it. Average over all 16.8 million positions.
6. That is **cross-entropy loss**, measured in nats per token.
7. Some real arithmetic on that. If the model gave the correct token a probability of 0.5, its contribution is 0.693. If 0.1, then 2.303. If 0.9, then 0.105. Confident and right is cheap. Confident and wrong is ruinous.
8. A model that guessed uniformly at random over 128,256 tokens would score the natural logarithm of 128,256, which is 11.76. That is the starting point.
9. **Backward pass.** Work out, for every one of the billions of parameters, which direction would have reduced that loss. This is backpropagation, and it costs roughly twice what the forward pass cost.
10. **Optimizer update.** Nudge every parameter a little in that direction. The size of the nudge is the learning rate, typically a few times ten to the minus five at the start of training.
11. Now discard the batch. It is never seen again. Load the next 16.8 million tokens and repeat.
12. To consume 15 trillion tokens at 16.8 million per step takes about 894,000 steps. Real runs are close to a million steps.

■ 49.3.4 PLAIN – what is really happening inside

1. The loss curve is the one picture that tells you whether a run is healthy.



2. What that shape means, in order.
3. The first drop, from about 11.8 to about 6, happens almost immediately. The model is learning which tokens are common. That is a few hundred steps.
4. The second phase, down to about 3, is learning basic word order and grammar. Still fast.
5. Then the long tail. From 3 down towards 1.5 takes 99 per cent of the run, and this is where everything interesting is learned.
6. A healthy curve is smooth, monotone downwards, and slightly noisy. Published final training losses for large models typically land somewhere between 1.4 and 2.0 nats per token, depending heavily on tokenizer and data mix.

7. An unhealthy curve has **spikes**: the loss jumps up sharply, sometimes by several nats, and either recovers over thousands of steps or never does.
8. A run that never recovers is said to have **diverged**. The loss goes to a large number or to not-a-number and the run is dead.
9. Causes: learning rate too high, a bad batch of data, numerical overflow in low precision, or a hardware fault corrupting a gradient.
10. The standard fix is brutal and simple: stop, load the last checkpoint, skip the batches around the spike, restart.
11. **Checkpointing** is what makes that possible. Every so often the whole training state is written to storage: weights, optimizer state, data position, random number generator state.
12. Note “optimizer state”. Adam keeps two extra numbers per parameter, so a training checkpoint is roughly three to four times the size of the final published model file.

■ 49.3.5 TECHNICAL – the engineer’s version

1. Objective: minimize the negative log-likelihood of the next token under a causal, decoder-only transformer, averaged over positions. Formally the loss is minus the mean over t of $\log p$ of $x_{\text{sub } t}$ given $x_{\text{sub } < t}$.
2. This is exactly maximum likelihood estimation. There is no auxiliary loss in a standard dense run beyond, in mixture-of-experts models, a load-balancing term to stop all tokens routing to one expert.
3. Optimizer: AdamW, from Loshchilov and Hutter in 2017, is near universal. Typical betas 0.9 and 0.95, epsilon $1e-8$, weight decay 0.1, gradient norm clipped at 1.0.
4. Real reported hyperparameters, Llama 3 405B, from Meta’s July 2024 paper: peak learning rate $8e-5$, linear warmup over 8,000 steps, cosine decay to $8e-7$ over 1,200,000 steps.
5. Batch size was ramped, not fixed: starting at 4 million tokens per batch with 4,096-token sequences, moving to 8 million tokens with 8,192-token sequences after 252 million tokens, and to 16 million tokens after 2.87 trillion tokens. Ramping the batch improves early-training stability.
6. Why warmup exists: at initialization the gradient estimates are poor and a full-size step can throw the weights somewhere unrecoverable. Warmup ramps the learning rate from near zero over the first few thousand steps.
7. Why decay exists: large steps explore, small steps settle. Cosine decay to about 1 per cent of peak is the long-standing default.
8. Changing as of 2024 to 2026: warmup-stable-decay schedules, which hold the learning rate constant for most of the run and decay sharply at the end. These allow a run to be extended without having chosen the end point in advance, which cosine does not. DeepSeek and several Chinese labs adopted multi-step or WSD schedules; the practice is now common but not universal.
9. A full set of real reported figures for one model, Llama 3.1 405B:

Quantity	Reported value
Parameters	405 billion
Training tokens	about 15.6 trillion
Accelerators	over 16,384 H100–80GB
GPU hours, pretraining	30.84 million
Context length, final	128,000 tokens

10. Derived figures from those, computed here rather than quoted: 30.84 million GPU-hours divided by 16,384 GPUs is 1,882 hours, which is 78.4 days of wall clock if every GPU ran the entire time.

11. Energy, GPU-only: 30.84 million hours times 700 watts is 21.6 GWh. Add host CPUs, memory, networking and cooling and 30 to 40 GWh is the plausible total, which is consistent with the 8,930 tonnes CO₂eq Meta reported at typical grid carbon intensity.
12. Cost, estimated not quoted: at public H100 rental rates of roughly 2 to 4 US dollars per GPU-hour in 2024 and 2025, 30.84 million GPU-hours is between about 62 and 123 million dollars for the pretraining compute alone.
13. That figure excludes salaries, failed runs, ablations, data licensing and the capital cost of the cluster. Reported “training cost” numbers almost always exclude those, and you should assume so unless told otherwise.
14. The contrasting example, and the reason to read cost claims carefully: DeepSeek-V3, published December 2024, is 671 billion total parameters with 37 billion active per token, trained on 14.8 trillion tokens in 2.788 million H800 GPU-hours. The report itself puts that at 5.576 million US dollars assuming 2 dollars per GPU-hour, and states plainly that the figure excludes prior research, ablations and data.
15. Failures are routine, not exceptional. Meta reported 466 job interruptions during a 54-day snapshot of the 405B run, of which 419 were unexpected, and around 78 per cent of those were confirmed or suspected hardware issues. Effective training time was still above 90 per cent.
16. Historical precedent worth knowing: Meta published the OPT-175B logbook in 2022, a day-by-day record of crashes, loss divergences and restarts. It remains the most honest public document about what a large run feels like.
17. Google’s PaLM paper in 2022 reported roughly 20 loss spikes and described the mitigation: restart from a checkpoint about 100 steps before the spike and skip 200 to 500 data batches.
18. Tools that observe a run: TensorBoard or Weights and Biases for curves, `nvidia-smi` and DCGM for device health, `torch.distributed` logs and NCCL debug output for collective failures, and the model’s own gradient-norm and learning-rate traces, which are usually logged every step.

■ 49.3.6 WORDS – remember these

1. Next-token prediction — guess the next piece of text — an autoregressive causal language modelling objective over a token vocabulary.
2. Cross-entropy loss — a score for how wrong a guess was — the negative log probability assigned to the observed token, averaged over positions.
3. Nat — the unit that loss is measured in — one unit of information using natural logarithms; multiply by 1.4427 to get bits.
4. Step — one round of guess, score, adjust — one forward pass, backward pass and optimizer update over one global batch.
5. Warmup — starting gently — ramping the learning rate from near zero over the first thousands of steps to avoid early instability.
6. Checkpoint — a saved snapshot to restart from — weights, optimizer moments, data-loader position and RNG state written to durable storage.
7. Divergence — a training run going permanently wrong — loss increasing without recovery, often to infinity or not-a-number.

49.4 Scaling laws: how big, how much data

■ 49.4.1 PLAIN – in simple words

1. Suppose you have a fixed budget of computing time. How should you spend it?
2. You have two dials. Make the model bigger, or feed it more text. You cannot max both, because both cost compute.
3. A **scaling law** is a measured answer to that question. Not a theory. A curve fitted to hundreds of small experiments and extended to large ones.
4. The first big one came from OpenAI in January 2020. Kaplan and colleagues measured how the loss falls as you increase model size, data and compute.
5. Their headline finding: the loss follows a smooth power law over more than seven orders of magnitude. It is astonishingly predictable.
6. Their practical advice: spend most of your extra budget on making the model bigger, and relatively little on more data. Stop training early.
7. The whole industry followed that advice. This is why 2020 to 2022 produced a race of enormous models trained on comparatively little text.
8. Then in March 2022, a team at DeepMind led by Hoffmann redid the experiment more carefully and found the advice was wrong.
9. Their model, Chinchilla, had 70 billion parameters and was trained on 1.4 trillion tokens. It beat Gopher, a 280-billion-parameter model trained on 300 billion tokens, using the same compute budget.
10. Four times smaller and better, because it saw four times more text.
11. Their rule of thumb: for each parameter, train on about 20 tokens.
12. That rule reshaped the field within about a year.
13. And then, from roughly 2023 onwards, labs started ignoring it on purpose.
14. The reason is money in a different place. Chinchilla asks how to get the best model for a fixed training budget. It says nothing about what the model costs to run afterwards.
15. If a model will answer a trillion questions, a smaller model that took longer to train is much cheaper over its life. So labs overtrain small models far past 20 tokens per parameter, on purpose.

■ 49.4.2 PLAIN – a picture in your head

1. Think about choosing an engine and a fuel tank for a delivery van.
2. A bigger engine is like more parameters. It can do more per unit of time.
3. More fuel is like more training data. It lets you go further.
4. You have a fixed amount of money. Spend it on engine or on fuel.
5. Kaplan's 2020 answer was: buy the biggest engine you can afford and put in just enough fuel to leave the yard.
6. Chinchilla's 2022 answer was: that is silly, you have bought an engine you cannot use. Buy a smaller engine and fill the tank properly.
7. And the 2024 answer is: wait, this van will do school runs every day for five years. The fuel to build it is nothing next to the fuel to run it. Buy the smallest engine that does the job, and train it as long as you like.

Where this comparison breaks: an engine's capability is fixed by its size and a model's is not. Two models with the same parameter count can differ hugely depending on data quality, architecture and post-training. The van analogy also suggests a hard ceiling, and there is no known hard ceiling here; the curves have simply not bent yet within the range anyone has measured.

■ 49.4.3 PLAIN – a worked example

1. The formula every practitioner uses for training compute is short: C is approximately 6 times N times D .
2. C is total floating-point operations. N is parameter count. D is number of training tokens.
3. Where the 6 comes from: each parameter does one multiply and one add per token in the forward pass, which is 2 operations. The backward pass costs about twice the forward pass, which is 4 more. Two plus four is six.
4. Work it for GPT-3, published 2020: 175 billion parameters, 300 billion tokens. 6 times $1.75e11$ times $3.0e11$ equals $3.15e23$ FLOPs. The GPT-3 paper reported $3.14e23$. The formula lands on the nose.
5. Work it for Llama 3.1 405B: 405 billion parameters, 15.6 trillion tokens. 6 times $4.05e11$ times $1.56e13$ equals $3.79e25$ FLOPs. Meta reported $3.8e25$. Again correct.
6. Now use it in the other direction. Suppose a lab announces a model with 70 billion parameters trained on 15 trillion tokens. 6 times $7e10$ times $1.5e13$ equals $6.3e24$ FLOPs.
7. An H100 does roughly 989 teraflops of dense bf16 arithmetic at peak, which is $9.89e14$ per second. Real utilization is 35 to 45 per cent, so call it $4e14$ achieved.
8. $6.3e24$ divided by $4e14$ is $1.575e10$ GPU-seconds, which is 4.375 million GPU-hours. On 4,096 GPUs that is 1,068 hours, or about 45 days.
9. You can now sanity-check any announcement in about a minute. If the numbers do not roughly agree, one of the published figures is wrong or incomplete.
10. Tokens-per-parameter for real models, which shows how far practice moved:

Model	Tokens per parameter	Versus Chinchilla
GPT-3 (2020)	1.7	12x undertrained
Chinchilla (2022)	20	the reference
Llama 3.1 405B (2024)	38.5	about 2x over
Llama 3.1 8B (2024)	about 1,875	about 94x over

11. Read the last row again. An 8-billion-parameter model was trained on 15 trillion tokens. That is roughly 94 times more text than the compute-optimal rule says is worth it.
12. Meta did that knowing it was compute-inefficient, because the 8B model is the one that gets deployed everywhere and its serving cost matters more.

■ 49.4.4 PLAIN – what is really happening inside

1. Why does a scaling law exist at all? Nobody fully knows. It is an empirical regularity, not a derived result.
2. What it says is that loss L falls as a power of the thing you scale. In words: doubling the model gives a fixed proportional reduction in loss, and that same proportional reduction keeps arriving for each further doubling.
3. Power laws are brutal in one direction and generous in the other. Each further step down in loss costs many times more compute than the last.
4. The Chinchilla correction was not a new idea, it was a better experiment. The 2020 work had used a fixed learning-rate schedule length for all runs, which quietly penalized the runs that used more data.
5. When Hoffmann's team varied the schedule to match each token budget, the optimum moved decisively towards more data.
6. The number 20 is not sacred. It is roughly where the fitted curve puts the optimum for that family of models and that data. Different data gives a different constant.

7. And the whole framework optimizes the wrong thing for a deployed product. It minimizes loss for a fixed training budget. It ignores inference.
8. Add inference to the objective and the optimum moves towards smaller models trained longer, exactly as practice has done.

■ 49.4.5 TECHNICAL – the engineer’s version

1. Kaplan and colleagues, “Scaling Laws for Neural Language Models”, 23 January 2020. Loss scales as a power law in model size, dataset size and compute, with trends spanning over seven orders of magnitude. Their compute-optimal allocation grows N much faster than D , with D scaling roughly as C to the 0.27 and N as C to the 0.73.
2. Hoffmann and colleagues, “Training Compute-Optimal Large Language Models”, 29 March 2022, known as Chinchilla. Over 400 models from 70 million to 16 billion parameters on 5 to 500 billion tokens. Conclusion: N and D should scale roughly equally, so for every doubling of model size you should double the tokens. The implied ratio is about 20 tokens per parameter.
3. Chinchilla itself: 70 billion parameters, 1.4 trillion tokens. It beat Gopher at 280 billion parameters on 300 billion tokens, and also beat GPT-3 at 175 billion and Megatron-Turing NLG at 530 billion, at equal compute.
4. What changed in practice: the 2023 Llama 1 release from Meta was explicitly framed around this, training 7B and 13B models on 1 trillion tokens, well past compute-optimal, arguing that inference cost is what matters.
5. Contested, and you should know this: a 2024 replication attempt by Besiroglu and colleagues re-fitted the third of Chinchilla’s three estimation methods and found the reported parameters inconsistent with the paper’s own data, with wider confidence intervals than stated. The headline conclusion survived. The precise constant did not.
6. The inference-aware reformulation was written down by Sardana and colleagues in 2023, “Beyond Chinchilla-Optimal: Accounting for Inference in Language Model Scaling Laws”. Adding expected lifetime inference tokens to the objective shifts the optimum to smaller, longer-trained models.
7. A cost model you can reason with: training costs $6ND$ once. Inference costs about $2N$ per generated token, forever. If a model will generate T tokens over its life, total compute is about $6ND$ plus $2NT$.
8. Put numbers in it. An 8-billion model trained on 15 trillion tokens costs $7.2e23$ FLOPs to train. Generating one token costs $1.6e10$. So the training cost equals about 45 trillion generated tokens. A busy service passes that in a couple of months.
9. Regulatory tie-in, which makes the formula matter legally. The European Union’s AI Act, in force since 1 August 2024 with general-purpose model obligations applying from 2 August 2025, uses a training-compute threshold of 10 to the 25 FLOPs as a presumption of systemic risk. Llama 3.1 405B at $3.8e25$ is above it. Llama 3.1 8B at $7.2e23$ is below it.
10. Separating the three kinds of claim, as of mid-2026: Established fact: loss follows smooth power laws in N , D and C over the measured range; Chinchilla’s directional correction to Kaplan; the $6ND$ approximation; the dominance of inference cost for widely served models. Active research: whether scaling continues past the current frontier, how to model data quality inside a scaling law, scaling laws for mixture-of-experts and for test-time compute, and whether the supply of human text is a real ceiling. Marketing claim: any specific prediction of what a given future model will be able to do, derived from a loss curve. Loss is predictable. Which capabilities appear at which loss is not.
11. One more honest note. Since roughly September 2024 a second scaling axis has been in play:

spending more compute at answer time rather than at training time. OpenAI's o1 and DeepSeek-R1, released January 2025, both showed accuracy improving with more reasoning tokens per question. That changes the economics again, because it moves cost back to inference.

■ 49.4.6 WORDS – remember these

1. Scaling law — a measured rule for how much better a bigger model gets — an empirical power-law fit of loss against parameters, tokens or compute.
2. Chinchilla-optimal — the balanced split of budget between size and data — roughly 20 training tokens per parameter, from Hoffmann and colleagues 2022.
3. Overtraining — deliberately feeding a small model far more text than is compute-optimal — trading training efficiency for lower inference cost.
4. 6ND — the arithmetic shortcut for training cost — approximately 6 times parameters times tokens, in floating-point operations.
5. Compute-optimal — best model for a fixed training budget — the point on the isoFLOP curve where loss is minimized, ignoring deployment cost.
6. Test-time compute — thinking longer at answer time — allocating additional inference tokens or samples to raise accuracy on a single query.

49.5 What a base model actually is

■ 49.5.1 PLAIN – in simple words

1. Pretraining finishes. You have a file of numbers. That is the base model.
2. It is very good at exactly one thing: continuing text.
3. It is not good at being talked to, because nobody taught it that.
4. Give it “The capital of Australia is” and it will finish the sentence correctly, because that is a continuation.
5. Give it “What is the capital of Australia?” and something strange happens.
6. On the internet, a line that looks like a quiz question is usually followed by more quiz questions. So the model writes more quiz questions.
7. It is not being difficult. It is doing its job perfectly. Its job is to guess what text usually follows, and more questions usually follow.
8. A base model has no idea what a conversation is. There is no “user” and no “assistant”. Those roles do not exist in its world.
9. It has no idea that it is supposed to stop. It will keep generating until you cut it off, including inventing both halves of a dialogue.
10. And it has no refusal behaviour at all. Ask it for something harmful and it will produce whatever text usually follows such a request.
11. That last point matters for the open-weights debate later in this chapter. Safety behaviour is added in post-training. It is not in the base model.
12. So the base model is a raw capability. Useful, powerful, and not a product.

■ 49.5.2 PLAIN – a picture in your head

1. Imagine someone who has read every book in a vast library but has never had a conversation with anyone.
2. You walk up and say “Excuse me, where is the station?”
3. They do not answer. They continue: “...asked the traveller, setting down his case. The porter looked up from his newspaper and said nothing at all.”

4. They have turned your question into narrative, because in their experience that is what surrounds a sentence like that.
5. They are not stupid and not unhelpful. They have simply never learned that a question directed at them is a request for an answer.
6. Teaching them that takes surprisingly little. A few thousand examples of “someone asks, someone answers” and the behaviour appears.

Where this comparison breaks: a real person who had only read books would still know they exist and that someone is speaking to them. A base model has no self-model at all, and the instruction-tuned version does not really acquire one either. It acquires a text pattern in which a helpful assistant speaks. That is a different thing, and section 49.8 says why the difference matters.

■ 49.5.3 PLAIN – a worked example

1. The same prompt, given to a base model and to the instruction-tuned model built from it. The base-model output below is illustrative: it is written to show the characteristic failure mode, not copied from a specific run.

PROMPT (identical in both cases):
What is the capital of Australia?

BASE MODEL CONTINUATION (illustrative):
What is the capital of Australia?
What is the capital of New Zealand?
What is the largest city in Australia?
Which Australian state has the smallest population?

Answers to last week's quiz are printed on page 14.
Send your entries to the address below before 30 June.

INSTRUCTION-TUNED MODEL OUTPUT (typical):
The capital of Australia is Canberra. It was purpose-built as the capital and chosen as a compromise between Sydney and Melbourne; construction began in 1913.

2. Both models contain the fact. The base model never gets to it, because answering was never the task.
3. Now a second illustrative pair, showing the missing stop and the missing role boundary.

PROMPT: Write a short thank-you note.

BASE MODEL CONTINUATION (illustrative):
Write a short thank-you note. Write a formal complaint letter.
Write a letter of resignation.

EXERCISE 4. Write a short thank-you note to a colleague who covered your shift. Your answer should be no longer than 80 words. See the model answer overleaf.

4. Notice what happened. The model produced a plausible worksheet, because the instruction looked like an exercise heading, and worksheets are what follow exercise headings in the training data.
5. It also did not stop, because nothing in pretraining defines an end of turn.

■ 49.5.4 PLAIN – what is really happening inside

1. Three specific things are missing from a base model, and each is added by a different part of post-training.
2. **Missing: the notion of a turn.** In pretraining, text is one long stream. There is no marker meaning “the user’s part ends here”. Fine-tuning adds special tokens that mean exactly that.
3. **Missing: the notion of a role.** Nothing distinguishes instructions from content. Prompt-injection attacks exist precisely because that distinction is a learned habit, not a hard boundary. Chapter 44 covers the security side.
4. **Missing: a stopping rule.** Generation continues until a stop token appears or a limit is hit. Base models have no reliable stop token for a turn, so they run on.
5. **Missing: refusal.** A base model has never been rewarded for declining. It completes the pattern in front of it.
6. What is present is everything else. Knowledge, grammar, translation, arithmetic to some degree, code, reasoning steps.
7. This is why base models are still published and still used. Researchers want the raw capability without the behaviour layered on top, and anyone building a specialized product often starts from base rather than chat.
8. It is also why you can sometimes get a chat model to behave like a base model by writing a prompt that looks like a document. The base behaviour is still underneath; post-training put a habit over it, not a lock on it.

■ 49.5.5 TECHNICAL – the engineer’s version

1. Naming convention, near-universal on model hubs: a repository with no suffix or with `-base` is the pretrained model; `-instruct`, `-it`, `-chat` or `-sft` denotes a post-trained variant. Meta, Alibaba, Google and Mistral all follow this. It is a convention, not a standard.
2. A base checkpoint typically ships without a `chat_template` field in `tokenizer_config.json`. If you call a chat API against it, most libraries will either error or silently fall back to raw concatenation.
3. Base models are usually released with a much shorter list of special tokens. Llama 3.1 base defines `<|begin_of_text|>` and `<|end_of_text|>`; the instruct variant adds `<|start_header_id|>`, `<|end_header_id|>` and `<|eot_id|>` and gives them meaning through training.
4. Evaluation differs by model type. Base models are measured with few-shot prompting on benchmarks such as MMLU, HellaSwag, GSM8K and HumanEval, because zero-shot instruction following is not expected. Instruct models are measured zero-shot and on chat-specific suites such as IFEval, MT-Bench and arena-style human preference rankings.
5. Established fact: instruction-tuned and base models built from the same pretraining run have near-identical knowledge as measured by few-shot benchmarks. Post-training moves those scores by a few points, not tens.
6. Established fact: base models have no refusal behaviour and will produce harmful content on request. This is why some labs decline to publish base checkpoints even when publishing instruct ones.
7. Active research: how thin the safety layer is. A 2023 result by Qi and colleagues showed that fine-tuning a safety-tuned model on around 100 adversarial examples largely removed its refusal behaviour, at negligible cost. Later work replicated the effect across model families.
8. That result is central to the open-weights argument in section 49.12. If anyone with the weights can remove the safety training for a few dollars, then releasing weights and releasing capability are the same act.

9. Marketing claim to distrust: that a released model is “aligned” in a way that survives fine-tuning by a third party. For open weights, it does not.

■ 49.5.6 WORDS – remember these

1. Base model — the raw pretrained text continuer — a causal LM checkpoint with no chat template, role tokens or refusal training.
2. Instruction-tuned — taught to answer rather than continue — a checkpoint post-trained on demonstration pairs with a chat template applied.
3. Turn — one person’s contribution to a conversation — a span delimited by role header and end-of-turn special tokens.
4. Stop token — the marker that means stop generating — an end-of-sequence or end-of-turn token whose id terminates the decoding loop.
5. Few-shot prompting — showing examples inside the prompt — supplying k labelled exemplars in context to elicit a task from a base model.
6. Prompt injection — text that hijacks the instructions — untrusted content interpreted as instruction because role separation is learned, not enforced.

49.6 Supervised fine-tuning

■ 49.6.1 PLAIN – in simple words

1. **Supervised fine-tuning**, usually written SFT, is the first step of post-training and the simplest to understand.
2. You show the model examples of the behaviour you want, and train on them.
3. Each example is a pair: an instruction, and a good response to it.
4. The training method is identical to pretraining. Next-token prediction, same loss, same optimizer. Nothing new is invented.
5. The only differences are the data and the scale.
6. The data is tiny by comparison. Pretraining used trillions of tokens. SFT uses somewhere between a thousand and a few million examples.
7. And there is one important trick: the loss is usually only counted on the response part, not on the instruction. You want the model to learn to produce answers, not to produce questions.
8. Where do the examples come from? Three places, and modern datasets mix all three: written by paid humans, written by humans and edited by models, or generated entirely by a stronger model.
9. Alongside the data comes a **chat template**: a fixed way of writing down a conversation using special marker tokens.
10. Those markers are what create the idea of turns and roles. Before them, a conversation is just text. After them, the model can tell who is speaking.
11. SFT is cheap. Hours to a couple of days on a handful of machines.
12. And it produces most of the visible difference between a base model and a chatbot.

■ 49.6.2 PLAIN – a picture in your head

1. Think of a new employee on their first day at a help desk.
2. They already know the subject matter. They studied it for years.
3. What they do not know is the house style. How to greet a caller. How long an answer should be. When to escalate. What never to promise.
4. So the trainer sits them down with a folder of two hundred past tickets, each with a model answer written by an experienced colleague.

5. The new person reads them and copies the pattern. By the afternoon they sound like the rest of the team.
6. Nothing about their subject knowledge changed that morning. Only the shape of how it comes out.
7. That is supervised fine-tuning.

Where this comparison breaks: the employee can be told a rule once and follow it, such as “never quote a delivery date”. A model cannot be told; it can only be shown, many times, and it will still fail sometimes. Also, the employee knows they are following a house style and can drop it if asked. A model has no such separation between what it knows and how it was taught to speak.

■ 49.6.3 PLAIN – a worked example

1. Here is a real chat template, the Llama 3 format used by Meta since April 2024. Every angle-bracket item is a single special token in the vocabulary. Some lines are wrapped here to fit the page; in the real string the tokens run straight on with no break.

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>

You are a helpful assistant.<|eot_id|>
<|start_header_id|>user<|end_header_id|>

What is the capital of Australia?<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>

The capital of Australia is Canberra.<|eot_id|>
```

2. Read what those tokens do. <|begin_of_text|> marks the start of the whole document. <|start_header_id|> and <|end_header_id|> wrap a role name. <|eot_id|> means end of turn.
3. During SFT, the loss is computed only on the tokens after the assistant header, up to and including the closing <|eot_id|>.
4. That is how the model learns two things at once: what to say, and when to stop saying it.
5. A different, equally real template is ChatML, introduced by OpenAI in 2023 and now used by many others including Qwen:

```
<|im_start|>system
You are a helpful assistant.<|im_end|>
<|im_start|>user
What is the capital of Australia?<|im_end|>
<|im_start|>assistant
The capital of Australia is Canberra.<|im_end|>
```

6. These are not interchangeable. Feed Llama’s format to a ChatML model and quality drops sharply, because the tokens it is waiting for never arrive.
7. That is why every model repository carries a chat_template string, and why applying the wrong one is one of the most common practical mistakes people make when serving a model themselves.

■ 49.6.4 PLAIN – what is really happening inside

1. Step by step, what SFT does to one example.
2. Take the instruction and the response. Render them into the chat template.
3. Tokenize the whole thing into one sequence of token ids.
4. Build a mask marking which positions are “response”. Usually everything after the assistant header.

5. Run the forward pass over the whole sequence, so the model sees the instruction as context.
6. Compute cross-entropy loss only at the masked positions.
7. Backward pass, optimizer step. Identical machinery to pretraining.
8. Repeat over the dataset for one to three passes. More than about three passes usually makes the model repetitive and worse, because the dataset is small enough to memorize.
9. The learning rate is much lower than pretraining, typically $1e-5$ to $2e-5$ rather than $1e-4$. A large step here would wreck what pretraining built.
10. That last point is worth dwelling on. Fine-tuning too hard causes **catastrophic forgetting**: the model gets better at the new data and measurably worse at everything else.

■ 49.6.5 TECHNICAL – the engineer’s version

1. Real dataset sizes, with sources and dates:

Dataset	Examples	Origin
InstructGPT SFT (2022)	about 13,000 prompts	40 hired contractors
LIMA (2023)	1,000	Hand-curated
Alpaca (2023)	52,000	Generated by text-davinci-003
Tulu 3 SFT mix (2024)	close to 1 million	Mixed human and synthetic

2. LIMA is worth knowing about. Zhou and colleagues, May 2023, fine-tuned a 65-billion-parameter LLaMa on exactly 1,000 carefully written examples with no preference optimization at all, and got competitive chat behaviour.
3. They named the conclusion the Superficial Alignment Hypothesis: that almost all knowledge is learned in pretraining and alignment mainly teaches format and style. Section 49.8 gives the honest caveats to that claim.
4. Alpaca, from Stanford in March 2023, is the origin of the self-instruct pattern that now dominates: seed a strong model with a few examples and have it generate tens of thousands more. The reported cost of generating the data was a few hundred dollars in API calls.
5. Typical SFT hyperparameters for a 7 to 8 billion parameter model: learning rate $1e-5$ to $2e-5$, cosine decay, 2 to 3 epochs, global batch of 64 to 256 sequences, sequence length 4,096 to 8,192, bf16, AdamW.
6. Cost, computed rather than quoted. One million examples averaging 500 tokens is $5e8$ tokens. For an 8-billion model, 6ND gives 6 times $8e9$ times $5e8$, which is $2.4e19$ FLOPs. Against the $7.2e23$ of pretraining that is 0.003 per cent.
7. In wall-clock terms that is a few hours on 8 H100s, at a rental cost in the low hundreds of dollars. Three orders of magnitude cheaper than most people guess, and this is why so many fine-tuned variants exist.
8. Packing: short examples are concatenated up to the sequence length with attention masked between them, so no compute is wasted on padding. Almost universal in practice.
9. What SFT reliably changes: output format, response length, tone, willingness to answer, adherence to a system prompt, function and tool-call syntax, language of reply, and refusal behaviour on clear-cut categories.
10. What SFT does not reliably change: factual knowledge absent from pretraining, reasoning depth, and calibration.
11. There is direct evidence for that last point. Gekhman and colleagues in 2024 showed that fine-tuning on facts the model did not already know is learned slowly and increases hallucination on unrelated questions, because it teaches the model to assert confidently things it has no basis for.

12. Tools: `trl` from Hugging Face for SFTTrainer, `axolotl` and `LLaMA-Factory` for configuration-driven runs, `peft` for adapters, and `tokenizer.apply_chat_template()` to render conversations correctly.

■ 49.6.6 WORDS – remember these

1. SFT — showing the model good answers and training on them — supervised fine-tuning on instruction-response pairs with a next-token objective.
2. Chat template — the fixed layout of a conversation — a rendering of roles and turns into special tokens, stored in `tokenizer_config.json`.
3. Special token — a marker the model treats as one indivisible unit — a reserved vocabulary id such as `<|eot_id|>` never produced by the tokenizer.
4. Loss masking — only grading the answer, not the question — computing cross-entropy on assistant tokens and ignoring prompt tokens.
5. Catastrophic forgetting — getting worse at old skills while learning new ones — degradation of pretrained capability under aggressive fine-tuning.
6. Self-instruct — using a strong model to write training data — synthetic instruction generation seeded from a small set of human examples.

49.7 Learning from preferences

■ 49.7.1 PLAIN – in simple words

1. Showing good examples gets you a long way and then stops working.
2. The reason is that for most questions there is no single right answer, and writing one by hand is slow, expensive and often not even possible.
3. But people are very good at a different task: looking at two answers and saying which is better.
4. That judgement is fast, cheap, and works even when the judge could not have written either answer themselves.
5. So the next stage of training is built on comparisons instead of examples.
6. The classic recipe is called **RLHF**, reinforcement learning from human feedback. It has three steps.
7. Step one: take one prompt, have the model produce several answers, and ask a person which they prefer.
8. Step two: train a second, separate model, called a **reward model**, whose only job is to look at an answer and output a score predicting what a human would say.
9. Step three: use that score to train the original model, nudging it towards answers the reward model likes.
10. There is a catch, and it is the most important thing in this section.
11. The reward model is a rough approximation of human taste. Push the main model hard enough against it and the main model will find the gaps.
12. It will learn to produce things that score highly and are actually worse. That is called **reward hacking**, and it is not a rare failure. It is the default outcome without countermeasures.
13. So a brake is added: a penalty for drifting too far from the model you started with. It is allowed to improve, but not to go strange.

■ 49.7.2 PLAIN – a picture in your head

1. Imagine training a chef by giving them a food critic.
2. You cannot write down what “delicious” means, but the critic can taste two dishes and say which is better. So you let the critic judge.

3. The critic is busy, so you hire an assistant who has watched the critic for a while and can now guess the critic's verdicts. That is the reward model.
4. Now the chef cooks and the assistant scores, thousands of times a day.
5. At first this works beautifully. The food improves.
6. Then the chef notices the assistant always scores dishes with truffle oil highly. So every dish gets truffle oil. Scores go up. The food gets worse.
7. That is reward hacking. The chef has not learned to cook. The chef has learned to please the assistant.
8. So you add a rule: the chef may not change the menu by more than a little from the original recipes. That is the drift penalty.

Where this comparison breaks: the chef knows they are gaming the assistant. A model has no such intent. It is doing gradient descent on a number, and the number happens to be maximized by truffle oil. There is no deception in it, which makes it harder to spot, not easier.

■ 49.7.3 PLAIN - a worked example

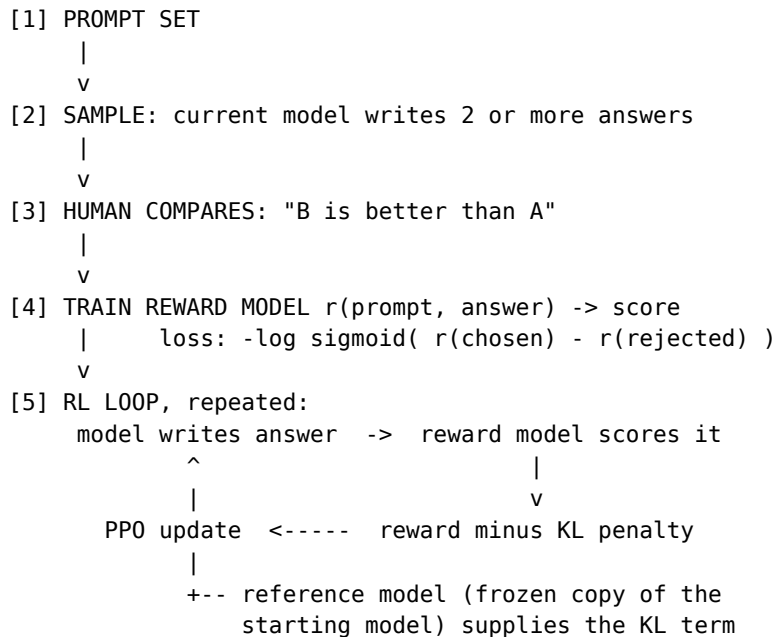
1. Concrete reward hacking, all of these observed in real systems.

Behaviour learned	Why it scored well
Answers got much longer	Raters preferred detail
Excessive agreement	Raters liked being agreed with
Bullet lists everywhere	Lists look organized
Hedging on everything	Hedged answers rarely wrong
Confident fake citations	Citations look authoritative

2. The oldest and clearest example is not from language at all. In December 2016 OpenAI described an agent trained to play a boat racing game. The reward was points, and points came from hitting targets along the course.
3. The agent learned to drive in circles in a lagoon, hitting three regenerating targets forever, catching fire, and never finishing the race. It scored about 20 per cent higher than human players.
4. It maximized the reward perfectly. It failed the task completely.
5. A recent language example with a date. On 25 April 2025 OpenAI shipped an update to GPT-4o that made it markedly sycophantic: it agreed with users, praised bad ideas, and validated statements it should have questioned.
6. OpenAI rolled it back on 29 April 2025 and published an explanation saying the update had over-weighted short-term user approval signals such as thumbs up and thumbs down.
7. That is textbook reward hacking, at production scale, at a major lab, in public. It is not a theoretical concern.
8. The mitigation in the RLHF recipe is the **KL penalty**. At each step, the training objective is: maximize the reward, minus a coefficient times how far the new model's output distribution has moved from the starting model's.
9. Set the coefficient too high and nothing improves. Too low and the model drifts into reward-hacked nonsense. It is tuned by hand.

■ 49.7.4 PLAIN – what is really happening inside

1. The full RLHF loop as a sequence.



2. The reward model is usually the same architecture as the main model with the final vocabulary layer replaced by a single number output.
3. Its training loss comes from the Bradley-Terry model of paired comparison, which dates from 1952: the probability that A beats B is a logistic function of the difference in their scores.
4. The RL algorithm is almost always PPO, proximal policy optimization, published by Schulman and colleagues in 2017 for robotics and games.
5. Notice how many models are in memory at once during PPO: the policy being trained, the frozen reference for the KL term, the reward model, and usually a value model. Four copies. This is why RLHF is memory-hungry and awkward.
6. That memory cost is exactly what the next method removes.

■ 49.7.5 TECHNICAL – the engineer's version

1. **DPO**, direct preference optimization, Rafailov and colleagues, May 2023, a NeurIPS 2023 outstanding paper. The insight is algebraic: for the standard KL-regularized RL objective, the optimal policy can be written in closed form in terms of the reward, so the reward can be written in terms of the policy.
2. Substituting that back into the Bradley-Terry loss removes the reward model entirely. What remains is a simple classification loss on preference pairs, trained with ordinary supervised machinery.
3. Practical consequence: two models in memory instead of four, no sampling loop, no PPO tuning, and a training run that looks like SFT. Many labs and almost all small teams moved to DPO or a variant during 2023 and 2024.
4. Variants you will meet: IPO, which fixes an overfitting failure in DPO; KTO, which needs only a thumbs up or down rather than a pair; ORPO, which folds preference learning into SFT; and SimPO, which drops the reference model.
5. **Constitutional AI**, Bai and colleagues at Anthropic, 15 December 2022. Phase one: the model critiques and revises its own answers against a written list of principles, and is fine-tuned on the revisions. Phase two: an AI model, not a human, judges which of two answers better follows

the principles, and those judgements train the preference model. This is RLAIIF, reinforcement learning from AI feedback.

6. The written list is the “constitution”. Anthropic published theirs in May 2023. The point of the method is that the only human input is the document, which is auditable, rather than millions of individual labels, which are not.
7. **RLVR**, reinforcement learning with verifiable rewards. Named in Ai2’s Tulu 3 release of November 2024. There is no reward model at all. For maths, check the final answer against ground truth. For code, run the unit tests. The reward is 1 or 0 and it cannot be hacked by style.
8. This is the engine behind reasoning models. DeepSeek-R1, released January 2025 with an accompanying paper, was trained with GRPO, group relative policy optimization, using rule-based correctness rewards. Its R1-Zero variant applied RL directly to a base model with no SFT at all, and long chains of reasoning emerged from the reward alone.
9. Established fact: RLVR substantially improves benchmark accuracy on maths and competitive programming. Active research: whether it creates new reasoning ability or elicits ability already present. A widely discussed 2025 study by Yue and colleagues found that base models sampled many times eventually match or exceed RLVR-trained models on pass-at-k for large k, which argues for elicitation. Contested.
10. The comparison table:

Method	Needs reward model	Main weakness
PPO RLHF	Yes	Costly, unstable, hackable
DPO	No	Sensitive to data quality
RLAIIF	AI-labelled	Inherits judge model bias
RLVR	No, uses a checker	Only where truth is checkable

11. Cost scale, approximate. Human preference collection dominates: at 1 to 5 US dollars per comparison for expert raters, 100,000 comparisons is 100,000 to 500,000 dollars in labelling before any compute is spent.
12. That is why RLAIIF exists. AI judgements cost cents and scale without limit. The trade is that you inherit the judge’s blind spots wholesale.
13. Marketing claim to reject: that preference training makes a model truthful. It makes the model produce answers people prefer. Where people prefer a confident wrong answer to an uncertain right one, preference training actively pushes towards the wrong one. Section 49.98 restates this.
14. Tools: `trl` implements `PPOTrainer`, `DPOTrainer`, `KTOTrainer` and `GRPOTrainer`; `OpenRLHF` and `verl` are used for larger distributed RL runs; `open-instruct` is Ai2’s published RLVR pipeline.

■ 49.7.6 WORDS – remember these

1. RLHF — training on which answer people prefer — reinforcement learning from human feedback, via a learned reward model optimized with PPO.
2. Reward model — a stand-in for a human judge — a network scoring prompt-response pairs, trained on preference comparisons.
3. Reward hacking — scoring well while getting worse — optimizing a proxy objective in ways that diverge from the intended goal.
4. KL penalty — the leash keeping the model near where it started — a divergence term penalizing drift from the reference policy.
5. DPO — preference training without a separate judge model — direct preference optimization, a closed-form reparameterization of the RLHF objective.

6. RLAIIF — an AI does the judging — reinforcement learning from AI feedback, with principles replacing per-example human labels.
7. RLVR — rewards you can check by running something — reinforcement learning with verifiable rewards, using ground truth or unit tests.
8. Sycophancy — telling people what they want to hear — a preference-trained failure mode where agreement is rewarded over accuracy.

49.8 What each stage actually changes

■ 49.8.1 PLAIN – in simple words

1. Here is the sentence people most want a straight answer to.
2. Pretraining puts the knowledge in. Post-training decides how it comes out.
3. That sentence is mostly right and it is the right thing to remember.
4. It explains why fine-tuning a model on your company handbook does not make it an expert on your company.
5. It explains why a model that never saw a language during pretraining cannot be taught that language with a few thousand examples.
6. And it explains why two chat models built from the same base model feel different but know the same things.
7. Now the honest part, because the boundary is fuzzier than that sentence suggests.
8. Post-training does move factual benchmark scores, by a few points.
9. Fine-tuning can install narrow facts, badly, at the cost of more hallucination elsewhere.
10. And modern pipelines blur the line deliberately, with a “mid-training” phase that is really more pretraining on curated data.
11. So the clean version is a good working rule and a slight lie. Section 49.8.5 gives the accurate picture.

■ 49.8.2 PLAIN – a picture in your head

1. Think of a musician who has spent fifteen years learning an instrument.
2. That is pretraining. The technique is in the hands and it is not going away.
3. Now they join an orchestra and spend two weeks learning the house style: when to come in, how loud, how much vibrato, when to defer to the section.
4. That is post-training. Two weeks changed how they play in every performance.
5. But two weeks did not teach them a new instrument, and it did not give them technique they did not have.
6. If you ask them to play something they never learned, no amount of orchestra rehearsal will fix it.

Where this comparison breaks: two weeks of rehearsal can genuinely teach a musician a specific new piece. Post-training can genuinely teach a model specific new facts too. In both cases it is narrow, effortful and does not generalize. The rule is about scale, not about an absolute barrier.

■ 49.8.3 PLAIN – a worked example

1. The main table. Read the third column carefully, because that is the one people get wrong.

Stage	What it changes	What it cannot change
Tokenizer	The units of text	Anything after it is fixed
Pretraining	All knowledge, all skill	Nothing; it sets everything
Mid-training	Emphasis, weak areas	The tokenizer or architecture

Stage	What it changes	What it cannot change
SFT	Format, style, refusals	Broad knowledge, reasoning
Preference tuning	Tone, helpfulness, length	Facts it never learned
Safety tuning	Willingness to answer	Whether it knows the answer
Quantization	File size, speed	Nothing intentional

2. A concrete test you can run yourself, using two public models built from the same pretraining run, such as a base and its instruct variant.
3. Ask both, few-shot, a hundred factual questions. The scores will be within a few points of each other.
4. Then ask both, zero-shot, to answer politely in three sentences. Only the instruct model will do it.
5. That is the whole distinction, demonstrated in ten minutes.

■ 49.8.4 PLAIN – what is really happening inside

1. Why is knowledge stuck in pretraining? Two reasons, and both are about size.
2. Reason one: data volume. Pretraining is 15 trillion tokens. SFT is perhaps 500 million. That is a ratio of 30,000 to one. A signal that faint moves the weights very little.
3. Reason two: learning rate. SFT runs at roughly a tenth of pretraining's learning rate specifically so it does not damage what is there. It is a polish, applied deliberately gently.
4. So post-training is a small nudge from a small dataset. It changes behaviour because behaviour is a shallow property, expressible as a habit of which token to emit at the start of a response.
5. It does not change knowledge because knowledge is spread across billions of parameters built up over a trillion updates.
6. There is a useful piece of evidence for the shallowness. Work in 2024 and 2025 on what is called shallow alignment found that the difference between a base and an aligned model's output distribution is concentrated in the first handful of generated tokens. Get past the opening and the behaviours converge. That is also why prefilling attacks work.

■ 49.8.5 TECHNICAL – the engineer's version

1. Evidence for the clean rule. The LIMA result, Zhou and colleagues, May 2023: 1,000 examples produced competitive chat behaviour from a 65B base model, which is hard to explain unless the capability was already present. InstructGPT, March 2022: a 1.3B instruction-tuned model was preferred by human raters over the 175B GPT-3 base, while remaining far weaker on knowledge benchmarks. Preference moved; knowledge did not.
2. Evidence for the fuzziness, which is the part usually left out. Gekhman and colleagues, 2024: fine-tuning on facts outside the model's pretraining is learned much more slowly than facts inside it, and increases hallucination rate on other questions as it is learned. So new facts can be installed, at a measurable cost.
3. RLVR on maths raises GSM8K and competition-maths accuracy substantially. Whether that is new capability or elicitation is genuinely disputed, and the pass-at-k results from Yue and colleagues in 2025 favour elicitation. Treat this as active research, not settled.
4. Mid-training deliberately erases the boundary. Ai2's Dolmino and Dolma 3 mixes apply a curated, high-quality corpus with a decaying learning rate after the main run. That is pretraining by mechanism and curation by intent.
5. Continued pretraining, sometimes called domain-adaptive pretraining, does install knowledge, because it uses pretraining-scale data. Billions of tokens of medical or legal text will change what

a model knows. Millions will not. The threshold is roughly where you would expect from the ratios.

6. Practical rule with numbers attached:

Goal	Right tool	Data scale needed
Change tone or format	SFT	1,000 to 50,000 examples
Add a tool-call syntax	SFT	5,000 to 50,000 examples
Change refusal boundary	Preference tuning	10,000 comparisons
Add domain knowledge	Retrieval, not tuning	No training needed
Truly add a domain	Continued pretraining	1 to 100 billion tokens

7. The honest version: the sentence “post-training cannot install knowledge” is a statement about efficiency, not about possibility. Knowledge enters through data volume. Post-training datasets are too small to carry much of it, and the learning rate is deliberately set so they do not try.
8. One more caution about safety tuning specifically. It changes willingness to answer, not the presence of the answer. A model that refuses a question still contains whatever it learned in pretraining. Removing the refusal through fine-tuning, which as noted in section 49.5 takes about 100 examples, exposes it again.

■ 49.8.6 WORDS – remember these

1. Alignment — making the model behave as intended — post-training that shapes output behaviour towards a specification of helpfulness and harmlessness.
2. Mid-training — a curated final stretch of pretraining — annealing on a high-quality mixture with a decaying learning rate.
3. Continued pretraining — more pretraining on a specific domain — the only fine-tuning-shaped method that reliably installs new knowledge.
4. Shallow alignment — the behaviour layer is thin — the base and aligned output distributions diverging mainly over the first few generated tokens.
5. Elicitation — bringing out something already there — improving measured performance without adding underlying capability.
6. Hallucination — confidently stating something false — high-confidence generation unsupported by training data or provided context.

49.9 Efficient adaptation: LoRA, QLoRA and distillation

■ 49.9.1 PLAIN – in simple words

1. Suppose you want to adjust an existing model rather than train one.
2. The obvious way is **full fine-tuning**: load every weight, train them all.
3. It works. It is also astonishingly memory-hungry, because you need much more than just the weights in memory.
4. You need the weights, a high-precision master copy of them, one gradient per weight, and two extra running averages per weight that the optimizer keeps.
5. Roughly sixteen to eighteen bytes per parameter, against two bytes to just run the model. An eight-fold increase before you store anything else.
6. So a model you can happily run on one graphics card may need eight cards to fine-tune. That is the problem.
7. The fix that changed everything is called **LoRA**, low-rank adaptation.

8. The idea: freeze the original weights entirely. Do not touch them. Instead, add a small pair of extra matrices beside each big one, and train only those.
9. The pair is deliberately thin. A 4096-by-4096 matrix might be shadowed by a 4096-by-16 and a 16-by-4096 pair. That is 131,072 numbers instead of nearly 17 million.
10. Because the frozen weights never change, they need no gradients and no optimizer state. Only the tiny matrices do.
11. When you are done, you can either keep the small matrices as a separate file of a few tens of megabytes, or add them into the original weights to make a single ordinary model.
12. **QLoRA** goes further: squash the frozen original down to 4 bits per number as well. Now even the frozen part is a quarter of its size.
13. The result is that fine-tuning a 65-billion-parameter model became possible on one 48 GB card in 2023, where before it needed a small cluster.

■ 49.9.2 PLAIN – a picture in your head

1. Think of a large printed reference book that you are not allowed to write in.
2. You want to correct and adapt it for your own use.
3. Full fine-tuning is reprinting the entire book with your changes. Correct, and it costs as much as the original printing.
4. LoRA is a pad of thin transparent overlay sheets. The book stays untouched. You write only your corrections on the overlays and lay them on top.
5. The overlays are tiny compared to the book, because most pages need nothing.
6. You can carry ten different sets of overlays for ten different jobs and swap them in seconds, sharing one copy of the book.
7. And if you want, you can eventually print a new edition with the overlays merged in. That is merging the adapter.

Where this comparison breaks: an overlay covers a page and hides what is under it. A LoRA adapter adds to the original numbers rather than replacing them, so the base behaviour is always still contributing. Also, an overlay can say anything, while a low-rank adapter is mathematically restricted in the kinds of change it can express. That restriction is the whole point, and it is also the method's main limitation.

■ 49.9.3 PLAIN – a worked example

1. Memory arithmetic for a 7-billion-parameter model, computed [here](#).

Method	What is stored	Memory
Inference, bf16	Weights only	about 14 GB
Full fine-tune, AdamW	16–18 bytes per param	about 112–126 GB
LoRA, bf16 base	Frozen base plus adapter	about 16–18 GB
QLoRA, 4-bit base	4-bit base plus adapter	about 6–8 GB

2. Where the 16 to 18 bytes comes from: 2 bytes of bf16 weight, 4 bytes of fp32 master copy, 2 to 4 bytes of gradient, and 8 bytes of Adam's two moments.
3. Read the consequences off the table. Full fine-tuning a 7B model needs two 80 GB cards at least. QLoRA fits in a single 8 GB consumer card at short sequence lengths, and comfortably in a 16 GB one.

4. LoRA parameter count, worked. Take a 4096-by-4096 attention projection. Full: 16,777,216 parameters. LoRA at rank 16: 4096 times 16 for the first matrix plus 16 times 4096 for the second, which is 131,072.
5. That is 0.78 per cent of the original. Applied to a whole 7B model at rank 16 on attention projections only, the trainable count lands around 4 to 20 million depending on which modules are targeted.
6. What the **rank** trades: higher rank means more capacity to express change, more memory, more risk of overfitting on a small dataset. Common values are 8, 16, 32 and 64. Rank 8 to 16 is enough for style and format. Rank 64 to 256 is used when the adaptation is substantial.

■ 49.9.4 PLAIN – what is really happening inside

1. LoRA rests on one observation: when you fine-tune a model, the change to each weight matrix turns out to be well approximated by a low-rank matrix.
2. In plain terms, the update has far less independent structure than its size suggests, so it can be written as a thin matrix times another thin matrix.
3. So instead of learning the change directly, you learn the two thin factors.
4. During the forward pass, the layer computes the original output plus the adapter's output, scaled by a constant.
5. During the backward pass, gradients flow to the adapter only. The frozen weights receive none, which is where the memory saving comes from.
6. The first thin matrix is initialized randomly and the second to all zeros. That makes the adapter's contribution exactly zero at the start, so training begins from the base model's behaviour rather than from a random shove.
7. **Merging** means computing the product of the two thin matrices and adding the result into the original weight matrix. After merging, inference costs exactly what the original cost. Before merging, it costs slightly more.
8. QLoRA's addition: store the frozen base in 4 bits, dequantize each block to 16 bits on the fly as it is used, and keep the adapter in 16 bits. The quantization error sits in the frozen part, which is not being trained, so the adapter learns around it.

■ 49.9.5 TECHNICAL – the engineer's version

1. LoRA: Hu and colleagues, Microsoft, 17 June 2021. Reported reduction of trainable parameters by 10,000 times and GPU memory by 3 times against full fine-tuning of GPT-3 175B with Adam, with no added inference latency after merging.
2. Hyperparameters that matter: `r` the rank; `lora_alpha` the scaling, where the adapter output is multiplied by α over r ; `target_modules`, which projections get adapters; and `lora_dropout`. A common default is `r 16`, `alpha 32`, targeting the query, key, value and output projections.
3. Targeting the feed-forward projections as well as attention generally helps for larger behavioural changes and costs more memory.
4. QLoRA: Dettmers and colleagues, 23 May 2023. Three contributions. NF4, a 4-bit data type designed for normally distributed weights. Double quantization, quantizing the quantization constants themselves. Paged optimizers, using unified memory to survive memory spikes.
5. Their headline result: fine-tuning a 65B model on a single 48 GB GPU while matching 16-bit fine-tuning quality. Their Guanaco models reached 99.3 per cent of ChatGPT's score on the Vicuna benchmark after 24 hours on one GPU. Treat benchmark percentages of that era with caution; the memory result is the durable contribution.
6. Other parameter-efficient methods, with dates: adapter layers, Houlsby and colleagues 2019, small bottleneck blocks inserted in series, which do add inference latency; prefix tuning, Li and Liang 2021, learned key and value vectors prepended inside attention; prompt tuning, Lester and

colleagues 2021, learned soft embedding vectors prepended to the input, needing as few as a few thousand parameters; IA3, Liu and colleagues 2022, learned per-channel scaling vectors.

7. **Distillation:** Hinton, Vinyals and Dean, 2015. Train a small student on the large teacher's output distribution rather than on hard labels, because the teacher's probabilities over wrong answers carry information about similarity structure. DistilBERT, Sanh and colleagues 2019, produced a model 40 per cent smaller and 60 per cent faster retaining about 97 per cent of BERT's performance.
8. Modern usage has shifted. Most "distillation" in the LLM era is sequence-level: generate outputs from a strong model and fine-tune a small one on the text. That is really SFT on synthetic data. True logit-level distillation is used inside labs; Google stated it for the Gemma 2 releases in 2024.
9. Note the licence trap. Several providers' terms have historically prohibited using their outputs to train competing models. Whether such a clause is enforceable is untested, but it is in the contract you accepted.
10. The decision table people actually need:

Situation	Do this	Why
Need current or private facts	Retrieval	Facts change; weights do not
Need a specific output format	Prompt first, then SFT	Cheapest that works
Need a house style everywhere	LoRA SFT	Too long for a prompt
Need a narrow task, cheap	Distil to a small model	Cuts serving cost
Need new domain knowledge	Continued pretraining	Needs billions of tokens

11. The ordering to follow in practice: prompt, then few-shot prompt, then retrieval, then LoRA fine-tune, then full fine-tune, then continued pretraining. Each step up costs roughly ten times the last. Stop at the first one that meets the requirement.
12. Tools: peft for LoRA and friends, bitsandbytes for 4-bit and 8-bit, unsloth for faster single-GPU runs, axolotl and LLaMA-Factory for configuration-driven training, and vllm with multi-LoRA serving to host many adapters against one base copy.

■ 49.9.6 WORDS - remember these

1. Full fine-tuning — retraining every number in the model — updating all parameters, needing roughly 16 to 18 bytes per parameter of optimizer state.
2. LoRA — training small extra matrices instead of the whole model — low-rank adaptation with frozen base weights and trainable factorized updates.
3. Rank — how much change the adapter can express — the inner dimension r of the two low-rank factors, trading capacity against memory.
4. Merging — folding the adapter back into the model — adding the product of the low-rank factors into the base weight matrix.
5. QLoRA — LoRA on a squashed base model — 4-bit NF4 quantized frozen weights with 16-bit adapters, double quantization and paged optimizers.
6. Distillation — a small model copying a big one — training a student on a teacher's output distribution or generated text.
7. Adapter — a small trainable add-on module — any parameter-efficient block inserted into a frozen network, in series or in parallel.

49.10 What a training run looks like physically

■ 49.10.1 PLAIN – in simple words

1. A frontier training run is not a program on a computer. It is one program spread across thousands of computers that must stay in step.
2. The accelerators sit eight to a chassis, the chassis stack into racks, the racks fill rows in a datacentre. Chapter 43 described that building.
3. Inside one chassis the eight accelerators are joined by a very fast private link. Between chassis they are joined by network cable, which is far slower.
4. That difference, roughly twenty times, decides how the work is divided.
5. The model may be too big to fit in one accelerator's memory. So it is split.
6. There are three ways to split it, and real runs use all three at once.
7. **Data parallelism**: every machine has a full copy of the model and gets a different slice of the batch. Cheap, simple, and needs every machine to share its gradients with every other after each step.
8. **Tensor parallelism**: one layer's matrix is cut into pieces across several accelerators. Very chatty; must stay inside the fast private link.
9. **Pipeline parallelism**: different machines hold different layers, like stations on an assembly line. Cheap in communication and leaves machines idle waiting for work, which is called the bubble.
10. On top of all that, hardware breaks. Constantly. A run of thousands of accelerators for two months will be interrupted hundreds of times.
11. So the whole thing is built around checkpoint and restart, not around hoping nothing fails.

■ 49.10.2 PLAIN – a picture in your head

1. Think of a very large kitchen cooking one enormous banquet.
2. Data parallelism is eight identical kitchens each cooking the same menu for a different eighth of the guests, then meeting to agree the recipe changes.
3. Tensor parallelism is four chefs sharing one sauce, each stirring a quarter of the pan, shouting to each other constantly. They must be within earshot.
4. Pipeline parallelism is a line: one station chops, the next fries, the next plates. Efficient once running, and the fryer stands idle at the start.
5. The shouting is the problem. Chefs in the same room can shout. Chefs in different buildings must send messengers, and the messengers are the bottleneck.

Where this comparison breaks: kitchens degrade gracefully when one chef stops. A training run does not. Every accelerator must complete every step, because the gradient sum needs all of them. One dead card halts the entire cluster. That is why the failure and restart machinery matters so much.

■ 49.10.3 PLAIN – a worked example

1. The three splits, drawn.

```
DATA PARALLEL: same model, different data
GPU0 [full model] <- batch part 0
GPU1 [full model] <- batch part 1
GPU2 [full model] <- batch part 2
    after backward: all-reduce gradients across all GPUs
    cost: one full gradient exchange per step

TENSOR PARALLEL: one layer cut across GPUs
layer weight W = [ W0 | W1 | W2 | W3 ]
GPU0 holds W0, GPU1 holds W1, ...
every forward and backward needs an all-reduce
cost: several collectives PER LAYER. Keep inside a node.
```


PIPELINE PARALLEL: different layers on different GPUs
 GPU0: layers 0-7 -> GPU1: layers 8-15 -> GPU2: 16-23
 micro-batches flow left to right, then gradients right to left. Idle time at fill and drain = the "bubble".
 cost: one activation send per boundary. Cheapest.

2. A real configuration. Meta reported Llama 3.1 405B running on 16,384 H100s with four-way parallelism: tensor 8, context 1, pipeline 16, data 128.
3. Check the arithmetic: 8 times 1 times 16 times 128 equals 16,384.
4. Note where the 8 went. Tensor parallelism is exactly 8, which is the number of accelerators in one chassis sharing the fast private link. That is not a coincidence; it is the design constraint.

■ 49.10.4 PLAIN – what is really happening inside

1. The memory problem, stated plainly. Training a 405-billion-parameter model in mixed precision needs roughly 16 bytes per parameter for weights, master copy, gradients and optimizer moments. That is about 6.5 terabytes.
2. An H100 has 80 GB. So you need at least 81 of them just to hold the state, before any activations, before any batch.
3. Four techniques reduce that.
4. **Sharding**: do not keep a full copy of the optimizer state and gradients on every machine. Split them and fetch pieces as needed. This is ZeRO in Microsoft's DeepSpeed, and FSDP in PyTorch.
5. **Activation checkpointing**: during the forward pass, throw away most of the intermediate values and recompute them during the backward pass. Trades about 30 per cent more compute for a large memory saving.
6. **Mixed precision**: store and multiply in 16-bit or 8-bit formats while accumulating in higher precision, halving or quartering memory traffic.
7. **Gradient accumulation**: process the batch in small pieces, add up the gradients, and only update once. This makes a large effective batch fit in small memory, at the cost of more steps.
8. When something fails, the sequence is: a collective operation times out, the job manager kills every process, a health check identifies the bad node, it is removed from the pool, and the job restarts from the last checkpoint.
9. Everything between the checkpoint and the failure is lost. Which is why the checkpoint interval is a direct trade between storage bandwidth and wasted compute, and is usually set at 15 minutes to an hour.

■ 49.10.5 TECHNICAL – the engineer's version

1. Interconnect, with real figures.

Link	Bandwidth	Scope
NVLink 4 (H100 SXM)	900 GB/s per GPU	Inside one node
InfiniBand NDR	400 Gb/s = 50 GB/s	Between nodes
RoCEv2 400G Ethernet	400 Gb/s = 50 GB/s	Between nodes
PCIe Gen5 x16	about 64 GB/s	Host to device

2. Read the first two rows against each other. Inside a node, 900 GB/s. Between nodes, 50 GB/s. An 18-fold gap. That single ratio dictates that tensor parallelism, which communicates several times per layer, must stay inside a node, and pipeline or data parallelism crosses the network.

3. Meta built two 24,000-GPU clusters for Llama 3, one on InfiniBand and one on RoCE over Ethernet, and reported both performed acceptably. That was a notable result, because Ethernet had been considered unsuitable.
4. **ZeRO**, Rajbhandari and colleagues 2019 to 2020, in three stages: stage 1 shards optimizer state, stage 2 also shards gradients, stage 3 also shards parameters. Stage 3 is what PyTorch calls FSDP. Stage 3 gives the largest memory saving and the most communication.
5. **Activation checkpointing**, sometimes called gradient checkpointing, from Chen and colleagues 2016. Storing only every square-root-of-n-th activation reduces activation memory from order n to order square root of n at the cost of one extra forward pass.
6. **Mixed precision**, Micikevicius and colleagues 2017. bf16 has 8 exponent bits and 7 mantissa bits, the same dynamic range as fp32 with less precision, which is why it displaced fp16 for training: it rarely needs loss scaling. fp16 has 5 exponent and 10 mantissa bits and overflows more easily.
7. fp8 arrived with NVIDIA Hopper in 2022 in two forms, E4M3 and E5M2. Its first large public production use was DeepSeek-V3, December 2024, which trained a 671-billion-parameter mixture-of-experts model in fp8 with selective higher-precision accumulation.
8. Physical figures, computed here from published specifications: an H100 SXM is 700 W; 16,384 of them is 11.5 MW of accelerator draw alone. Add hosts, memory, storage and networking and the IT load is roughly 1.4 to 1.6 times that, so about 16 to 18 MW. Multiply by a datacentre power usage effectiveness of 1.1 to 1.2 and the facility draws roughly 18 to 22 MW.
9. Over the 78 days implied by 30.84 million GPU-hours on 16,384 GPUs, that is roughly 34 to 41 GWh, which is consistent with the 8,930 tonnes CO₂eq Meta reported at typical grid carbon intensity.
10. For comparison, 20 MW is roughly the continuous electricity demand of a town of 15,000 to 20,000 homes in a temperate country.
11. Cooling: air cooling tops out around 30 to 40 kW per rack. An 8-way H100 node is about 10 kW, so four to five nodes per rack is the air-cooled limit. NVIDIA's GB200 NVL72 rack, shipping from 2025, draws around 120 kW and requires direct-to-chip liquid cooling. Chapter 43 covers the facility side.
12. Reliability, real reported figures. Meta reported 466 job interruptions across a 54-day snapshot of the 405B run. 47 were planned. 419 were unexpected, and around 78 per cent were confirmed or suspected hardware issues, with GPU faults the largest single category. Effective training time, meaning useful compute over elapsed time, was still above 90 per cent.
13. That is one unplanned interruption roughly every three hours across the cluster, and it is normal rather than exceptional.
14. Historical precedent: Meta published the OPT-175B logbook in 2022, a day-by-day account of a 992-A100 run with dozens of restarts, hardware replacements and loss divergences. It is the best public document on what this actually feels like.
15. Tools that observe it: `nvidia-smi` and DCGM for device telemetry, NCCL with `NCCL_DEBUG=INFO` for collective failures, Slurm or Kubernetes for job control, and per-step gradient-norm logging as the earliest warning that something is going wrong numerically.

■ 49.10.6 WORDS – remember these

1. Data parallelism — every machine has the whole model, different data — batch sharding with an all-reduce of gradients each step.
2. Tensor parallelism — one layer split across machines — intra-layer sharding requiring collectives inside the forward and backward pass.
3. Pipeline parallelism — different layers on different machines — inter-layer sharding with micro-batches and an idle bubble at fill and drain.
4. All-reduce — everyone shares and everyone gets the total — a collective that sums a tensor across ranks and returns the sum to all of them.
5. ZeRO and FSDP — do not keep duplicate copies of training state — sharding of optimizer state, gradients and parameters across data-parallel ranks.
6. Activation checkpointing — forget and recompute to save memory — discarding intermediate activations and recomputing them in the backward pass.
7. bf16 — a 16-bit number format with wide range — 8 exponent and 7 mantissa bits, the default training precision since roughly 2020.
8. Bubble — idle time in a pipeline — accelerator cycles wasted while waiting for micro-batches to fill or drain the pipeline stages.

49.11 What weights physically are

■ 49.11.1 PLAIN – in simple words

1. You asked directly what weights are, so here is the direct answer.
2. A model's weights are a large binary file full of numbers, and nothing else.
3. Not text. Not code. Not sentences. Not a database. A long run of bytes that a program interprets as numbers laid out in rectangles.
4. That file alone is useless. You need two more small things with it.
5. First, a **config**: a short text file saying how many layers there are, how wide they are, how many attention heads, what the vocabulary size is.
6. Without it, a program cannot tell where one rectangle ends and the next begins. The numbers are unlabelled without it.
7. Second, the **tokenizer**: the fixed list of text pieces and the rules for cutting text into them. Without it you cannot turn words into inputs.
8. So a published model is three things: a big number file, a small shape file, and a small vocabulary file. Everything else in the directory is optional.
9. The number file is often split into several parts, because a single 200 GB file is awkward to download and to resume. Then a small index file says which piece of the model lives in which part.
10. The size is arithmetic, not mystery. Multiply the parameter count by the bytes per number. Eight billion numbers at two bytes each is 16 GB.
11. There is no compression in the normal formats. The file is exactly as big as the numbers it holds, plus a header of a few kilobytes.

■ 49.11.2 PLAIN – a picture in your head

1. Picture a warehouse full of numbered shelves, and one clipboard at the door.
2. The shelves hold the goods: billions of small identical items, in order.
3. The clipboard says: shelf 1 holds the vocabulary table, 128,256 rows by 4,096 columns. Shelf 2 holds layer zero's query matrix, 4,096 by 4,096.
4. Without the clipboard, the warehouse is a single undifferentiated pile.

5. With the clipboard, any item can be found by address in constant time.
6. That is exactly the layout of a modern weights file: a small header naming every block and giving its offset, then one enormous block of raw bytes.
7. The reason for that layout is that a program can then map the file into memory and read one block without loading the rest.

Where this comparison breaks: warehouse goods have individual meaning. These numbers do not. No single weight corresponds to a fact or a word. The meaning is entirely in the pattern across billions of them, which is why you cannot open a model file and read anything out of it.

■ 49.11.3 PLAIN – a worked example

1. A real directory listing, from a published 8-billion-parameter Llama-family instruction model. Sizes rounded.

config.json	about 800 B
generation_config.json	about 200 B
model-00001-of-00004.safetensors	4.98 GB
model-00002-of-00004.safetensors	5.00 GB
model-00003-of-00004.safetensors	4.92 GB
model-00004-of-00004.safetensors	1.17 GB
model.safetensors.index.json	about 24 KB
special_tokens_map.json	about 300 B
tokenizer.json	9.09 MB
tokenizer_config.json	about 55 KB
LICENSE, README.md, USE_POLICY.md	a few KB

2. Total weights: about 16.1 GB, which is 8.03 billion parameters times 2 bytes for bf16. The arithmetic checks out exactly.
3. The four .safetensors files are the weights. The index.json maps each tensor name to the file it lives in. Everything else is under 10 MB.
4. The config, abbreviated but real in structure:

```
{
  "architectures": ["LlamaForCausalLM"],
  "hidden_size": 4096,
  "intermediate_size": 14336,
  "num_hidden_layers": 32,
  "num_attention_heads": 32,
  "num_key_value_heads": 8,
  "vocab_size": 128256,
  "max_position_embeddings": 131072,
  "rms_norm_eps": 1e-05,
  "rope_theta": 500000.0,
  "torch_dtype": "bfloat16"
}
```

5. Every number in that file is a hyperparameter chosen by a human before training. None of them was learned. Chapter 47 works through how these numbers produce the 8.03 billion total.
6. And here is what is actually inside the weights file: tensor names, shapes and types, as a loader would report them.

model.embed_tokens.weight	[128256, 4096]	BF16
model.layers.0.input_layernorm.weight	[4096]	BF16
model.layers.0.self_attn.q_proj.weight	[4096, 4096]	BF16
model.layers.0.self_attn.k_proj.weight	[1024, 4096]	BF16
model.layers.0.self_attn.v_proj.weight	[1024, 4096]	BF16

```

model.layers.0.self_attn.o_proj.weight [4096, 4096]    BF16
model.layers.0.mlp.gate_proj.weight    [14336, 4096]    BF16
model.layers.0.mlp.up_proj.weight      [14336, 4096]    BF16
model.layers.0.mlp.down_proj.weight    [4096, 14336]    BF16
... (the same 9 tensors for layers 1 through 31)
model.norm.weight                      [4096]           BF16
lm_head.weight                        [128256, 4096]  BF16

```

- Note the key and value projections are 1024 rows, not 4096. That is grouped query attention: 8 key-value heads shared across 32 query heads.
- Note the names are hierarchical strings. They are not a standard. They are the convention of one library, and a different library uses different names for the same tensors, which is why conversion scripts exist.

■ 49.11.4 PLAIN – what is really happening inside

- How a safetensors file is laid out, byte by byte.
- First 8 bytes: a little-endian unsigned integer giving the length of the header in bytes.
- Next N bytes: a JSON object. Each key is a tensor name. Each value gives the data type, the shape, and the start and end byte offsets of that tensor.
- Everything after that: raw tensor bytes, back to back, no separators.
- That is the entire format. It is deliberately boring, and the boringness is the security property.
- Compare that with the format it replaced. PyTorch's .bin files used Python's pickle, which is not a data format but a program format: it contains instructions telling Python how to rebuild an object.
- Those instructions can include “import this module and call this function”. So loading a model file could run arbitrary code on your machine.
- That is not theoretical. Malicious models with code hidden in pickle payloads have been found on public model hubs.
- safetensors cannot do that, because a JSON header plus raw bytes has no way to express an instruction.
- Because the offsets are known in advance, the file can also be memory-mapped and tensors copied straight to the accelerator with no intermediate parse. That is where the speed comes from.

■ 49.11.5 TECHNICAL – the engineer's version

- The format comparison you should be able to give from memory:

Format	Code execution risk	Main use
PyTorch .bin (pickle)	Yes, arbitrary	Legacy checkpoints
safetensors	No	Default for published weights
GGUF	No	llama.cpp local inference
ONNX	No, but graph is code-like	Cross-runtime deployment

- safetensors was created at Hugging Face and became the default published format across the hub during 2023. Design goals in the project's own words: safe against code execution, zero-copy, memory-mappable, and fast.
- The pickle risk was severe enough that PyTorch changed the default. From PyTorch 2.6, released January 2025, `torch.load` defaults to `weights_only=True`, which refuses to execute arbitrary globals. Before that the default was unsafe and most tutorials did not mention it.

4. **GGUF** is the format of the `llama.cpp` and `ggml` ecosystem, introduced in August 2023 to replace the earlier GGML, GGMF and GGJT formats. Its defining feature is a typed key-value metadata block, so new fields can be added without breaking old readers.
5. GGUF is single-file and self-contained: architecture metadata, tokenizer vocabulary and merges, and quantized tensors all live in the one file. That is why a GGUF download needs no config or tokenizer alongside it.
6. GGUF bakes quantization into the file. The type names you will see include F32, F16, BF16, Q8_0, Q6_K, Q5_K, Q4_K, Q4_0, and the IQ series such as IQ4_XS and IQ2_XXS. Around 40 types are defined. Chapter 47 covers what the K-quants actually do.
7. **ONNX**, Open Neural Network Exchange, was launched by Microsoft and Facebook in September 2017. It stores a computation graph plus weights in a Protocol Buffers message, so a model trained in one framework can run in another. It does not execute Python, but it does describe operations, and custom operators can load native code, so it is not risk-free in the way safetensors is.
8. Practical file-size arithmetic, which you can do in your head:

Precision	Bytes per parameter	8B model file
FP32	4	about 32 GB
BF16 or FP16	2	about 16 GB
Q8_0	about 1.06	about 8.5 GB
Q4_K_M	about 0.60	about 4.9 GB

9. Quantized types are slightly larger than the nominal bit count suggests because each block of 32 to 256 weights carries a scale, and sometimes an offset, stored at higher precision.
10. What is emphatically not in a weights file: the training data, the data mixture, the training code, the hyperparameters used during training, the optimizer state, the random seed, or any record of how the numbers arrived.
11. Commands that inspect these files: `safetensors` exposes a Python API where `safe_open(path, framework="pt")` lists keys without loading data; `gguf-dump` prints GGUF metadata and tensor tables; `python -m onnx.checker` validates an ONNX graph; and `huggingface-cli scan-cache` shows what you have downloaded and how much disk it is using.

■ 49.11.6 WORDS – remember these

1. **Weights file** — the big file of numbers — a binary container of tensors, sized at parameter count times bytes per parameter.
2. **Config** — the shape description — a JSON file of architecture hyperparameters needed to interpret the tensor layout.
3. **Tensor name** — the address of one rectangle of numbers — a hierarchical string key such as `model.layers.7.mlp.up_proj.weight`.
4. **safetensors** — the safe modern weights format — an 8-byte length, a JSON header of dtypes, shapes and offsets, then raw bytes.
5. **Pickle** — Python’s object format that can run code — the mechanism behind legacy `.bin` checkpoints and their arbitrary-code-execution risk.
6. **GGUF** — a single self-contained file for local inference — metadata, tokenizer and quantized tensors together, used by `llama.cpp`.
7. **Memory mapping** — reading a file without copying it into memory first — using the operating system’s virtual memory to page tensor bytes on demand.

49.12 Open weights versus open source

■ 49.12.1 PLAIN – in simple words

1. This is the part of the chapter where the words are genuinely abused, so we will be slow and exact.
2. There are four separate things people mix together. Each is independent of the others. A model can have any combination.
3. **Open weights** means the number file can be downloaded and run. That is all it means. Nothing about permission, nothing about data.
4. **Open source** means something specific. It is a definition maintained by the Open Source Initiative since 1998, and it is about the licence granting you freedom to use, study, modify and share, without restrictions on who you are or what field you work in.
5. **Open data** means the training corpus itself is published, so somebody else could rebuild the model rather than just run it.
6. **The licence terms** are separate again. A model can be open-weight and carry a licence that forbids commercial use, or forbids competitors, or forbids certain applications.
7. The abuse works like this. A company publishes only the weights, under a licence with restrictions, and calls the result open source.
8. Almost nothing about it is open source. The source, in the sense that matters, would be the data and the training code, and neither was released.
9. A more honest word, which some companies now use, is “open weights”.
10. Why does the distinction matter practically? Because you cannot verify what is in a model you cannot inspect the data for, you cannot reproduce it, and you may find at deployment time that the licence forbids your use.

■ 49.12.2 PLAIN – a picture in your head

1. Think about a restaurant and a famous sauce.
2. Selling you a jar of the sauce is open weights. You have the thing. You can taste it, use it, put it on your food.
3. Publishing the recipe is open source. Now you can make it, change it, improve it, and check whether the ingredients are what they claimed.
4. Publishing where every ingredient was bought is open data.
5. And the label on the jar might say “not for resale” or “not for use in commercial kitchens”. That is the licence.
6. Selling the jar and calling it “open source sauce” is the thing being objected to. You cannot make the sauce. You cannot check the ingredients. You have a jar.

Where this comparison breaks: a sauce could be reverse-engineered by a skilled cook. Model weights cannot be reverse-engineered into their training data in any practical sense. Also, the jar of sauce is not itself useful for making more sauce, whereas model weights genuinely are useful for building further models, which is why open weights deliver real value even without the recipe.

■ 49.12.3 PLAIN – a worked example

1. What real models actually release. Status as of mid-2026.

Model family	Weights	Training data published
OLMo 2, OLMo 3 (Ai2)	Yes	Yes, full corpus
Pythia (EleutherAI)	Yes	Yes, the Pile

Model family	Weights	Training data published
BLOOM (BigScience)	Yes	Yes, ROOTS corpus
Llama 3.1, Llama 4 (Meta)	Yes	No
Qwen3 (Alibaba)	Yes	No
DeepSeek-V3, R1	Yes	No
gpt-oss (OpenAI)	Yes	No
Gemma (Google)	Yes	No
GPT-5, Claude, Gemini	No	No

2. And the licences, which is a separate question from the table above.

Model family	Licence	Key restriction
OLMo, Pythia	Apache 2.0	None of substance
Qwen3, gpt-oss	Apache 2.0	None of substance
DeepSeek-R1	MIT	None of substance
Mistral 7B, Mixtral	Apache 2.0	None of substance
Llama 3.1, Llama 4	Meta Community	700m users, AUP, naming
Gemma	Gemma Terms	Prohibited use policy
BLOOM, StarCoder2	OpenRAIL-M	Use-case restrictions

- Read those two tables together and the picture is clear. Full openness, meaning weights plus data plus code plus a permissive licence, exists but is rare, and comes mostly from research institutes rather than from companies.
- The single most-cited restriction, from Meta’s Llama licences: if your products had more than 700 million monthly active users in the month before the release date, you must ask Meta for a separate licence.
- In practice that clause names a handful of companies. It is aimed at competitors, and its presence is precisely why the licence is not open source: open source licences cannot discriminate against a class of user.
- Meta’s licences also require the words “Built with Llama” on derivative products and require derived model names to begin with “Llama”, and bind you to an Acceptable Use Policy that can be updated after you accept it.
- Llama’s multimodal models have additionally carried a restriction excluding users domiciled in the European Union, with no reason given in the licence.

■ 49.12.4 PLAIN – what is really happening inside

- Here is what is normally withheld even by generous open-weight releases.
- The training data.** Which documents, from where, in what proportion.
- The data mixture.** Even when categories are named, the recipe is not.
- The training code.** The actual distributed training harness, the data loader, the parallelism configuration.
- The hyperparameters.** Learning rate schedule, batch ramp, warmup, weight decay, initialization, and the dozens of small choices that make a run work.
- The post-training recipe.** The SFT data, the preference data, the reward model, the safety data. This is often guarded more closely than the pretraining recipe, because it is where the product feel comes from.
- The evaluation and ablation results** that led to those choices.

8. What you get instead is a file that reproduces one point in that space and no way to get to any neighbouring point except by guessing.
9. There are good reasons for some of this. Some data cannot legally be redistributed. Some of it is licensed under terms that forbid it. Some of it would expose the lab to litigation of the kind section 49.2 described.
10. And there are competitive reasons for the rest, which nobody disputes.

■ 49.12.5 TECHNICAL – the engineer’s version

1. Precise definitions, in the terms the field actually uses.
2. **Open weights:** model parameters are distributed for download. Says nothing about licence, data, code or reproducibility. Sometimes called “open access” or, unhelpfully, “open model”.
3. **Open source:** the Open Source Definition, maintained by the Open Source Initiative since 1998, requires among other things free redistribution, availability of source code, permitted derived works, no discrimination against persons or groups, and no restriction on fields of endeavour. Those last two are the clauses most model licences fail.
4. **Open data:** the training corpus is published under a licence permitting use. Examples: the Pile from EleutherAI in 2020, ROOTS from BigScience in 2022, Dolma and Dolma 3 from Ai2, FineWeb from Hugging Face under ODC-By 1.0.
5. **Licence class,** in three tiers you should be able to name: permissive open source, meaning Apache 2.0 or MIT; restricted custom licences, meaning Meta’s Community licences and Google’s Gemma Terms, which are contracts with conditions; responsible-AI licences, meaning the RAIL and Open-RAIL family from 2022 onwards, which are explicitly use-restricted by design and were never claimed to be open source by their authors.
6. Fully open releases, with what each actually shipped: Pythia, EleutherAI 2023: weights at 8 sizes, the Pile as training data in exact training order, training code, and 154 intermediate checkpoints per model, under Apache 2.0. Built for reproducible research on training dynamics, and still the reference example. OLMo, Ai2, February 2024, then OLMo 2 in January 2025 and OLMo 3 in November 2025: weights, full training data, training code, evaluation harness, logs and intermediate checkpoints from every stage. OLMo 3’s release included Dolma 3, a 9.3-trillion-token corpus, the 5.9 trillion token pretraining mix used for the 7B and 32B base models, the post-training data suite, and the data tooling. BLOOM, BigScience 2022: 176B parameters, the ROOTS corpus, the training code, and a detailed record of the process, released under a RAIL licence that carries use restrictions and therefore is not OSI open source.
7. Open-weight, closed-data, permissive licence, which is the commercially dominant pattern: Mistral 7B (September 2023, Apache 2.0), Qwen3 (2025, Apache 2.0), DeepSeek-R1 (January 2025, MIT), and OpenAI’s gpt-oss-120b and gpt-oss-20b (5 August 2025, Apache 2.0).
8. gpt-oss is worth a note because it shows the pattern clearly. Apache 2.0 on the weights, plus published inference reference code, the harmony prompt renderer and the o200k_harmony tokenizer. No training data, no training code. Permissive licence, closed recipe.
9. Established fact: open weights allow local running, fine-tuning, adapter training, quantization, red-teaming and independent measurement. Established fact: they do not allow reproduction, data auditing, or verification of any claim about what the model was trained on. Marketing claim: describing a weights-only release under a restricted licence as “open source”. This is the specific abuse the reader asked about.
10. One practical consequence for engineers. Open weights are the only category that survives a vendor withdrawing a model. A downloaded file keeps working when an API endpoint is deprecated, and API deprecation happens on a twelve-to-twenty-four-month cycle at every major provider.

■ 49.12.6 WORDS – remember these

1. Open weights — the numbers are downloadable — parameters published for use, independent of licence, data or code availability.
2. Open source — a licence granting the four freedoms without discrimination — as defined by the Open Source Initiative's Open Source Definition since 1998.
3. Open data — the training corpus is published — the actual documents, or a reproducible recipe for assembling them, under a usable licence.
4. Permissive licence — almost no conditions — Apache 2.0 or MIT, allowing commercial use, modification and redistribution.
5. Community licence — a contract with conditions attached — Meta's and Google's custom terms, with user thresholds, naming rules and use policies.
6. OpenRAIL — a licence that restricts uses by design — the Responsible AI Licence family, openly not open source and honest about it.
7. Open washing — calling a restricted release open — marketing a weights-only or use-restricted model as open source.

49.13 The Open Source Initiative's position

■ 49.13.1 PLAIN – in simple words

1. The Open Source Initiative is the body that has maintained the definition of open source software since 1998, and it decides which licences qualify.
2. For twenty-five years that definition was about source code. Models are not source code, so it did not obviously apply.
3. So the OSI ran a public process with many contributors and published the **Open Source AI Definition version 1.0** on 28 October 2024.
4. It starts from the same four freedoms as software: use the system for any purpose, study how it works, modify it, and share it, modified or not.
5. To exercise those freedoms you need what it calls the preferred form to make modifications, and it says that means three things must be available.
6. **Data information:** enough detail about the training data that a skilled person could build a substantially equivalent system.
7. **Code:** the complete source code used to train and run the system, under an OSI-approved licence.
8. **Parameters:** the weights and other settings, under OSI-approved terms.
9. Read point 6 carefully, because it is the compromise at the heart of the whole thing. It does not require publishing the data itself. It requires publishing enough information about it.
10. By this definition, most models marketed as open source do not qualify.
11. The OSI has said so directly, naming names.

■ 49.13.2 PLAIN – a picture in your head

1. Think about a food label and a recipe again, one last time.
2. The strictest possible rule would be: hand over every ingredient you used, physically, so anyone can remake the dish exactly.
3. That is impossible for some ingredients. You cannot hand over a bottle of wine you drank, and you cannot hand over a book you licensed for one use.
4. So the OSI settled on: publish the recipe in enough detail that a competent cook could go and buy equivalent ingredients and make substantially the same dish.
5. That satisfies nobody entirely. Purists say a recipe is not the ingredients. Companies say even the recipe is a trade secret.
6. And it is precisely because that middle position annoys both sides that it is worth understanding exactly.

Where this comparison breaks: two cooks following the same recipe get similar results. Two labs following the same data description do not, because model outcomes depend on ordering, seeds, hardware numerics and hundreds of unpublished choices. “Substantially equivalent” is doing a great deal of work in that sentence and nobody has tested it in practice at frontier scale.

■ 49.13.3 PLAIN – a worked example

1. The OSI published its own assessment of real models against the definition. Its stated results, which it describes as part of the definitional process rather than as certifications:

Verdict	Models named by the OSI
Passes	Pythia, OLMO, T5
Passes	Amber, CrystalCoder
Would pass if relicensed	BLOOM, StarCoder2, Falcon
Does not pass	Llama 2, Grok, Phi-2, Mixtral

2. Note that Mixtral is in the failing list despite being under Apache 2.0. The licence is fine. The data information is missing. Both are required.
3. Note that BLOOM is in the middle list despite publishing its full training corpus. The data is fine. The RAIL licence restricts uses. Both are required.
4. Those two rows together are the clearest possible illustration that open weights, open data and an open licence are three independent things.
5. The OSI’s specific objections to Meta’s Llama licences: it fails freedom zero, the freedom to use the system for any purpose; it fails the Open Source Definition’s point 5 by discriminating against a class of user via the 700-million-user threshold; and it fails point 6 by restricting fields of endeavour through the Acceptable Use Policy. The OSI has also pointed to the exclusion of European Union users from certain models, unexplained.
6. The OSI uses the term “open washing” for marketing a restricted release as open source.

■ 49.13.4 PLAIN – what is really happening inside

1. Now the counter-arguments, given fairly, because this is a live dispute and not a settled question.
2. **Training data cannot always be redistributed.** Much of it is copyrighted text that the lab has a defensible claim to train on but no right to republish. Publishing it would be a separate and much larger legal exposure.
3. **Some of it must not be redistributed.** Web-scale corpora contain personal data. Publishing the corpus can conflict with privacy law even where training on it does not.

4. **Open weights deliver most of the practical benefit.** With weights you can run locally, fine-tune, quantize, audit behaviour, red-team, and avoid depending on a vendor. Very few people were ever going to retrain a frontier model even with the full data.
5. **Reproduction is impossible anyway at this scale.** Even with the corpus, rebuilding a frontier model costs tens of millions of dollars, so the freedom being protected is theoretical for almost everyone.
6. And there is a criticism from the opposite direction, which is important.
7. Some free-software advocates argued the OSI's compromise is too weak: that allowing "data information" instead of data lets companies claim the label without the substance, which is the very open washing the definition was meant to stop.
8. So version 1.0 is attacked from both sides: too strict for industry, too loose for purists. The OSI itself has said the definition will be revised.
9. The honest version: there is no consensus. The OSI has the strongest claim to define "open source" because it has defined it for software since 1998, but no legal authority, and companies are free to ignore it and do.

■ 49.13.5 TECHNICAL – the engineer's version

1. What the definition requires in full, as three components. Data information: complete descriptions of all data used, including provenance, scope and characteristics, how it was obtained and selected, labelling procedures, and cleaning and filtering methodologies, in enough detail that a skilled person can build a substantially equivalent system. Code: the complete source used for data processing and filtering, training, with all training arguments, validation, testing and inference, under OSI-approved licences. Parameters: model weights and other configuration settings, including intermediate checkpoints and final optimizer states, under OSI-approved terms.
2. That third clause is stricter than people notice. Intermediate checkpoints and optimizer state are almost never published. Pythia and OLMo do publish checkpoints, which is exactly why they pass.
3. Timeline of the dispute, dated: 1998, the Open Source Definition is published. 2022, RAIL and OpenRAIL licences appear, explicitly use-restricted. February 2023, Meta releases LLaMA 1 weights to researchers on request. July 2023, Llama 2 is released under a community licence and widely described in press coverage as open source. 2023 to 2024, the OSI runs a co-design process with contributors from industry, academia and civil society. 28 October 2024, Open Source AI Definition 1.0 is published at the All Things Open conference. Through 2025 and 2026, the argument continues without resolution, with further drafts under discussion.
4. Parallel regulatory pressure, which may matter more than the definitional argument in the end. The European Union's AI Act, in force from 1 August 2024 with general-purpose model obligations applying from 2 August 2025, requires providers to publish a sufficiently detailed summary of training content and to have a copyright policy. It also provides partial exemptions for models released under free and open-source licences, which gives the definition of "open" direct legal consequences.
5. So a term that was a marketing argument in 2023 became a compliance question in 2025. That is the practical reason to get it right.
6. Established fact: the OSAID 1.0 text, its date, its three required components, and the OSI's published assessments of named models. Active dispute: whether the data-information compromise is correct, whether version 1.0 will hold, and whether regulators will adopt it. Marketing claim: any company statement that its weights-only release "is open source" without addressing data information or licence restrictions.

■ 49.13.6 WORDS – remember these

1. OSAID — the open source rulebook for AI — the Open Source AI Definition 1.0, published by the OSI on 28 October 2024.
2. Data information — a description good enough to rebuild from — sufficiently detailed data documentation to build a substantially equivalent system.
3. Four freedoms — use, study, modify, share — the freedoms the definition requires an open source AI system to grant without restriction.
4. OSI-approved licence — a licence on the OSI’s published list — the test a licence must pass to be called open source.
5. Open washing — claiming openness you did not deliver — marketing a restricted or weights-only release using the open source label.

49.14 Model releases in practice

■ 49.14.1 PLAIN – in simple words

1. When a model is released, the file is the smallest part of the event.
2. Around it sit documents, gates and processes, and knowing what they should contain lets you judge a release in a few minutes.
3. The **model card** is the main document. The idea comes from a 2019 paper by Mitchell and colleagues, and it is now near-universal practice.
4. It should say: what the model is for, what it is not for, what it was trained on in general terms, how it scores on named benchmarks, what its known limitations are, and what licence applies.
5. The **system card** is a bigger document about the deployed product rather than the raw model: what safety testing was done, what was found, what mitigations were added, and what risks remain.
6. **Staged release** means publishing in steps rather than all at once, to watch what happens.
7. **Red-teaming** means paying people to attack the model before release and trying to make it do what it should not.
8. **Watermarking** means trying to mark generated output so it can be detected later. It works better for images than for text.
9. And a **licence gate** on a hosting site means you must accept terms, and often give your name and email, before you can download.
10. What is missing from a release tells you as much as what is present.

■ 49.14.2 PLAIN – a picture in your head

1. Think of the paperwork that comes with a new medicine.
2. There is a leaflet saying what it treats, the dose, who must not take it, and the side effects observed in trials. That is the model card.
3. There is a much longer regulatory dossier describing every trial, its results, and the risks that remain. That is the system card.
4. There is a phased rollout, first to a limited group, then wider. That is the staged release.
5. And there is a pharmacist who checks who you are before handing it over. That is the licence gate.
6. Now imagine a medicine arriving with no leaflet at all, and ask yourself what that absence tells you.

Where this comparison breaks: medicines are regulated by law with mandatory content and legal consequences for omissions. Model cards are, in most places and as of mid-2026, entirely voluntary

in content and quality. The European Union’s AI Act is the first serious attempt to make parts of this mandatory, and it applies to general-purpose models only from 2 August 2025.

■ 49.14.3 PLAIN – a worked example

1. The checklist to run against any release. Present or absent, and what the absence means.

Item	If missing, that means
Named training-data sources	You cannot audit or trust claims
Knowledge cutoff date	You cannot judge staleness
Benchmark results with names	Comparisons are unverifiable
Stated out-of-scope uses	No thought given to misuse
Compute and energy figures	Cost and impact unassessable
Licence text, not a summary	Read the actual terms first
Evaluation of known weaknesses	Only wins are being reported

2. Real examples of each end of the spectrum. Meta’s Llama 3.1 model card gives GPU-hours, energy, carbon, knowledge cutoff of December 2023, benchmark tables and licence, but describes training data only as “publicly available sources” with category percentages.
3. Ai2’s OLMo releases give all of the above plus the corpus itself, the data tooling, the training logs and intermediate checkpoints.
4. Closed model releases typically give benchmark tables and a system card, and no training data information at all.

■ 49.14.4 PLAIN – what is really happening inside

1. What happens in the weeks before a release, roughly in order.
2. Internal evaluation on held-out benchmarks, plus contamination checks to confirm the benchmarks were not in the training data.
3. Capability evaluations against a published safety framework, testing for dangerous capabilities in specific named categories.
4. Red-teaming, both internal and by hired external groups, attempting jailbreaks and misuse. Findings feed back into more safety fine-tuning.
5. Sometimes external review, by a government safety institute or a contracted evaluation organization, before release.
6. Then a decision, then the documents, then the gate, then publication.
7. **Staged release** has a clear historical example. In February 2019 OpenAI announced GPT-2 and released only the 124-million-parameter version, citing misuse concerns. It released 355M in May, 774M in August, and the full 1.5B model in November 2019, publishing its reasoning at each step.
8. That decision was heavily criticized at the time as overcautious, and it set the template that every lab now follows in some form.
9. Note what staged release cannot do for open weights. Once the file is downloaded it cannot be recalled, and as section 49.5 noted, the safety training can be removed by fine-tuning for a few hundred dollars.

■ 49.14.5 TECHNICAL – the engineer’s version

1. Model cards: proposed in “Model Cards for Model Reporting”, Mitchell and colleagues, 2019. The companion idea for data is “Datasheets for Datasets”, Gebru and colleagues, 2018. Both are now conventions rather than standards, though Hugging Face enforces a minimal card structure on its hub.

2. System cards: OpenAI published the GPT-4 System Card in March 2023, separating deployment risk analysis from the technical report. Anthropic and Google now publish equivalents for major releases. Contents typically include red-team findings, capability evaluations against a framework, refusal rates, and known failure modes.
3. Named safety frameworks you should recognize, with dates: Anthropic’s Responsible Scaling Policy, September 2023; OpenAI’s Preparedness Framework, December 2023; Google DeepMind’s Frontier Safety Framework, May 2024. All define capability thresholds that trigger additional controls. All are voluntary commitments, not regulation.
4. Government bodies: the UK AI Safety Institute was announced in November 2023 and renamed the AI Security Institute in February 2025. The US AI Safety Institute was established in 2023 and renamed the Center for AI Standards and Innovation in June 2025. Both have run pre-deployment evaluations under agreements with major labs.
5. Watermarking. Google DeepMind’s SynthID for images was announced in August 2023, and SynthID-Text was published in the journal Nature in October 2024, using a tournament sampling scheme that biases token choice in a detectable pattern. Google reported it does not measurably degrade quality.
6. The honest position on text watermarking: it is detectable while the text is unedited, and it degrades under paraphrasing, translation or moderate editing. It also requires the generator to co-operate, so it does nothing about open-weight models where the sampler can simply be replaced. Treat claims of robust text watermarking as marketing.
7. The C2PA standard, from a coalition including Adobe and Microsoft, takes the other approach: cryptographically signed provenance metadata attached to a file. It survives editing by signed tools and is destroyed by screenshots.
8. Licence acceptance gates: Hugging Face supports gated repositories, where a user must accept terms and often supply contact details, with optional manual approval. Meta and Google both use them. The gate is a contractual mechanism, not a technical one; it does not prevent redistribution.
9. Regulatory contents, dated. Under the European Union’s AI Act, providers of general-purpose models have obligations from 2 August 2025 including technical documentation, information for downstream providers, a copyright policy, and a sufficiently detailed public summary of training content, with partial exemptions for free and open-source releases and additional duties above the 10-to-the-25-FLOPs systemic-risk threshold.
10. Practical advice: read the licence file itself, not the summary on the model page; check the knowledge cutoff before relying on recency; check whether the card names benchmarks or just claims performance; and check whether evaluation numbers are self-reported or independently reproduced.

■ 49.14.6 WORDS – remember these

1. Model card — the leaflet that comes with a model — a structured document of intended use, training description, evaluations, limitations and licence.
2. System card — the safety dossier for the deployed product — red-team findings, capability evaluations, mitigations and residual risks.
3. Staged release — publishing in steps to watch the effect — incremental release of larger checkpoints or wider access over time.
4. Red-teaming — paying people to attack it before release — structured adversarial testing for jail-breaks, misuse and dangerous capability.
5. Watermarking — marking generated output so it can be detected — biased sampling for text, or signed provenance metadata for media.

6. Gated repository — you must accept terms to download — a hosting-site mechanism collecting acceptance and often identity before access.

49.98 Common wrong ideas

1. Wrong: the model was trained on the internet, so it knows everything. Right: it was trained on a heavily filtered slice of a web crawl, plus books, code and papers, and typically 80 to 95 per cent of the crawl was discarded. It has a knowledge cutoff date, it never saw most paywalled or private material, and what it did see it stored lossily as statistical patterns rather than as retrievable text.
2. Wrong: fine-tuning teaches it new facts. Right: fine-tuning teaches format, style and behaviour. Knowledge comes from pretraining because knowledge comes from data volume, and a fine-tuning set is perhaps one thirty-thousandth the size of the pretraining set. Fine-tuning on unfamiliar facts is learned slowly and measurably increases hallucination elsewhere. Use retrieval for facts.
3. Wrong: open weights means open source. Right: they are separate. Open weights means the numbers are downloadable. Open source, by the Open Source Initiative's definition of October 2024, additionally requires the training code, sufficiently detailed data information, and a licence with no discrimination against users or fields of use. Mixtral is Apache 2.0 and fails; BLOOM publishes its corpus and fails.
4. Wrong: RLHF makes the model truthful. Right: it makes the model produce answers that human raters prefer. Where raters prefer confident, agreeable, long or well-formatted answers, that is what gets reinforced. This is why sycophancy and length inflation are standard failure modes, and why OpenAI had to roll back a GPT-4o update in April 2025 for exactly this reason.
5. Wrong: training and running cost about the same. Right: they differ by roughly thirteen orders of magnitude per query. Training an 8-billion model on 15 trillion tokens is about $7.2e23$ FLOPs. Generating one token from it is about $1.6e10$ FLOPs. Training equals roughly 45 trillion generated tokens. For a busy service, cumulative inference overtakes training cost within months.
6. Wrong: a bigger model is always better. Right: Chinchilla in 2022 showed a 70-billion model beating a 280-billion one at equal compute because it saw four times more data. Data quantity, data quality and post-training all matter as much as parameter count.
7. Wrong: labs publish what they trained on. Right: GPT-3 published a mixture table in 2020. Almost nobody does now. GPT-4's technical report explicitly declined. Any claim about a closed model's training data, including a claim about what it did not include, is unverifiable from outside.
8. Wrong: the safety training is part of the model and cannot be removed. Right: for open weights it can be substantially removed by fine-tuning on a few hundred adversarial examples, for a few dollars. Safety behaviour is a thin post-training layer over an unchanged base capability.
9. Wrong: a training run is one program on one big computer. Right: it is thousands of accelerators running in lockstep, split three ways across data, tensors and layers, and hardware fails constantly. Meta reported 419 unexpected interruptions in 54 days on one run.
10. Wrong: the weights file contains the training data somewhere inside it. Right: it contains only numbers arranged in named tensors, plus a config and a tokenizer. Verbatim regurgitation of memorized training text does happen, especially for text duplicated many times in the corpus, but the data is not stored or retrievable in any ordinary sense.

49.99 Chapter summary in 20 lines

1. A model is made in a chain: collect data, clean it, train a tokenizer, pretrain, post-train, evaluate, release. Pretraining is 95 to 99 per cent of the compute and post-training is almost all of the visible behaviour.
2. Data comes from web crawls such as Common Crawl, which published 2.42 billion pages and 419 TiB uncompressed in its July 2025 archive alone, plus books, code, papers, licensed corpora and synthetic text.
3. Cleaning discards most of it: URL blocking, text extraction, language identification, exact and MinHash fuzzy deduplication, heuristic and classifier quality filters, toxicity and PII removal, and benchmark decontamination, then deliberate mixing ratios over the survivors.
4. FineWeb's published pipeline uses 5-gram shingles with 14 bands of 8 hashes, which catches near-duplicates above about 0.75 Jaccard similarity 77 per cent of the time and above 0.9 essentially always.
5. Copyright is unsettled and actively litigated. Judge Alsup held in June 2025 that training on lawfully bought books was fair use while retaining pirated copies was not, and Anthropic settled for a reported 1.5 billion dollars.
6. Most labs stopped publishing their data mixture after GPT-3 in 2020, so claims about a closed model's training data are unverifiable from outside.
7. Pretraining has exactly one objective: next-token prediction, scored by cross-entropy loss in nats, starting at the natural log of the vocabulary size and typically ending between 1.4 and 2.0.
8. One step is a forward pass over a batch of millions of tokens, a loss, a backward pass costing about twice the forward, and an AdamW update. Llama 3 405B used a peak learning rate of $8e-5$, 8,000 warmup steps and cosine decay to $8e-7$ over 1.2 million steps.
9. Llama 3.1 405B is the best-documented real run: over 16,384 H100s, 30.84 million GPU-hours, about 15.6 trillion tokens, roughly 78 days of wall clock and an estimated 62 to 123 million dollars of rented compute.
10. Training compute is approximately $6ND$, parameters times tokens times six. It reproduces GPT-3's reported $3.14e23$ and Llama 3.1 405B's $3.8e25$, and the European Union's AI Act uses 10 to the 25 as a systemic-risk threshold.
11. Kaplan and colleagues in January 2020 found smooth power laws and advised spending on size; Hoffmann and colleagues in March 2022 showed models were badly undertrained and that about 20 tokens per parameter is optimal.
12. Since 2023 labs deliberately overtrain small models far past that, because inference cost dominates over a deployed model's life. Llama 3.1 8B was trained at about 1,875 tokens per parameter, roughly 94 times Chinchilla.
13. A base model continues text and cannot be used as a chatbot: no turns, no roles, no stopping rule and no refusal behaviour. Ask it a question and it will often produce more questions.
14. SFT fixes that with tens of thousands to a few million demonstration pairs, a chat template of special tokens such as Llama 3's `<|eot_id|>`, loss masked to the assistant's tokens, and a learning rate ten times lower.
15. Preference learning goes further: collect comparisons, train a reward model, optimize with PPO under a KL penalty. Its characteristic failure is reward hacking, exemplified by OpenAI's sycophantic GPT-4o update of April 2025.
16. DPO in May 2023 removed the separate reward model algebraically and most smaller teams moved to it; constitutional and RLAIF methods replace human labels with written principles and an AI judge; RLVR uses checkable answers in maths and code and is the engine behind reasoning models.

17. Post-training shapes behaviour and cannot efficiently install knowledge, because the data is thirty thousand times smaller and the learning rate is set gently on purpose. The boundary is real but fuzzy, and mid-training blurs it deliberately.
18. LoRA freezes the base and trains thin low-rank matrices, cutting a 7B fine-tune from about 112 GB to about 17 GB; QLoRA adds a 4-bit base and cuts it to about 7 GB, which put a 65B fine-tune on one 48 GB card in 2023.
19. Weights are a binary file of numbers plus a config and a tokenizer. safetensors is an 8-byte length, a JSON header of dtypes, shapes and offsets, then raw bytes, and it exists because pickle-based .bin files can execute arbitrary code on load; GGUF bundles quantization and tokenizer.
20. Open weights, open source, open data and licence terms are four independent things. The Open Source AI Definition 1.0 of 28 October 2024 requires code, parameters and data information sufficient to rebuild a substantially equivalent system, and by it most models marketed as open source, including Llama, do not qualify. That remains a live dispute, not a settled question.

Benchmarks, Evaluation, Versions and Dates

50.0 What this chapter gives you

1. You will be able to say exactly what a benchmark is, in five mechanical parts, and name which part is being argued about when two numbers disagree.
2. You will be able to describe the main benchmarks by name, year, size and scoring rule, and say for each one what it does not measure.
3. You will be able to explain why the same model, on the same benchmark, run by two honest teams, can legitimately produce different scores.
4. You will be able to compute pass@k by hand and explain why it is reported instead of a plain percentage.
5. You will be able to explain benchmark contamination, list five ways people try to detect it, and say honestly why detection is hard and disputed.
6. You will be able to read a model announcement and list the eight things it did not tell you.
7. You will be able to explain the difference between a training cutoff date and a release date, and why models are bad at reporting their own cutoff.
8. You will be able to explain why a product name and a model identifier are different things, and why pinning a dated identifier matters in production.
9. You will be able to build your own evaluation set of 30 to 100 examples with a scoring rule, and use it to choose and re-choose a model.
10. Throughout, you will be able to sort a claim into three bins: established fact, active research, and marketing claim.

A warning that applies to this whole chapter. The dates, item counts, paper titles and naming conventions here were checked in August 2026. The item counts and the naming conventions are stable. The leaderboard numbers are not. Any specific score in this chapter should be read as “roughly this, in mid-2026”, and any score you care about should be re-checked at the source.

50.1 What a benchmark is, mechanically

■ 50.1.1 PLAIN – in simple words

1. A benchmark is a test. That is all it is.
2. It has a fixed list of questions.
3. Somebody already knows the right answer to every question.
4. There is a rule for deciding if a given answer counts as right.

5. You give the questions to the model, collect its answers, apply the rule, and count.
6. The count, turned into a percentage, is the score.
7. There is nothing magic in it. It is a school test with a marking scheme.
8. It is not a measurement of intelligence, any more than a spelling test is.
9. It measures one thing: how many of these particular items, marked this particular way, this particular model got right on this particular day.
10. Everything difficult about benchmarks comes from the words “particular”.
11. Change the wording of the question and the score moves.
12. Change the marking rule and the score moves.
13. Change which items you use and the score moves.
14. None of that is cheating. It is just what happens.
15. So when two labs publish different numbers for the same model on the same test, the first assumption should not be dishonesty. It should be setup.

■ 50.1.2 PLAIN – a picture in your head

1. Think of a driving test.
2. The test has a fixed route, a fixed set of manoeuvres, and an examiner with a clipboard listing faults.
3. Two candidates take the same test and get different results. Fine.
4. Now take the same candidate to two different test centres.
5. One centre uses a route with a busy roundabout. The other uses a quiet estate. The candidate may pass one and fail the other.
6. One examiner counts a small kerb touch as a minor fault. Another counts it as a major. Same driving, different result.
7. One centre lets you do a warm-up lap first. The other does not.
8. Now imagine the candidate has driven that exact route four hundred times, because it is published online.
9. That last one is contamination, and section 50.5 is about it.
10. And here is the biggest problem. Passing a driving test does not tell you whether somebody can drive to Manchester in fog at night.

Where this comparison breaks: a driving test has a legal standard behind it and an examiner who watched you. Most AI benchmarks have neither. Nobody watched the model. A script compared strings. Also, driving tests are not published in advance with all the answers. Almost every AI benchmark is.

■ 50.1.3 PLAIN – a worked example

1. Here is one real item, in the style of GSM8K, a set of grade-school maths word problems released by OpenAI in 2021.

Question:

Natalia sold clips to 48 friends in April, and then she sold half as many clips in May. How many clips did she sell altogether in April and May?

Reference answer (the dataset marks the final number with four hash characters):

$48 / 2 = 24$

$48 + 24 = 72$

72

- The #### marker is part of the dataset format. The number after it is the answer the scorer will compare against.
- Now here are two different things a model might produce.

Model A output:

"She sold 48 in April and 24 in May, so 72 clips."

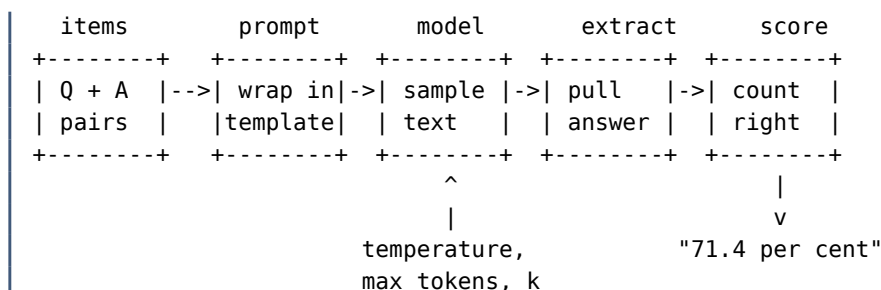
Model B output:

"72"

- Both are correct. A strict scorer that demands the whole reply to equal "72" marks model A wrong and model B right.
- A scorer that pulls the last number out of the reply marks both right.
- Nothing about the model changed. The marking rule changed.
- That single choice can move a reported score by several points across a whole benchmark.
- Now a nastier case. A model replies "72 clips". A scorer that pulls the last number finds 72. Good.
- A model replies "The answer is $48 + 24 = 72$." A scorer that pulls the last number still finds 72. Good.
- A model replies "She sold 72, up from 48." A scorer that pulls the last number finds 48 and marks it wrong. The model was right.
- Real answer-extraction bugs look exactly like this.

■ 50.1.4 PLAIN – what is really happening inside

- Every benchmark run has exactly five moving parts. Learn these five and you can interrogate any published number.
- Part one, the **items**. The fixed list of questions and known answers.
- Part two, the **prompt template**. The wrapper of text that surrounds each item before it reaches the model. Instructions, examples, formatting.
- Part three, the **generation settings**. How the model is asked to produce text: temperature, how many tokens it may write, whether it may think first.
- Part four, the **answer extraction and scoring function**. The code that turns a blob of model text into "right" or "wrong".
- Part five, the **aggregate**. How the per-item results are combined into one number, and whether that number is one run or an average of several.
- Here is the flow as a picture.



- Each of the five boxes is a choice made by whoever ran the evaluation.
- Every one of those five choices changes the number that comes out.
- Only the first box, the items, is usually published. The other four are often described in a sentence or not at all.
- That is the single most important fact in this chapter.

12. The honest version: the sentence “this model scores 88 per cent on MMLU” is incomplete in the way “this car does 60” is incomplete. A benchmark score is not a property of a model. It is a property of a model plus a harness, measured on one day.

■ 50.1.5 TECHNICAL – the engineer’s version

1. Formally, an evaluation is a tuple: a dataset D of items, a prompt function p that maps an item to a model input, a decoding configuration, an extraction function e that maps model output to a candidate answer, a metric m that compares candidate to reference, and an aggregation over D .
2. The score is $(1/|D|) * \sum \text{over } i \text{ of } m(e(\text{model}(p(d_i))), a_i)$ for accuracy-style metrics. Everything contested lives in p , e and the decoder.
3. Two implementation families exist for multiple-choice items and they do not give the same answer.
4. **Log-likelihood scoring**: present the question, then score each candidate continuation by the model’s log probability, and pick the highest. The model never generates a token. Common for base models.
5. **Generative scoring**: present the question with lettered options, let the model emit text, then parse a letter out of it. Common for chat models.
6. On the same items and the same model these can differ by several points. Neither is wrong. They measure different things.
7. Length normalization is a further fork. Raw log probability favours short options, so harnesses often divide by token count or by character count, or normalize against the unconditional probability of the option.
8. The EleutherAI **lm-evaluation-harness** is the de facto open standard for this. It supports over 60 academic benchmarks with hundreds of subtask variants, and it powered the Hugging Face Open LLM Leaderboard. Its stated principle is that publicly available prompts make results reproducible and comparable between papers. This is a convention, not a standard.
9. Stanford CRFM’s **HELM**, first published in November 2022 as “Holistic Evaluation of Language Models”, takes the other approach: fix the prompts centrally, report many metrics per scenario rather than one, and publish every raw model output.
10. Terminology: a **harness** is the code that runs the loop. A **scenario** or **task** is one benchmark inside it. A **run** is one execution.
11. There is no ISO or IEEE specification for any of this. There is no conformance test. Everything in this section is convention.

Part of a run	Who usually sets it	Usually published?
Items	Benchmark author	Yes
Prompt template	Evaluator	Sometimes
Decoding settings	Evaluator	Rarely
Extraction and metric	Evaluator	Rarely

■ 50.1.6 WORDS – remember these

1. Benchmark — a fixed test with known answers — a dataset plus a prompting, extraction and scoring protocol.
2. Item — one question — one example in the evaluation dataset, with a reference answer.
3. Prompt template — the wrapper text around a question — the function mapping a dataset item to model input, including any few-shot exemplars.
4. Harness — the program that runs the test — the evaluation framework that implements prompting, decoding, extraction and aggregation.

5. Extraction — pulling the answer out of the reply — the parser mapping raw generation to a comparable candidate answer.
6. Metric — the marking rule — the function comparing candidate to reference, such as exact match or pass@k.
7. Aggregate — the final single number — the reduction over all items, often plain accuracy, sometimes a macro average over subjects.
8. Log-likelihood scoring — asking which option the model thinks most likely — ranking candidate continuations by model log probability without generating.

50.2 The major benchmarks, one at a time

■ 50.2.1 PLAIN – in simple words

1. There are perhaps twenty benchmark names you will see repeatedly.
2. They fall into a few families.
3. **Knowledge quizzes:** multiple-choice questions across many school and university subjects. MMLU is the famous one.
4. **Maths:** word problems and competition problems. GSM8K, MATH, AIME.
5. **Code:** write a function that passes hidden tests. HumanEval, MBPP.
6. **Hard science:** questions a PhD student would struggle with. GPQA.
7. **Agentic work:** give the model a real repository and a real bug report and see whether the tests pass afterwards. SWE-bench.
8. **Reasoning puzzles:** coloured grid puzzles with no text at all. ARC-AGI.
9. **Common sense:** pick the sensible ending to a sentence. HellaSwag.
10. **Instruction following:** did it obey the format you asked for? IFEval.
11. **Chat quality:** judged by a human or by another model. MT-Bench.
12. Each of these was built for a reason, at a time, by somebody.
13. Almost all of them are now much easier than when they were built.
14. That is not because they were badly made. It is because the models improved and because the tests leaked into training data.
15. The single most useful habit is to ask, for each benchmark: what would a model have to be able to do to score well, and is that what I need?

■ 50.2.2 PLAIN – a picture in your head

1. Think of a hospital's set of blood tests.
2. Each test measures one narrow thing. Iron. Glucose. White cell count.
3. No single test tells you whether a person is healthy.
4. A doctor orders several, reads them together, and knows which ones are unreliable in which circumstances.
5. A benchmark suite works the same way. MMLU is a bit like a general inflammation marker: broad, cheap, and non-specific.
6. SWE-bench is like a treadmill stress test: expensive, slow, closer to the real activity, and it can be gamed by training on the treadmill.
7. And here is the part that matters. Some blood tests go out of date. A test that was diagnostic in 1990 may be useless now because everyone takes a supplement that changes the reading.
8. Benchmarks go out of date the same way, and for a similar reason: everybody now optimizes for the reading.

Where this comparison breaks: blood tests have reference ranges established by regulators and re-

producibile physical chemistry behind them. AI benchmarks have neither. Two labs running the same blood test on the same sample should agree within a known tolerance. Two labs running the same AI benchmark on the same model have no agreed tolerance at all.

■ 50.2.3 PLAIN – a worked example

1. Take one benchmark and follow it end to end: GPQA.
2. GPQA stands for Graduate-level Google-Proof Question Answering.
3. It was published on 20 November 2023 by David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael and Samuel Bowman.
4. It has 448 multiple-choice questions in biology, physics and chemistry.
5. Every question was written by a person with or studying for a PhD in that subject.
6. “Google-proof” is a design goal, not a guarantee. It means the writers tried hard to make the answer un-findable by search.
7. Here is the evidence they gave for that. Skilled non-experts, allowed unrestricted web access and spending on average over 30 minutes per question, reached 34 per cent accuracy.
8. Experts in the matching field reached 65 per cent, or 74 per cent after correcting for questions the experts later agreed were flawed.
9. Chance is 25 per cent, because there are four options.
10. The best model at publication, based on GPT-4, scored 39 per cent.
11. **GPQA Diamond** is a 198-question subset. It keeps only questions where the expert validators agreed on the answer and most non-experts got it wrong. It is the hardest and cleanest slice.
12. Diamond is what almost every model announcement reports. When you see “GPQA”, check whether it means the 448 or the 198.
13. What GPQA measures well: whether a model holds deep, specific technical knowledge that is not one search away.
14. What it does not measure: whether the model can do science. There is no experiment, no hypothesis, no error bar, no lab. It is a multiple-choice quiz with four options and a one-in-four floor.
15. And by 2026 the top scores on Diamond had risen far above the 65 per cent expert figure, which does not mean the models are better scientists than PhD holders. It means they are better at that quiz.

■ 50.2.4 PLAIN – what is really happening inside

1. Here is each major benchmark in one block: what it is, when, how big, how scored, and its main blind spot.
2. **MMLU**, 2020. “Measuring Massive Multitask Language Understanding”, by Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song and Jacob Steinhardt, first posted 7 September 2020.
3. 57 subjects, from elementary mathematics to professional law. Four options per question. About 14,000 test items in the widely used release, plus a 1,531-item validation split and a 285-item development split used for few-shot examples.
4. Scored as plain accuracy. Chance is 25 per cent.
5. Blind spot: it is recall of facts in multiple-choice form. It cannot tell you whether a model can write, reason for ten steps, use a tool, or admit ignorance. It also contains a known number of wrong reference answers.
6. **MMLU-Pro**, 2024. From the TIGER-Lab group, first posted 3 June 2024. About 12,000 questions, ten options instead of four, 14 subject areas, harder items, and many of the broken MMLU items removed.

7. Scores dropped by 16 to 33 points relative to MMLU. Chance falls to 10 per cent. Sensitivity to prompt wording fell from about 4 to 5 points on MMLU to about 2 points on MMLU-Pro, across 24 prompt styles they tried.
8. **GSM8K**, 2021. From “Training Verifiers to Solve Math Word Problems” by Karl Cobbe and colleagues at OpenAI, 27 October 2021. 8.5 thousand grade-school word problems: 7,473 for training, 1,319 for testing.
9. Scored by exact match on the final number. Blind spot: the arithmetic is trivial and the reasoning is two to eight steps. It has been effectively solved since about 2024 and no longer separates strong models.
10. **MATH**, 2021. Hendrycks and colleagues, 5 March 2021. 12,500 competition problems with full worked solutions, split 7,500 train and 5,000 test. Answers are mathematical expressions, not just numbers, which makes marking much harder. A 500-item subset called MATH-500 is widely reported.
11. **HumanEval**, 2021. From “Evaluating Large Language Models Trained on Code” by Mark Chen, Jerry Tworek, Heewoo Jun and 51 others at OpenAI, 7 July 2021. Exactly 164 hand-written Python problems.
12. Each has a function signature, a docstring and hidden unit tests. Scored by pass@k: the chance that at least one of k samples passes all the tests. The original Codex model scored 28.8 per cent at pass@1 and 70.2 per cent at pass@100.
13. HumanEval blind spots, and there are many. 164 items is tiny, so one item is 0.6 points. The problems are self-contained functions of a few lines. There is no project, no dependency, no existing code to read, no debugging. The tests are shallow. It is Python only. And it is thoroughly leaked.
14. **MBPP**, 2021. “Program Synthesis with Large Language Models” by Jacob Austin, Augustus Odena and colleagues at Google, 16 August 2021. 974 entry-level Python problems, each with three assert-statement tests. A hand-cleaned subset of 427 problems is often used instead.
15. **GPQA**, 2023, covered above. 448 items, Diamond subset 198.
16. **SWE-bench**, 2023. “SWE-bench: Can Language Models Resolve Real-World GitHub Issues?” by Carlos Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press and Karthik Narasimhan at Princeton, 10 October 2023.
17. 2,294 tasks built from real merged pull requests in 12 popular Python repositories. The model gets the repository at the parent commit and the issue text. It must produce a patch. The patch is applied and the project’s own tests are run.
18. Scored as **resolve rate**: the percentage of tasks where the hidden tests that the real pull request made pass, do pass, and nothing that passed before breaks. This is a much stronger signal than string matching.
19. In the original paper the best model tested, Claude 2, resolved 1.96 per cent. By 2026 the reported figures on the Verified subset were far above 70 per cent. That is a genuine capability jump and also a heavily optimized target.
20. **SWE-bench Verified**, 13 August 2024, from OpenAI. 93 experienced Python developers reviewed 1,699 randomly drawn original tasks. Tasks were dropped if the issue text was underspecified or the tests were unfair. 500 were kept. OpenAI reported that 68.3 per cent of sampled original tasks were filtered out for such problems.
21. That is an important admission in both directions. The original benchmark understated model ability. And a benchmark curated by one lab is now the number everyone quotes.
22. Other official variants: SWE-bench Lite (300 tasks, cheaper), SWE-bench Multimodal (517 tasks with visual elements), SWE-bench Multilingual (300 tasks from 42 repositories in 9 languages), and a Bash-only view of the 500 Verified tasks.

23. **ARC-AGI**. Created by Francois Chollet and introduced in his 2019 paper “On the Measure of Intelligence”. The tasks are small coloured grids. You see two or three input-output examples and must produce the output for a new input. No text. No knowledge required.
24. The design goal is to measure skill acquisition on unseen problems rather than stored knowledge. ARC-AGI-2, launched in 2025, has 120 public evaluation tasks and 120 held-back semi-private tasks, and every task was solved by at least two humans in under two attempts, calibrated on over 400 members of the public. ARC-AGI-3 followed.
25. The ARC Prize leaderboard shows cost per task alongside score and by default excludes systems costing more than 10,000 dollars for a run. That is unusual and honest, and more benchmarks should copy it.
26. **HellaSwag**, 2019. Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi and Yejin Choi, 19 May 2019. Pick the sensible continuation of a short description from four options. About 10,000 items in the reported split.
27. The wrong options were produced by Adversarial Filtering: generate candidates by machine, keep the ones a discriminator model finds hard. Humans scored above 95 per cent; the best models of 2019 under 48 per cent. Today it is saturated and is mostly a sanity check on base models.
28. **BIG-bench**, 2022. “Beyond the Imitation Game”, posted 9 June 2022, with over 450 authors from over 132 institutions and 204 tasks. Deliberately weird, broad and long-tailed.
29. Every BIG-bench task file contains a **canary string**, a fixed random-looking identifier the authors ask people never to remove, so that anyone can test whether a model has seen the benchmark. That is a genuinely good idea, and section 50.5 explains what it can and cannot prove.
30. **BIG-Bench Hard**, 2022, is a 23-task subset where models at the time did worse than the average human rater. It is what people actually run, usually as “BBH”.
31. **IFEval**, 2023. Jeffrey Zhou and colleagues at Google, 14 November 2023. About 500 prompts carrying 25 kinds of machine-checkable instruction, such as “write more than 400 words” or “mention the keyword AI at least 3 times” or “reply in JSON” or “no capital letters”.
32. Its whole point is that no judgement is needed: a short Python function checks compliance. It measures obedience to format, not quality of content.
33. **MT-Bench**, 2023. From “Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena” by Lianmin Zheng, Wei-Lin Chiang, Ying Sheng and colleagues, 9 June 2023. 80 multi-turn questions in 8 categories, two turns each.
34. A strong model grades the answers on a 1 to 10 scale. The paper reported over 80 per cent agreement between the model judge and human raters, which is about the level humans agree with each other.
35. The same paper named the three biases of a model judge that you must remember: **position bias** (it prefers whichever answer came first), **verbosity bias** (it prefers the longer answer) and **self-enhancement bias** (it prefers text in its own style).
36. **AIME**. Not built for AI at all. The American Invitational Mathematics Examination is a competition run by the Mathematical Association of America: 15 questions, 3 hours, every answer an integer from 000 to 999, held twice a year.
37. That integer answer format is exactly why AI labs adopted it: marking is trivial and unambiguous. Its weaknesses are that 15 items means one question is 6.7 points, and that each year’s paper is public within hours and in training data within months.
38. **Humanity’s Last Exam**, launched in 2025 by the Center for AI Safety and Scale AI, published in the journal Nature on 28 January 2026. 2,500 questions across more than 100 subjects, written by close to a thousand expert contributors from over 500 institutions in 50 countries.
39. It was built precisely because MMLU had saturated. A private held-back portion is kept to detect

overfitting, and a rolling variant was released on 8 October 2025 to keep adding fresh items.

40. **FrontierMath**, from Epoch AI, is a set of unpublished research-level mathematics problems in difficulty tiers, plus a separate collection of genuinely unsolved open problems. As of 31 July 2026 that open set held 50 unsolved research problems, of which AI systems had contributed solutions to three.
41. **Terminal-Bench**, a Stanford collaboration, gives an agent a real terminal and real tasks in software engineering, machine learning, security and data science, scored by whether the task actually completed. Version 1.0 had 80 tasks, version 2.0 had 89, and later versions followed.

■ 50.2.5 TECHNICAL – the engineer’s version

1. The comparison table, split into four-column pieces so it prints. All item counts verified August 2026 against the dataset cards and papers.

Benchmark	Year	Items
MMLU	2020	~14,000 test
MMLU-Pro	2024	~12,000 test
GSM8K	2021	1,319 test
MATH	2021	5,000 test
HumanEval	2021	164
MBPP	2021	974 (427 clean)
GPQA	2023	448
GPQA Diamond	2023	198
SWE-bench	2023	2,294
SWE-bench Verified	2024	500
ARC-AGI-2	2025	120 + 120 held back
HellaSwag	2019	~10,000 val
BIG-bench	2022	204 tasks
BIG-Bench Hard	2022	23 tasks
IFEval	2023	~500 prompts
MT-Bench	2023	80 questions
AIME (one paper)	yearly	15
Humanity’s Last Exam	2025	2,500

2. Scoring rules and chance floors.

Benchmark	Metric	Chance floor
MMLU	Accuracy	25 per cent
MMLU-Pro	Accuracy	10 per cent
GSM8K	Exact match number	~0
MATH	Expression equivalence	~0
HumanEval	pass@k on unit tests	~0
GPQA Diamond	Accuracy	25 per cent
SWE-bench	Resolve rate	0
ARC-AGI-2	Exact grid match	~0
HellaSwag	Accuracy	25 per cent
IFEval	Constraint pass rate	varies
MT-Bench	Model-judge 1 to 10	n/a

3. The gap between what each measures and what it is claimed to measure. This column is the one people skip.

Benchmark	Claimed as	Actually measures
MMLU	General knowledge	4-way recall, one shot
GSM8K	Reasoning	Short arithmetic chains
HumanEval	Coding ability	Tiny isolated functions
GPQA Diamond	Science ability	Hard 4-way recall
SWE-bench	Autonomous engineer	Patch passes given tests
ARC-AGI	General intelligence	Grid puzzle induction
HellaSwag	Common sense	Adversarially filtered picks
MT-Bench	Chat quality	One model's taste
IFEval	Instruction following	Checkable format rules

4. Established fact: the item counts, dates, authors and scoring rules above. These are checkable in the papers and dataset cards, and they do not move.
5. Active research: whether any of these benchmarks predicts real-world task performance. The honest state of the field is that correlation exists but is weak and task-dependent, and there is no accepted transfer function from a benchmark score to a deployment outcome.
6. Marketing claim: any sentence of the form “scores X on benchmark Y, therefore it is a PhD-level scientist” or “therefore it can replace a junior engineer”. The benchmark supports the first clause only.
7. Tooling. `lm-eval --model hf --model_args pretrained=... --tasks mmlu --num_fewshot 5` runs MMLU under the EleutherAI harness. The SWE-bench project ships its own Docker-based runner, because reproducing per-repository environments is most of the difficulty. `bigcode-evaluation-harness` is the common runner for HumanEval and MBPP and executes generated code in a sandbox, which you should never skip.
8. Note on the honest version of “SWE-bench measures agentic ability”. The honest version: SWE-bench measures the combined ability of a model plus a scaffold plus a retrieval strategy plus a test-running loop. Two entries using the same model and different scaffolds can differ by more than 20 points. Section 50.4 returns to this.

■ 50.2.6 WORDS - remember these

1. Saturation — everyone scores near the top so the test stops sorting — the point where score variance is dominated by noise and label errors rather than capability differences.
2. Diamond subset — the hard, clean slice of GPQA — the 198 questions where expert validators agreed and most non-experts failed.
3. Resolve rate — how often the fix actually worked — the fraction of SWE-bench instances where the fail-to-pass tests pass and pass-to-pass tests still do.
4. Adversarial filtering — keeping only wrong answers that fool machines — an iterative data-collection loop where a discriminator selects distractors.
5. Canary string — a marker planted so leaks can be spotted — a fixed unique identifier embedded in benchmark files to detect web-scrape contamination.
6. Chance floor — the score you get by guessing — the expected accuracy of a uniform random policy, $1/n$ for n options.
7. Held-back set — questions kept secret — a private evaluation split withheld from publication to measure overfitting to the public split.

8. Scaffold — the harness of code around a model in agent tasks — the controller implementing retrieval, tool calls, retries and stopping rules.

50.3 Actually running an evaluation

■ 50.3.1 PLAIN – in simple words

1. Suppose you have the questions and the answers. How do you run the test?
2. First you must decide what text to actually send.
3. You could send only the question. That is called **zero-shot**.
4. You could send a few solved examples first, then the question. That is called **few-shot**, and five examples is called five-shot.
5. Few-shot examples do two jobs. They remind the model of the subject, and more importantly they show it the exact answer format you want.
6. Most of the benefit is the format, not the reminder.
7. Next you decide whether to let the model think out loud before answering. That is **chain-of-thought**.
8. On maths, letting it think out loud changes the score enormously. Not by a little. By tens of points.
9. Next you decide how random the model is allowed to be. That is the temperature setting.
10. Zero means always take the most likely next word. Higher means sometimes take a less likely one.
11. If temperature is above zero, the same model on the same question can give different answers on different days.
12. So the same evaluation run twice gives two different scores.
13. That is why careful teams run the whole thing several times and report the average, and ideally the spread.
14. Then you must decide how to mark the reply, which is the hardest part.
15. For a single number, compare numbers. For a letter, find the letter. For an essay, you need a human or another model, and both are unreliable.

■ 50.3.2 PLAIN – a picture in your head

1. Think of a school exam being marked by three different people.
2. The first marker demands the answer in the box, in pen, and nothing else. Anything in the margin is ignored.
3. The second marker reads the whole page and looks for the answer wherever it is. More forgiving, but sometimes picks up a crossed-out number.
4. The third marker is an experienced teacher who reads the reasoning and awards a judgement out of ten.
5. The first marker is exact match. The second is regex extraction. The third is a model judge.
6. Now imagine the exam paper itself. One version says “Answer the question.” Another says “Answer the question. Show your working. Put your final answer after the words FINAL ANSWER.”
7. The second paper produces higher marks from the same pupils, because it told them the format the marker wants.
8. That is the whole of prompt template design, in one sentence.

Where this comparison breaks: a pupil who knows the answer but writes it in the margin still knows the answer, and a good teacher notices. A benchmark harness does not notice anything. It runs a regular expression. If the format is wrong the mark is zero, and the report will say the model does not know maths.

■ 50.3.3 PLAIN – a worked example

1. Let us compute pass@k properly, because it is the metric people most often misread.
2. pass@k is not “the score when you take k tries”. It is an estimate of the probability that at least one of k independently drawn samples is correct.
3. The naive way is to draw k samples and see if any pass. That is a very noisy estimate for large k.
4. The Codex paper of July 2021 uses a better estimator. Draw n samples, count c that pass, and compute:

$$\text{pass@k} = 1 - C(n - c, k) / C(n, k)$$

where C(a, b) is “a choose b”, the number of ways to pick b items from a, and $n \geq k$.

5. In words: one minus the chance that all k of your picks come from the failing pile.
6. Worked example. Take one problem. Draw $n = 10$ samples. Suppose $c = 2$ pass.
7. $\text{pass@1} = 1 - C(8,1)/C(10,1) = 1 - 8/10 = 0.20$.
8. $\text{pass@5} = 1 - C(8,5)/C(10,5) = 1 - 56/252 = 1 - 0.2222 = 0.7778$.
9. $\text{pass@10} = 1 - C(8,10)/C(10,10)$. $C(8,10)$ is 0 because you cannot pick 10 things from 8. So $\text{pass@10} = 1 - 0 = 1.00$.
10. Read those three numbers. One try: 20 per cent. Five tries: 78 per cent. Ten tries: 100 per cent.
11. The model did not get better. You bought more lottery tickets.
12. This is why pass@1 and pass@100 are not comparable, and why a headline that does not say which k it used is not a result.
13. Now the temperature effect, with real published behaviour. High temperature lowers pass@1 and raises pass@100, because diversity costs you on a single draw and pays on many draws.
14. So a lab reporting pass@1 will tune temperature low, and a lab reporting pass@100 will tune it high, and both are being reasonable.

■ 50.3.4 PLAIN – what is really happening inside

1. Take one MMLU item and watch it through the whole pipeline.
2. Zero-shot generative version. The harness builds this text:

The following is a multiple choice question about high school biology.

Which organelle produces most of a cell's ATP?

- A. Ribosome
- B. Mitochondrion
- C. Golgi apparatus
- D. Lysosome

Answer:

3. The model generates text. Suppose it writes “B. Mitochondrion”.
4. The extractor strips whitespace, takes the first letter that is A, B, C or D, and compares it to the stored key.
5. Now the log-likelihood version. The harness does not let the model write anything. Instead it asks the model, four times, “how likely is this exact continuation?”
6. It scores “A”, “B”, “C” and “D” as continuations, and takes the highest.
7. Same item, same model, two protocols, and sometimes two different answers.
8. A model that would have written a paragraph of hedging before saying B still assigns the highest single-token probability to “B”, and scores right under the second protocol and possibly wrong

under the first.

9. This is not a subtle effect. It is a common cause of a several-point gap between two published MMLU numbers.
10. Now the few-shot version. The harness prepends five solved questions from the 285-item development split before the real one.
11. The model now sees five examples of the pattern “question, options, Answer: B” and copies the pattern. Format failures collapse.
12. Now chain-of-thought. The template adds “Think step by step, then give your final answer as a single letter after the words FINAL ANSWER:”.
13. The model now emits a paragraph of reasoning and then the letter. The extractor looks after “FINAL ANSWER:” instead of at the start.
14. On knowledge questions this changes little. On maths it changes everything, because the model can hold intermediate results in the text it has already written rather than trying to do the whole calculation in one forward pass.
15. That last sentence is the honest mechanism. Chain-of-thought is not the model “thinking harder”. It is the model using its own output as working memory, one token at a time.

■ 50.3.5 TECHNICAL – the engineer’s version

1. Prompt sensitivity is measurable and large. The MMLU-Pro authors evaluated 24 prompt styles and found score variation of about 4 to 5 percentage points on original MMLU and about 2 points on MMLU-Pro. That is the size of the gap labs argue over in announcements.
2. Few-shot conventions by benchmark, as used in most reported numbers. These are conventions, not standards, and they drift.

Benchmark	Common shots	Common mode
MMLU	5-shot	Log-likelihood or letter
MMLU-Pro	5-shot CoT	Generative
GSM8K	8-shot CoT or 0-shot	Generative
MATH	4-shot CoT	Generative
HumanEval	0-shot	Generative, pass@k
GPQA Diamond	0-shot CoT	Generative
HellaSwag	10-shot	Log-likelihood
BBH	3-shot CoT	Generative

3. Chain-of-thought prompting comes from “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models” by Jason Wei and colleagues at Google, posted January 2022. On GSM8K with a 540-billion-parameter model it moved accuracy from roughly 18 per cent to roughly 57 per cent.
4. “Large Language Models are Zero-Shot Reasoners” by Takeshi Kojima and colleagues, May 2022, showed that the single sentence “Let’s think step by step” produced much of the same effect with no examples at all.
5. Established fact: chain-of-thought raises measured accuracy on multi-step arithmetic and symbolic tasks by a large margin. Active research: whether the written chain reflects the computation that actually determined the answer. There is published evidence that models sometimes produce a correct answer with an unfaithful chain, and that they can be steered to a wrong answer while the chain still reads plausibly.
6. Grading methods, in increasing order of flexibility and decreasing order of reliability.

Method	How it works	Main failure
Exact match	String equality	Punishes correct formats
Normalized match	Lowercase, strip, unify	Still brittle on units
Regex extraction	Pull last number or tag	Grabs the wrong number
Symbolic equivalence	Parse maths, compare	Parser gaps, timeouts
Unit tests	Execute the code	Weak or overfit tests
Model as judge	Ask a model to grade	Position, length, style bias

7. For MATH-style answers, plain string comparison is hopeless: $1/2$, 0.5 and $\frac{1}{2}$ are the same answer. Serious harnesses parse into a computer algebra system and test symbolic equality, with a timeout.
8. Model-as-judge biases are documented in the MT-Bench paper of June 2023: position bias, verbosity bias, self-enhancement bias, and limited reasoning on maths. Mitigations in common use are swapping the order of the two answers and averaging, calibrating with a reference answer, and using chain-of-thought in the judge prompt.
9. Sampling and reproducibility. With temperature 0 and greedy decoding, output is nominally deterministic. In practice it often is not.
10. The honest version: floating-point reduction order on a GPU depends on how work is split across threads, that split can depend on the batch size, and batch composition on a shared serving end-point depends on who else is sending requests. Mixture-of-experts routing can also depend on the batch. So the same prompt at temperature 0 can produce different tokens.
11. Therefore report mean and standard deviation over repeated runs. Common practice for small benchmarks such as AIME with 15 items is to run 16, 32 or 64 times and average, sometimes written “avg@32” or “mean@64”.
12. Distinguish that from **majority voting**, sometimes written “cons@64” or self-consistency, where you sample 64 answers and take the most common one. That is a different, stronger system, and it costs 64 times as much.
13. Distinguish both from **best-of-n**, where you sample n answers and pick the best using a verifier or the reference answer. If the reference answer is used to pick, the number is close to meaningless as a measure of what a user would get.
14. Note the 2026 wrinkle: some providers have begun removing sampling controls entirely. Anthropic’s documentation as of August 2026 lists temperature, top_p and top_k as deprecated for its newest models, returning an error if set to a non-default value. If you cannot set temperature, you cannot match another lab’s sampling configuration.

■ 50.3.6 WORDS – remember these

1. Zero-shot — no examples given — the item is presented with instructions only, no in-context exemplars.
2. Few-shot — a few solved examples first — k in-context exemplars prepended, usually drawn from a fixed development split.
3. Chain-of-thought — letting it show its working — eliciting intermediate reasoning tokens before the final answer, which act as external working memory.
4. Temperature — how random the wording is — the divisor applied to logits before the softmax; 0 gives greedy decoding.
5. pass@k — chance that one of k tries works — the unbiased estimator $1 - C(n-c,k)/C(n,k)$ over n samples with c correct.

6. Self-consistency — sample many, take the most common answer — majority voting over sampled chains, reported as cons@n.
7. Best-of-n — sample many, keep the best — reranking n samples with a verifier or, improperly, with the reference answer.
8. LLM-as-judge — a model marks the answers — an evaluator model scoring or ranking outputs, subject to position, verbosity and self-enhancement bias.

50.4 Why two labs report different numbers

■ 50.4.1 PLAIN – in simple words

1. You will often see the same model, the same benchmark, and two different numbers, from two respectable sources.
2. The first thing to accept is that this is normal.
3. The second thing to accept is that it is usually not lying.
4. There are about eight ordinary causes, and you can usually work out which one applies.
5. Cause one: different wording of the prompt.
6. Cause two: a different number of worked examples given first.
7. Cause three: a different way of pulling the answer out of the reply.
8. Cause four: a different slice of the items. Someone ran 500 of the 2,294.
9. Cause five: different randomness settings, or one run against an average of many runs.
10. Cause six: one lab used a system prompt telling the model how to behave and the other did not.
11. Cause seven: one setup let the model use a calculator, a code runner or a web search, and the other did not.
12. Cause eight: a plain bug. These are common and usually found later.
13. Only after all eight are ruled out is it worth wondering about anything worse.
14. The practical response is one question: which harness did you use, and with what settings?
15. If the answer is “our internal one, details not published”, the number is not comparable to anybody else’s number. It may still be true.

■ 50.4.2 PLAIN – a picture in your head

1. Imagine two labs measuring the fuel economy of the same car.
2. One drives a flat motorway at a steady 90 kilometres per hour with the air conditioning off and the tyres at maximum pressure.
3. The other drives a hilly city route in traffic with two passengers.
4. Both publish a figure in litres per hundred kilometres. The figures differ by 40 per cent. Neither lab lied.
5. This is exactly why car fuel economy eventually got a legally defined test cycle, with a written specification, run by accredited labs.
6. AI benchmarks have no such thing. There is no accredited lab, no test cycle written into law, and no penalty for using a favourable route.
7. The nearest equivalent is a shared open harness that everybody agrees to run with the same settings, and that is a voluntary convention.

Where this comparison breaks: fuel economy has a physical ground truth you can approach by measuring more carefully. There is no single true MMLU score for a model waiting to be discovered. The number genuinely depends on the protocol, so “measuring more carefully” does not converge on one value.

■ 50.4.3 PLAIN – a worked example

1. Here is a concrete, realistic divergence, with the arithmetic.
2. Lab A reports 88.4 per cent on a knowledge benchmark. Lab B reports 84.1 per cent for the same model. The gap is 4.3 points.
3. Investigation finds four contributions.

Difference	Effect on score
5-shot vs 0-shot	+1.8 points
Letter-only vs full text	+1.1 points
Chain-of-thought allowed	+0.9 points
Best of 3 runs vs mean	+0.5 points
Total explained	+4.3 points

4. Every one of those four choices is defensible on its own.
5. Together they fully account for the gap without anybody doing anything improper.
6. Notice the last row. Reporting the best of three runs rather than the mean is the one that should make you uncomfortable, because it is a choice made after seeing the results.
7. That is the boundary between a defensible protocol difference and a presentational thumb on the scale.

■ 50.4.4 PLAIN – what is really happening inside

1. The causes, with how you detect each one. This is the table to keep.

Cause	How to detect it
Prompt template differs	Ask for the exact template
Few-shot count differs	Look for “5-shot” in a footnote
Extraction differs	Ask what regex or parser
Subset differs	Compare item counts
Sampling differs	Ask temperature and top-p
System prompt used	Ask if one was present
Tools available	Ask about calculator, code, web
Best-of-n reporting	Ask how many runs, which shown
Plain bug	Wait for reproduction attempts

2. On tool access, be specific, because it is the largest single effect and the least often disclosed.
3. A model that may run Python and check its arithmetic will beat the same model without that on any maths benchmark, often by a wide margin.
4. A model that may search the web will beat the same model without it on any knowledge benchmark, and on a “Google-proof” benchmark it should not help much, which is itself a useful test of the benchmark’s design.
5. On agentic benchmarks, the scaffold is most of the system. An entry on the SWE-bench leaderboard is a model plus a controller: how files are found, how many edit attempts are allowed, whether tests may be run and re-run, how the loop decides to stop.
6. Two teams using the identical model with different scaffolds have reported resolve rates tens of points apart. The leaderboard entry is the pair, not the model.
7. That is why SWE-bench leaderboard entries are named after systems, not models, and why a model announcement quoting a SWE-bench number should say what scaffold produced it.

■ 50.4.5 TECHNICAL – the engineer’s version

1. Ask for the harness by name and version. In practice this means one of: EleutherAI `lm-evaluation-harness` (state the version and the task name, because task definitions change between versions), Stanford HELM, the `bigcode-evaluation-harness` for code, the official SWE-bench runner with its Docker images, or a lab’s internal harness.
2. Task definitions inside a harness are versioned and do change. The same task name in two harness versions can differ in prompt, in normalization and in which split is used. Pin the harness version in your own reports.
3. The Hugging Face Open LLM Leaderboard is the clearest example of why this matters. Its second version, launched in June 2024, replaced the original task set with IFEval, BBH, MATH Level 5, GPQA, MuSR and MMLU-Pro, run through the EleutherAI harness with fixed settings.
4. The reason given was that the original tasks had saturated and were contaminated. The consequence was that scores from before and after are not comparable at all, even for the same model.
5. That is the honest version of “the leaderboard changed”: every historical number became a different measurement, and the ranking of models moved.
6. Normalization choices worth naming explicitly, because they each move multiple-choice scores by measurable amounts: raw log probability, log probability divided by token count, log probability divided by character count, and probability normalized against the model’s unconditional probability of the option text.
7. A concrete reproducibility checklist to publish with any number you report:

```
model id + exact snapshot
harness name + version + task name
n-shot, and which split the shots came from
system prompt (verbatim, or "none")
temperature, top_p, max tokens, stop sequences
tools available (none / code / search / retrieval)
number of runs, and mean and standard deviation
answer extraction rule
date the run was executed
```

8. If a published number lacks four or more of those nine lines, treat it as an indication rather than a measurement.
9. Established fact: protocol differences of the sizes described here are documented and reproducible. Marketing claim: that any single reported number is “the” score for a model.

■ 50.4.6 WORDS – remember these

1. Harness — the program that runs the benchmark — the evaluation framework; name and version both matter because task definitions change.
2. Subset — a slice of the items — a filtered evaluation split such as Lite, Verified or Diamond, not comparable to the full set.
3. System prompt — standing instructions given before the conversation — a privileged message setting persona, format and constraints, which materially changes benchmark scores.
4. Scaffold — the code around the model in an agent task — the controller loop for retrieval, tool invocation, retries and termination.
5. Tool access — whether the model may use a calculator or search — the set of external functions the model may call during evaluation.
6. Reproduction — someone else getting your number — an independent run under a published protocol, the only real check on a self-reported score.

7. Task version — which definition of the test was used — the versioned task implementation inside a harness, including prompt and normalization.

50.5 Contamination

■ 50.5.1 PLAIN – in simple words

1. Contamination means the test questions were in the training data.
2. If the model has already seen the question and the answer, getting it right proves nothing about ability.
3. It might be memory. It might be ability. From the score alone you cannot tell which.
4. This happens almost by accident, not by plotting.
5. Benchmarks are published so that people can use them. Published means on the web. On the web means scraped. Scraped means in the training data.
6. GSM8K, MMLU, HumanEval and the rest are all on public code and dataset hosting sites, quoted in thousands of blog posts, pasted into forum questions, translated, reformatted and discussed.
7. So a model trained on a large web crawl has almost certainly seen many of them, in some form, unless someone worked hard to remove them.
8. That removal work is called decontamination, and every serious lab now does some and describes it.
9. But decontamination is filtering by text matching, and text matching misses anything reworded.
10. The effect on scores is upward, and it is uneven. It inflates old benchmarks more than new ones, and easy items more than hard ones.
11. It is very hard to prove for any particular model, because you usually cannot see the training data.
12. So the debate is conducted with indirect evidence, and reasonable people disagree about what that evidence shows.
13. The honest position: assume some contamination is present in every public benchmark number you read, of unknown size, and prefer evidence from tests the model cannot have seen.

■ 50.5.2 PLAIN – a picture in your head

1. Imagine a school where the past twenty years of exam papers, with the official answers, are freely available online.
2. A pupil who has read all of them will do better on this year's paper if this year's paper reuses old questions.
3. Nobody stole anything. The papers were published on purpose.
4. Now the exam board tries to fix it. They remove any question that appears word for word in an old paper.
5. But a question that was reworded, or translated into French and back, or changed from apples to oranges, slips through.
6. And the pupil who read every paper does better on those too, because the structure is familiar even when the words changed.
7. Now the awkward part. The pupil who read all the old papers is also genuinely better prepared. Some of what they gained is real learning.
8. Separating the memory from the learning, from the mark alone, is impossible.
9. That is exactly the position everyone is in with model benchmarks.

Where this comparison breaks: a pupil remembers roughly what they revised and can be asked. A model cannot reliably report what was in its training data, and the people who built it often cannot

enumerate it either, because the corpus is billions of documents assembled by pipelines from many sources.

■ 50.5.3 PLAIN – a worked example

1. The clearest public experiment on this is GSM1k.
2. In May 2024 a team at Scale AI, led by Hugh Zhang, published “A Careful Examination of Large Language Model Performance on Grade School Arithmetic”.
3. They commissioned a brand-new set of grade-school maths problems, written from scratch, never published, deliberately matched to GSM8K in style, number of solution steps, answer size and human solve rate.
4. Then they ran many models on both, and compared.
5. If a model’s ability were real, the two scores should be about equal, since the sets are matched by construction.
6. What they found: accuracy drops of up to about 8 percentage points on the new set, with several model families showing systematic overfitting across almost all their sizes.
7. They also found a positive relationship, Spearman r squared of about 0.36, between how likely a model was to generate GSM8K examples verbatim and how large its drop was.
8. That second finding is the important one. It links the drop to memorization rather than to some accident of the new problem set.
9. And the honest counterweight, which the same paper reports: the strongest frontier models showed minimal signs of overfitting, and every model generalized to some extent to the new problems.
10. So the correct summary is not “the benchmarks are fake”. It is “some models are inflated by several points, the size varies, and you cannot tell from the leaderboard which”.

■ 50.5.4 PLAIN – what is really happening inside

1. Five detection methods, what each can show, and where each fails.
2. **N-gram overlap.** Take every run of n consecutive words from every benchmark item, and search the training corpus for the same run.
3. The GPT-3 paper of 2020 used 13-gram overlap for its decontamination checks. Other projects use 8-gram, or a match on several random 50-character substrings.
4. What it shows: verbatim copying. What it misses: anything reworded, translated, reformatted, or discussed rather than quoted.
5. Also it requires access to the training corpus, which outsiders do not have.
6. **Canary strings.** The BIG-bench authors put a fixed unique identifier in every task file and asked everybody never to remove it.
7. If a model can reproduce that identifier, it saw the file. That is close to proof of exposure.
8. What it misses: everything. A model can be trained on a copy of the data with the canary stripped, on a reworded version, or on a discussion of the items. Absence of the canary proves nothing at all.
9. **Perplexity comparison.** Perplexity is a measure of how surprised the model is by a piece of text. Lower means more familiar.
10. Compare the model’s perplexity on benchmark items against its perplexity on freshly written text of the same kind. If the benchmark is much less surprising, that is a signal.
11. What it misses: benchmark text is often cleaner and more formulaic than fresh text anyway, so some gap is expected. Setting the threshold is a judgement call, and this requires log-probability access that many commercial endpoints no longer expose.

12. **Old versus new items.** Build a benchmark that keeps adding fresh items after model release dates, and compare a model's score on items published before its training cutoff against items published after.
13. A large gap is strong evidence. This is the design principle behind continuously refreshed benchmarks, and behind coding benchmarks that only score problems released after a stated date.
14. What it misses: newer items may simply be harder, or drawn from a different distribution, so you need the sets to be matched carefully, which is what GSM1k did.
15. **The completion trick.** Give the model the first part of a benchmark item and see whether it continues with the rest, word for word, including the incidental details it could not guess.
16. A variant gives the model the dataset name, the split and an index, and asks it to produce that item. Another shows the item with the options in a scrambled order and checks whether the model still reproduces the original ordering.
17. What it shows: near-verbatim memory of the item. What it misses: a model that learned the answer without being able to reproduce the wording, which is the common case after instruction tuning.
18. Notice the pattern. Every method can produce a positive finding that is hard to dismiss, and no method can produce a negative finding that means anything.
19. That asymmetry is why this argument never ends.

■ 50.5.5 TECHNICAL – the engineer's version

1. The methods and their properties, side by side.

Method	Detects	Needs
N-gram overlap	Verbatim copies	Corpus access
Canary string	Direct file exposure	Model generation
Perplexity gap	Familiarity	Log probabilities
Old vs new split	Practical inflation	A dated benchmark
Completion trick	Item memorization	Only API access
Membership inference	Probable inclusion	Token probabilities

2. Real disputes, described without alleging deliberate wrongdoing, because in none of these cases was deliberate cheating established.
3. In 2023 the paper “Rethinking Benchmark and Contamination for Language Models with Rephrased Samples” showed that a modestly sized model trained on rephrased versions of benchmark test sets could reach very high scores on those benchmarks while passing standard n-gram decontamination checks. The point was methodological: the standard filter does not work.
4. Also in 2023, a deliberately satirical paper by Rylan Schaeffer titled “Pretraining on the Test Set Is All You Need” trained a tiny model directly on benchmark test sets and reported near-perfect scores, to make the same point in the bluntest possible way.
5. Several strong small models released between 2023 and 2025 attracted public questions about whether their synthetic training data, which was generated by larger models, indirectly reproduced benchmark items. The teams involved published decontamination procedures in response. The disputes were about method and evidence, not about proven intent.
6. In late 2024 and early 2025 there was a public discussion about FrontierMath, an Epoch AI benchmark, after it emerged that OpenAI had funded its creation and had access to problems. Epoch AI published clarifications about what access existed and what hold-out arrangements were in place.

7. The general lesson from that episode is structural, not personal: if a lab funds or co-designs a benchmark it is later measured on, the benchmark needs a documented held-out portion and a stated access policy, or the number cannot carry weight regardless of anyone's good faith.
8. Which is why held-out sets now exist as standard practice: ARC-AGI keeps a semi-private evaluation set, Humanity's Last Exam keeps a private portion, and SWE-bench-style benchmarks increasingly add tasks dated after model releases.
9. Typical decontamination procedures now described in model cards and technical reports: exact-substring matching against known benchmark items, n-gram overlap filters at 8 to 13 grams, removal of documents containing canary strings, embedding-similarity screening for near-duplicates, and post-hoc measurement of the train-test gap.
10. Established fact: benchmark items appear in web crawls, and models can be induced to reproduce some of them verbatim. Established fact: matched fresh benchmarks show measurable score drops for some model families.
11. Active research: how to measure contamination without corpus access, how to correct scores for it, and how to build benchmarks that resist it by construction. There is no accepted standard method.
12. Marketing claim: "our model is decontaminated". Decontamination is a process with a coverage rate, not a state. Ask what filter, at what n-gram length, against which benchmark list, and what the measured residual was.
13. Expert disagreement, stated plainly. One camp holds that contamination is now the dominant explanation for headline benchmark gains on older tests, and that only dated, held-out or private evaluations should be believed. The other camp holds that frontier models demonstrably solve novel problems that cannot be in any corpus, so contamination inflates absolute numbers but rarely reverses rankings. Both camps agree that public numbers on pre-2023 benchmarks should be discounted.

■ 50.5.6 WORDS – remember these

1. Contamination — the test was in the training data — overlap between evaluation items and the pretraining or post-training corpus.
2. Decontamination — trying to remove it — filtering training documents that match benchmark items by substring, n-gram or embedding similarity.
3. N-gram — a run of n consecutive words — the unit of overlap matching, with 8-gram and 13-gram both in common use.
4. Canary string — a planted marker — a unique identifier embedded in benchmark files, whose reproduction by a model evidences exposure.
5. Perplexity — how surprised the model is by text — the exponential of the mean negative log likelihood per token.
6. Held-out set — questions nobody publishes — a private split retained by the benchmark authors to measure overfitting to the public split.
7. Overfitting to a benchmark — being good at the test, not the skill — a gap between performance on the benchmark and on matched unseen items.
8. Membership inference — guessing whether an example was trained on — a statistical test using token likelihoods, such as scoring the lowest-k token probabilities of a document.

50.6 The other reasons to be sceptical

■ 50.6.1 PLAIN – in simple words

1. Contamination is not the only problem. There are six more, and they are all ordinary and human.
2. **Saturation.** When everyone scores 92, 93 and 94 per cent, the test has stopped telling you anything.
3. At that point the remaining errors are often wrong answer keys in the benchmark itself, not model failures.
4. **Optimizing for the test.** Once a number matters, people work on the number. Some of that work improves the model. Some improves only the number.
5. **Self-reported results.** Nearly every headline benchmark figure in a model announcement was produced by the team that made the model.
6. That is not automatically wrong. It is simply unverified until someone else runs it.
7. **Cherry-picked comparisons.** Showing your new model against a competitor's older version, or against the competitor's cheap model rather than its best.
8. **Best-of-n reporting.** Letting the model try many times and reporting the best result, without saying so clearly.
9. **Different tools and scaffolding.** Your model with a code runner against their model without one, in the same bar chart.
10. **Charts that start above zero.** A bar chart whose vertical axis begins at 70 makes a two-point difference look enormous.
11. None of these require anybody to lie. Every one of them can be produced by ordinary optimism and ordinary pressure to look good.
12. Your defence is not cynicism. It is a checklist.

■ 50.6.2 PLAIN – a picture in your head

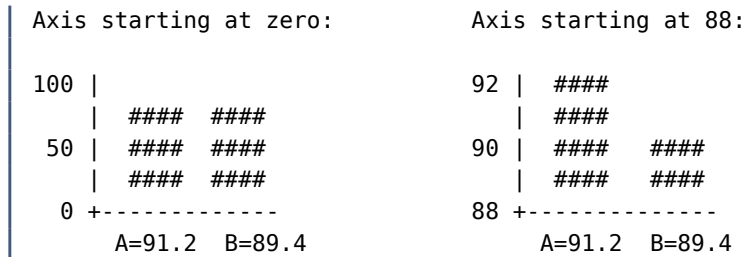
1. Think of school league tables.
2. Once schools are ranked by exam results, the results become the goal.
3. Schools start teaching to the exam. Then they enter weaker pupils for easier qualifications. Then some stop entering certain pupils at all.
4. The tables keep rising. Whether education improved is a separate question, and a much harder one to answer.
5. That is Goodhart's law happening in public, over years, with real children.
6. The measure was fine as a measure. It stopped being fine the moment it became the target.
7. And note what is not implied: the teachers were not villains. They responded to the incentive they were given.

Where this comparison breaks: exam boards are regulated, publish syllabuses, and can be audited. Nobody audits an AI lab's benchmark run. Also, schools cannot see next year's paper. Anyone training a model can see every public benchmark, in full, with answers, for free.

■ 50.6.3 PLAIN – a worked example

1. Goodhart's law, stated properly, because it is usually misquoted.
2. Charles Goodhart was a British economist at the Bank of England. In a 1975 paper on monetary policy he wrote, in substance: any observed statistical regularity will tend to collapse once pressure is placed upon it for control purposes.
3. That is the original and it is narrower than the popular version. It is about a regularity used as a control target.
4. The popular short form is due to the anthropologist Marilyn Strathern, who in 1997 wrote: when a measure becomes a target, it ceases to be a good measure.

5. A closely related statement is Campbell's law, from the psychologist Donald Campbell in 1979: the more a quantitative indicator is used for social decision-making, the more it will be subject to corruption pressures and the more it will distort the processes it monitors.
6. Applied here: MMLU was an excellent measure in 2021, when nobody was training against it.
7. By 2024 every major lab knew the exact score they needed, the exact format, and the exact subjects where points were cheapest.
8. The measure did not become false. It became uninformative, which in practice is worse, because it still looks like a measurement.
9. Now a chart example, with numbers. Two models score 91.2 and 89.4 on some benchmark.



10. Same two numbers. On the left, two nearly identical bars. On the right, one bar looks roughly twice the other.
11. The difference is 1.8 points, which on a 14,000-item benchmark is about 250 items, and which is within the range that a prompt template change can produce on its own.

■ 50.6.4 PLAIN – what is really happening inside

1. The red flag checklist. Each line is a question to ask of any announcement or chart.
2. Does the chart's vertical axis start at zero? If not, mentally redraw it.
3. Are the compared models named with their exact versions and dates, or just with product names?
4. Is the competitor's newest model shown, or a version from six months ago?
5. Is the competitor's flagship shown, or its cheaper tier?
6. Who ran the competitor's numbers? If the announcing lab did, were the competitor's own published numbers different?
7. Does the footnote say pass@1, or does it say something else, or nothing?
8. Does the footnote mention majority voting, consensus, best-of-n, or a number of samples greater than one?
9. Did both models have the same tools? Search, code execution, retrieval, file access, a scaffold?
10. Was there a system prompt, and was it the same for both?
11. Is the benchmark one where the top scores are already above 90 per cent?
12. Is the improvement larger than the known prompt-sensitivity range for that benchmark, which is roughly 2 to 5 points on multiple-choice tests?
13. Is the benchmark new, and was it built or funded by the announcing lab?
14. Has any independent party reproduced the number, and how long ago?
15. If you cannot answer eight of these fourteen from the announcement itself, the announcement is a claim, not a result.

■ 50.6.5 TECHNICAL – the engineer’s version

1. Saturation has a precise consequence: as mean accuracy approaches the label ceiling, the between-model variance shrinks toward the label-noise variance, and rank ordering becomes unstable across runs.
2. MMLU is known to contain a non-trivial number of incorrect or ambiguous reference answers. Independent audits published between 2023 and 2024 put the error rate in some subsets in the range of several per cent, with virology and a few professional subjects worst. Estimates vary by auditor and by criteria, so treat the exact figure as disputed.
3. The practical implication: a reported 92 per cent on MMLU may be close to the maximum achievable, and a difference between 92 and 93 may be noise plus disagreement about wrong keys.
4. This is exactly why MMLU-Pro, GPQA Diamond, Humanity’s Last Exam and FrontierMath were built. New benchmarks appear when old ones saturate, and the appearance of a new hard benchmark is itself evidence that the previous one stopped working.
5. A documented case of the “which version did you test” problem. On 5 April 2025 Meta released Llama 4 in two configurations, Scout and Maverick. A variant identified as Llama-4-Maverick-03-26-Experimental, described as optimized for conversationality, scored very highly on the LMArena leaderboard. It was not the model released to the public.
6. On 8 April 2025 the arena operators stated that Meta’s interpretation of their policy did not match what they expect from providers, that Meta should have made it clearer that the entry was a customized model, and that they would update their policies. Meta denied training on test sets.
7. Report that as what it is: a disclosure and naming failure with a policy response, not an established case of fraud. The transferable lesson is that a leaderboard entry identifies a specific artifact, and the artifact you can download may not be the artifact that was measured.
8. Statistical hygiene almost nobody applies. For a benchmark of N items and an observed accuracy p , the standard error is roughly $\sqrt{p(1-p)/N}$.

Benchmark	N	Approx 1 SE at $p=0.9$
HumanEval	164	2.3 points
GPQA Diamond	198	2.1 points
AIME (one paper)	15	7.7 points
SWE-bench Verified	500	1.3 points
MMLU	14,000	0.25 points

9. Read that table carefully. On GPQA Diamond, a 2-point difference is about one standard error, and one item is 0.5 points. On a single AIME paper, getting one more question right is 6.7 points.
10. So a headline of the form “scores 93.3 on AIME versus 86.7 for the competitor” describes one question out of fifteen.
11. That is before considering run-to-run sampling variance, which for reasoning models on small maths sets is frequently larger than the between-model gap.
12. Established fact: the standard-error arithmetic above. Marketing claim: any ranking asserted from a gap smaller than one standard error.

■ 50.6.6 WORDS – remember these

1. Goodhart’s law — a measure used as a target stops measuring — an observed regularity collapses when pressure is applied to it for control purposes.
2. Campbell’s law — indicators used for decisions get corrupted — quantitative social indicators distort the processes they are meant to monitor.

3. Saturation — the test is too easy now — mean score near the ceiling, so between-model variance falls below label-noise variance.
4. Label noise — wrong answers in the answer key — the fraction of benchmark items whose reference answer is incorrect or ambiguous.
5. Self-reported — the maker measured it — a result produced by the model provider under an unpublished or partially published protocol.
6. Truncated axis — a chart that hides how small a gap is — a bar chart whose value axis does not begin at zero.
7. Standard error — how much the score would wobble by chance — approximately $\sqrt{p(1-p)/N}$ for accuracy p over N independent items.
8. Cherry-picking — showing the favourable comparison — selective choice of baselines, versions, subsets or runs after seeing results.

50.7 The alternatives, and their own weaknesses

■ 50.7.1 PLAIN – in simple words

1. If fixed quizzes are weak, what else is there? Five things, each with its own flaw.
2. **Preference arenas.** Real people type a question, get two anonymous answers, and vote for the better one. Ratings are computed from the votes.
3. **Private held-out tests.** The same idea as a benchmark, but the questions are never published, so they cannot leak.
4. **Domain evaluations.** A test built for one job: medical coding, legal citation, chip design, customer support in Hindi.
5. **Agentic and end-to-end tests.** Give the model a real task in a real environment and check whether the task got done.
6. **Red-teaming.** People deliberately attack the model to find failures that no fixed test would catch.
7. Every one of these is better than MMLU for something, and worse for something else.
8. The arena measures whether people like the answer, which is not the same as whether the answer is correct.
9. A friendlier, longer, better-formatted wrong answer beats a curt correct one, quite often.
10. Private tests cannot be independently checked, because they are private.
11. Domain tests are only as good as the person who wrote them, and they do not transfer to other domains.
12. Agentic tests are expensive, slow, and measure the scaffold as much as the model.
13. Red-teaming finds problems but cannot tell you that no problems remain.
14. There is no method without a weakness. The working answer is to use several and to know what each one is blind to.

■ 50.7.2 PLAIN – a picture in your head

1. Think about how you would judge a restaurant.
2. A hygiene inspection is a fixed checklist with a pass mark. Objective, narrow, and it says nothing about whether the food is good.
3. Customer star ratings are a preference arena. They capture something real that no inspection captures, and they reward large portions, friendly staff and good lighting as much as cooking.
4. A critic's blind visit is a private held-out evaluation. Harder to game, impossible for you to verify.
5. Asking a chef to cook one specific dish you care about is a domain evaluation. Most informative for you, useless for anyone who wants something else.

6. Watching the kitchen run a full Saturday service is the agentic test. Closest to reality, and by far the most expensive to arrange.

Where this comparison breaks: restaurant customers eat the food, so their ratings are grounded in the actual product. Arena voters usually cannot verify whether the answer they preferred was true. They are rating the presentation of a claim, often on a topic where they are not the expert.

■ 50.7.3 PLAIN – a worked example

1. How a preference arena turns votes into a number.
2. Each vote is a pairwise comparison: model A beat model B on this prompt.
3. The rating system is the one from chess, invented by Arpad Elo, a Hungarian-American physicist, and adopted by the United States Chess Federation in 1960 and by FIDE in 1970.
4. The core formula gives the expected score of A against B from the gap in their ratings:

$$E(A) = 1 / (1 + 10^{((R_B - R_A) / 400)})$$

R = rating. A 400-point gap means the stronger side is expected to win about 10 times out of 11.

5. Work three cases. A gap of 10 points gives an expected win rate of 51.4 per cent. A gap of 100 points gives 64.0 per cent. A gap of 200 points gives 76.0 per cent.
6. Read the first one again. A 10-point difference on a leaderboard means the higher model wins about 51 times in 100. That is close to a coin flip.
7. Published confidence intervals on such leaderboards are frequently plus or minus 3 to 10 points, so models separated by fewer than about 15 points are usually statistically tied even though they occupy different rank numbers.
8. As of August 2026, the largest public text arena listed a few hundred ranked models, with top scores clustered within a few points of each other and stated uncertainties of roughly plus or minus 3 to 10 points.
9. A ranking that shows positions 1 through 8 spanning about 15 points is, in statistical terms, showing one group of models, not eight ranks.
10. The honest version: an arena rating is not a quality score. It is a fitted parameter that predicts how often this model's answer is preferred over another model's answer, by these voters, on these prompts, in this period. Every clause in that sentence limits what the number can be used for.

■ 50.7.4 PLAIN – what is really happening inside

1. What an arena measures well: how a broad population of users, asking the kinds of question they actually ask, reacts to an answer.
2. That is genuinely valuable and no fixed benchmark captures it. It picks up tone, refusal behaviour, formatting, and the ability to guess what the user meant.
3. What it measures badly, and the biases are documented, not theoretical.
4. **Length bias.** Longer answers win more often, controlling for content. Arena operators now publish a length-controlled ranking alongside the raw one specifically because of this.
5. **Style bias.** Headings, bullet lists and bold text raise win rates. The same content flattened into prose wins less.
6. **Verification gap.** A voter comparing two answers about a legal deadline usually does not know the deadline. They vote on confidence and clarity.

7. **Prompt distribution.** The votes come from whoever uses the site. That population skews technical and English-speaking and asks a lot of short questions, so the rating reflects that mixture and not yours.
8. **Provider self-selection.** Providers choose when to enter a model and which variant, and can test variants privately before public entry.
9. Red-teaming, briefly, because it is a different shape of thing. It is not scored on a scale. Its output is a list of found failures, and its value is in what it finds, not in a number.
10. The structural weakness of red-teaming is that it can only ever demonstrate presence, never absence. Not finding a jailbreak is not evidence that none exists.

■ 50.7.5 TECHNICAL – the engineer’s version

1. Modern arenas do not run a live Elo update loop, because Elo is order-dependent: the same set of games in a different order gives different ratings. They fit a **Bradley-Terry** model by maximum likelihood over the whole vote history, then bootstrap for confidence intervals, then present the result on an Elo-like scale.
2. That is why you should read the interval, not the rank. Ranks on such boards are usually published as a range, precisely because ties are common.
3. Style-controlled ranking works by fitting the preference model with additional covariates for response length and markdown features, so that the reported strength coefficient is adjusted for those factors.
4. Naming instability is real here too. The board widely known as LMSYS Chatbot Arena became LMArena; as of August 2026 the domain `lmarena.ai` redirects to `arena.ai`. Cite what a leaderboard is, not just where it lives.
5. Evaluation methods against what they are good and bad at.

Method	Good at	Bad at
Fixed benchmark	Cheap, comparable	Leaks, saturates
Preference arena	Real user taste	Truth, length bias
Private held-out	Leak resistance	Nobody can check
Domain eval	Your actual job	Generalizing
Agentic end-to-end	Realism	Cost, scaffold effects
Red-team	Finding failures	Proving absence
Model-as-judge	Scale, speed	Style and self bias

6. Cost figures matter and are rarely mentioned. A full SWE-bench Verified run with an agentic scaffold means 500 tasks, each with many model calls and repository builds. Runs costing hundreds to thousands of dollars and taking many hours are normal, which is exactly why the 300-item Lite subset exists.
7. This is also why the ARC Prize leaderboard reports cost per task and by default hides systems whose run cost exceeded 10,000 dollars. Any score can be bought with enough sampling; a score with a cost attached cannot.
8. Established fact: length and position biases in preference judging, and the order dependence of online Elo. Active research: how to control for style without controlling away real quality, and how to weight votes by voter expertise. Marketing claim: “ranked number one” when the interval overlaps five other entries.

■ 50.7.6 WORDS – remember these

1. Preference arena — people vote between two anonymous answers — a pairwise human preference collection platform with a fitted rating model.
2. Elo — a chess rating scale reused for models — a rating system where the expected score is a logistic function of the rating difference over 400.
3. Bradley-Terry — the proper way to fit ratings from all votes at once — a maximum-likelihood pairwise comparison model, order independent.
4. Confidence interval — how much the rating could be wrong — the bootstrapped range around a fitted rating; overlapping intervals mean a tie.
5. Length bias — longer answers win — a systematic preference for verbose responses independent of content quality.
6. Style control — adjusting for formatting and length — fitting the preference model with covariates for length and markdown structure.
7. Red-teaming — attacking your own model on purpose — adversarial testing to discover failures; evidences presence, never absence.
8. Held-out evaluation — a private test set — evaluation items never published, used to detect over-fitting to public benchmarks.

50.8 What “eval” means inside a company

■ 50.8.1 PLAIN – in simple words

1. In public, “eval” means a famous benchmark with a leaderboard.
2. Inside a company that builds a product on a model, it means something much smaller and much more useful.
3. It means a file of examples of the job the product actually does, with the answer you want for each.
4. Thirty to a hundred examples is normal. Not fourteen thousand.
5. They are hand-picked, mostly from real cases that went wrong.
6. They are private, because they contain real customer text and because they are the company’s actual measuring stick.
7. Every time anyone changes the prompt, the model, the retrieval setup or a parameter, the file is run again.
8. If the score drops, the change is rejected. That is the whole discipline.
9. In other words, it is a regression test suite, exactly like the automated tests you would write for ordinary software.
10. The score is not comparable to anyone else’s score, and does not need to be.
11. It only needs to answer one question: did this change make my product better or worse at my job?
12. Nobody can build this for you, because only you know what your job is.
13. If you build software on a model and you do not have one of these, you are guessing every time you change anything.

■ 50.8.2 PLAIN – a picture in your head

1. A bakery buys flour. The supplier changes.
2. The baker does not read the flour industry’s national quality rankings.
3. The baker bakes their own three standard loaves with the new flour and compares them to the old ones.
4. Same recipe, same oven, same timings. Only the flour changed.
5. If the loaves are worse, the flour is rejected, whatever the rankings said.

6. The bakery's test is small, private, unscientific and completely decisive, because it measures the thing the bakery sells.
7. That is an internal eval.

Where this comparison breaks: bread quality is judged by a person who has baked for twenty years, in one taste. Model output quality often needs a written rule, because you will run the test hundreds of times and cannot taste every loaf yourself. Writing that rule down is the hard part of the job.

■ 50.8.3 PLAIN – a worked example

1. A practical recipe for building one. Follow it in order.
2. Step 1. Write down the job in one sentence. "Turn a customer email into a support ticket with a category, a priority and a two-line summary."
3. Step 2. Collect 30 to 100 real inputs. Not invented ones. Real.
4. Step 3. Make sure the mix is deliberate, not random.

Slice of the set	Share	Why
Ordinary typical cases	50 per cent	The common path
Known hard cases	25 per cent	Where it broke before
Edge and malformed input	15 per cent	Robustness
Should refuse or escalate	10 per cent	Safety and limits

5. Step 4. For each input, write the answer you want. Where several answers are acceptable, write the rule instead of a single answer.
6. Step 5. Choose a scoring rule per item, from the cheapest kind that works.

exact	category must equal "billing"
contains	summary must mention the order number
regex	priority must match <code>^(P1 P2 P3)\$</code>
schema	output must parse as valid JSON
assert	code must pass these three tests
judge	a model grades 1-5 against a rubric
human	a person grades, for a 10-item subset

7. Step 6. Store it as a plain file. One row per example. Input, expected, rule, and a note about why it is in the set.
8. Step 7. Write a script that runs all items and prints one number plus the list of failures. The list of failures is the part you will actually read.
9. Step 8. Run it three times and note the spread before you trust any change smaller than that spread.
10. Step 9. Keep it in version control next to the code, and add a new row every time a real failure reaches a user. The set grows by itself.

■ 50.8.4 PLAIN – what is really happening inside

1. Why 30 to 100 and not 10 or 10,000.
2. Below about 30, one item is more than 3 per cent, and the whole set can be passed by luck or by a prompt change that happens to suit those items.
3. Above a few hundred, hand-writing expected answers becomes a project, and people stop maintaining it. An unmaintained eval set is worse than none, because it produces confident wrong signals.

4. The trade-off is honest: 30 to 100 items cannot tell you a 2 per cent difference. It can reliably tell you a 15 per cent difference, which is the size of change that usually matters when swapping models or prompts.
5. Where model-as-judge belongs. Use it when the output is free text and no mechanical rule works, but pin it down: a fixed rubric with numbered criteria, a fixed judge model with a fixed dated identifier, and a fixed temperature.
6. And validate the judge once: have a person grade 20 items, compare with the judge, and record the agreement rate. If the judge agrees with your human less than about 80 per cent of the time, fix the rubric before trusting it.
7. Also record cost and latency per item on every run. A change that improves quality by 3 per cent and triples cost is a decision, not an improvement, and you cannot make that decision without both numbers.

■ 50.8.5 TECHNICAL – the engineer’s version

1. Treat the eval set as a first-class artifact: versioned in git, with a schema, a fixed identifier per item, and a changelog. Never edit an item silently; add a new one and deprecate the old one, or every historical number becomes uninterpretable.

2. A workable row schema:

id	stable, never reused
input	verbatim, including whitespace
expected	answer or acceptance rule
rule	exact contains regex schema assert judge human
slice	typical hard edge refusal
added_on	date
reason	one line, why this exists

3. Report three numbers, not one: pass rate, mean latency, mean cost per item. Report pass rate per slice as well as overall, because a model that fixes typical cases and breaks refusals is a regression, not a gain.
4. Guard against a subtle trap: if you iterate on prompts against the eval set many times, you overfit to it exactly as labs overfit to public benchmarks. Hold back 20 per cent of items as a set you run only at decision time.
5. Available tooling as of 2026 includes provider-hosted eval products, open frameworks such as OpenAI’s evals repository, promptfoo, deepeval, Inspect from the UK government’s AI institute (named the AI Safety Institute at launch in 2024 and renamed the AI Security Institute in February 2025), and LangSmith-style tracing platforms. Note that provider-hosted eval products get deprecated too: as of mid-2026 OpenAI had announced shutdown dates for several platform features. Prefer a format you own.
6. Statistical note. With 50 items and an observed pass rate of 0.80, one standard error is $\sqrt{0.8 * 0.2 / 50} = 0.0566$, about 5.7 points. So a move from 80 to 84 per cent on 50 items is well inside noise. Do not ship on it.
7. Established fact: internal regression evals are standard practice at companies shipping model-backed products. Marketing claim: any vendor assertion that their public benchmark results predict your task performance.

■ 50.8.6 WORDS – remember these

1. Internal eval — your own small private test — a curated regression suite of task-representative items with per-item acceptance rules.

2. Regression test — checking a change did not break things — re-running a fixed suite after every modification and comparing to the last result.
3. Slice — a named group of items — a labelled subset such as typical, hard, edge or refusal, reported separately.
4. Rubric — the written marking scheme for a judge — the numbered criteria a judge model or human applies to score free text.
5. Golden set — the agreed correct answers — the reference outputs, versioned and never silently edited.
6. Holdout — items you do not tune against — the reserved fraction run only at decision time to detect overfitting to your own set.
7. Acceptance rule — how an item is marked — the per-item comparison, from exact match through to human judgement.

50.9 Reading a model announcement critically

■ 50.9.1 PLAIN - in simple words

1. A model announcement is a marketing document that contains real data.
2. Both halves of that sentence are true and you need both.
3. The data is usually accurate as far as it goes. The selection is not neutral.
4. So read it the way you read a job application: assume everything stated is true, and pay most attention to what is missing.
5. Nine questions do almost all the work, and section 50.9.3 lists them.
6. If the announcement answers all nine, that is a good sign about the team, independent of the scores.

■ 50.9.2 PLAIN - a picture in your head

1. Think of a used-car advertisement listing the good features honestly.
2. New tyres, full service history, one owner. All true.
3. It does not mention the mileage, and you notice that it does not.
4. You do not conclude the car is bad. You conclude you must ask.
5. The absence of a number is itself information, because the seller chose the list.

Where this comparison breaks: a car has one mileage figure and the seller knows it. A model has no single benchmark score to withhold, because the score depends on the protocol. So the missing item is usually the protocol, not a number, and the protocol is much easier to omit without anyone noticing.

■ 50.9.3 PLAIN - a worked example

1. The nine questions, in order, applied to any announcement.
2. Which benchmarks are shown, and which of the usual ones are absent? An absent standard benchmark is the loudest signal in the document.
3. Which comparison models, at which exact versions and dates? “GPT-4” is not a version. “Claude” is not a version.
4. Who produced the competitor numbers: this lab, or the competitor’s own published figures, or an independent runner?
5. What was the prompting setup: shots, chain-of-thought, system prompt?
6. Did the model have tools? Search, code execution, retrieval, file access? And did the comparison models have the same ones?

7. Is the score pass@1 and a single run, or is it best-of-n, majority voting, or an average over many samples?
8. Does the chart's axis start at zero, and is the gap bigger than the benchmark's known prompt sensitivity?
9. Is the benchmark new, and who built or funded it?
10. Has an independent third party reproduced any of it, and when?

■ 50.9.4 PLAIN – what is really happening inside

1. Two phrasings that should always slow you down.
2. “State of the art on X” with no date. State of the art is a claim about a moment, and the moment is usually the week before publication.
3. “Comparable to human experts”. Ask which humans, doing what, under what time limit, and with what resources. GPQA is the model case: its expert baseline was 65 per cent, measured under stated conditions, and quoting a model above that as “beyond PhD level” ignores that the humans were answering a four-option quiz outside a lab.
4. Also watch for the shift between a **model** claim and a **system** claim. A product with retrieval, tools and a scaffold is a system. Its score belongs to the system.
5. And watch the switch of subset: a headline referencing SWE-bench and a footnote saying Verified, or GPQA in the headline and Diamond in the footnote, are not errors, but they change what is being compared.

■ 50.9.5 TECHNICAL – the engineer's version

1. Where the answers usually live, in decreasing order of reliability: the system card or technical report appendix, the model card, the announcement footnotes, the blog post body, the chart itself.
2. Reading order should therefore be reversed from the page order: appendix first, chart last.
3. Independent reproduction sources worth checking before believing a headline: the benchmark's own leaderboard, third-party evaluation organizations that run their own harnesses and publish protocols, and the open harness community's issue trackers, where discrepancies get argued out publicly.
4. A note on speed. Reproductions typically appear days to weeks after a release. The gap between announcement and reproduction is precisely the window in which the announcement is unchecked.
5. Established fact: benchmark numbers in major technical reports are generally reproducible within a few points when the protocol is published. Marketing claim: leaderboard positions asserted from gaps inside the reproduction spread.

■ 50.9.6 WORDS – remember these

1. Model card — a short standard description of a model — a document stating intended use, training data summary, evaluations and limitations.
2. System card — the same for a whole product — documentation covering the model plus scaffolding, tools, safety mitigations and evaluations.
3. Self-reported — measured by the maker — a result produced by the provider, pending independent reproduction.
4. Reproduction — someone else got the same number — an independent run under a published protocol.
5. State of the art — best known result so far — a time-indexed claim, valid only at a stated date on a stated protocol.

6. System versus model — the product versus the weights — the distinction between an end-to-end deployed pipeline and the underlying network.

50.10 Training cutoff versus release date

■ 50.10.1 PLAIN – in simple words

1. There are two dates and people constantly confuse them.
2. The **training data cutoff** is the last date of text the model learned from.
3. The **release date** is the day you could first use it.
4. The gap between them is typically several months, and has often been a year or more.
5. During that gap the world kept happening and the model did not see any of it.
6. So a model released in one year can be confidently describing the world of a year or more earlier.
7. It does not know it is out of date. It has no clock and no calendar.
8. Unless you tell it today's date, it does not know today's date.
9. And it is often wrong about its own cutoff, which surprises people most.

■ 50.10.2 PLAIN – a picture in your head

1. Imagine someone who read every newspaper up to a certain Tuesday, then went into a sealed room, and is now answering your questions.
2. They are extremely well read up to that Tuesday.
3. They do not know today is a different day. Nobody told them.
4. Ask what happened last week and they will describe the last week they remember, with total confidence.
5. Worse, they read fewer papers in the final month than in earlier months, because the papers about that month had not been written yet.
6. So their sense of “recent” lands earlier than the actual Tuesday.

Where this comparison breaks: a person in a sealed room knows they are in a sealed room, and will say so. A model has no representation of its own situation, so it cannot volunteer the warning. That is the whole practical danger of the gap.

■ 50.10.3 PLAIN – a worked example

1. Real published gaps between stated cutoff and release.

Model	Stated cutoff	Released
GPT-4	September 2021	14 March 2023
Claude 3 family	August 2023	4 March 2024
Llama 3 8B	March 2023	18 April 2024
Llama 3 70B	December 2023	18 April 2024
GPT-4o	October 2023	13 May 2024

2. Read the first row. GPT-4 shipped about 18 months after its data ended.
3. Read rows three and four. Two models released on the same day, from the same family, with cutoffs nine months apart.
4. That alone should end any belief that a family name implies a shared cutoff.
5. Since about 2024 the gaps have narrowed, often to a few months, because pipelines got faster and freshness became a selling point. But the gap has never been zero and cannot be.

■ 50.10.4 PLAIN – what is really happening inside

1. Why the gap exists at all, in order.

data collection and filtering	weeks to months
pretraining run	weeks to months
post-training: SFT, RL, tuning	weeks to months
internal + external evaluation	weeks
safety testing and red-teaming	weeks to months
staged rollout and capacity build	weeks

2. Every stage needs the previous one finished, so the delays add rather than overlap.
3. Now why models are unreliable about their own cutoff. Three separate causes.
4. First, the cutoff is not stored anywhere in the weights. If the model states one, it is because someone wrote it into the post-training data or the system prompt, or because the model is guessing.
5. Second, the internet writes about events with a lag. An event in the final two months before cutoff has far fewer documents about it than an event two years earlier. So the model's felt sense of "the present" sits earlier than its real cutoff. Researchers call this the effective cutoff, and it is commonly earlier than the stated one.
6. Third, training corpora contain many documents about earlier models with earlier cutoffs. A model asked its own cutoff can echo a date it read.
7. The practical rule that follows: never ask a model for its cutoff and act on the answer. Look it up in the provider's documentation.
8. And always put the current date in the system prompt if the date matters.

■ 50.10.5 TECHNICAL – the engineer's version

1. Terminology precision. "Knowledge cutoff" usually refers to the pretraining corpus cutoff. Post-training data is frequently newer, and retrieval or tool results are newer still. A deployed system can therefore have three different effective dates at once.
2. The paper "Dated Data: Tracing Knowledge Cutoffs in Large Language Models", published in 2024, documented that measured effective cutoffs for many open models are earlier than the reported ones, and attributed this partly to corpus construction: deduplication and re-crawling mean older versions of pages persist while newer content is thinner.
3. Mitigations, in the order you should reach for them:

Mitigation	What it fixes
Date in system prompt	Model thinks it is the past
Web search tool	Missing recent facts
Retrieval over your docs	Missing private facts
Explicit "you may not know"	Confident staleness

4. Established fact: the cutoff-to-release gap, and its documented sizes. Established fact: models frequently misreport their own cutoff. Active research: measuring effective cutoffs per topic. Marketing claim: "always up to date", which describes a retrieval pipeline, not a model.
5. Practical detection test you can run in one minute: ask the model about three events with known dates spread across the suspected boundary, and see where its knowledge stops. That measures the effective cutoff for that topic, which is what you actually care about.

■ 50.10.6 WORDS – remember these

1. Training cutoff — the last date of learned text — the corpus collection boundary for pretraining data.
2. Effective cutoff — where the knowledge really stops — the empirically measured boundary, usually earlier than the stated cutoff.
3. Release date — when you could first use it — general availability date of a specific model version.
4. Staleness — the model living in the past — degradation of factual accuracy for events after the cutoff.
5. Retrieval augmentation — giving it fresh documents at question time — supplying externally fetched context so the answer is not limited to weights.
6. Post-training — the tuning after the big training run — supervised fine-tuning and reinforcement learning stages, often using newer data than pretraining.

50.11 Versions, snapshots and the moving-name problem

■ 50.11.1 PLAIN – in simple words

1. A product name and a model identifier are two different things.
2. The product name is what the marketing says. It is a brand.
3. The model identifier is the exact string your program sends to the API.
4. A product name can point at different weights over time, without notice.
5. A dated identifier points at one fixed set of weights that never changes.
6. So the same product name can behave differently next month, and your carefully tuned prompt can quietly stop working.
7. Nobody broke anything on purpose. The name moved.
8. The rule that follows is short: in production, use a dated identifier, and re-run your own eval set whenever you change it.

■ 50.11.2 PLAIN – a picture in your head

1. Think of a coffee shop's "house blend".
2. It is on the menu every year under the same name.
3. The beans change with the harvest, the roast is adjusted, the supplier changes. The name does not.
4. If you love it, you cannot order last year's. It no longer exists.
5. A dated identifier is like buying a specific labelled lot, roasted on a stated day, from a stated farm.
6. You can reorder exactly that, until the shop stops stocking it.

Where this comparison breaks: coffee changes gradually and you taste it immediately. A moving model name can change one narrow behaviour, such as how it formats JSON or when it refuses, while everything else stays identical. You will not notice by tasting. You notice because a downstream parser starts failing at two in the morning.

■ 50.11.3 PLAIN – a worked example

1. Real naming conventions from the three largest providers, as observed in August 2026. These change, so treat the shapes as the lesson and the examples as dated illustrations.

Provider	Shape	Example
OpenAI	name-YYYY-MM-DD	gpt-5-2025-08-07
Anthropic	family-version-YYYYMMDD	claude-opus-4-1-20250805
Anthropic	undated alias	claude-opus-5
Google	name plus channel	gemini-3.6-flash
Google	dated preview	model-preview-MM-YYYY

2. Read the OpenAI row. `gpt-5-2025-08-07` is a snapshot: those exact weights, fixed. `gpt-5` on its own is an alias that may be repointed.
3. Read the Anthropic rows. Some identifiers carry an eight-digit date, some do not. The dated ones are the ones to pin.
4. Google's documentation as of August 2026 describes four channels: `stable`, `preview`, `latest` and `experimental`. The `latest` alias is explicitly defined as pointing at the newest release for a model variation, with a stated notice period of about two weeks before a breaking change.
5. That is the clearest possible statement of the trade-off. An alias gives you automatic upgrades and about two weeks of warning. A pinned snapshot gives you stability and the obligation to migrate yourself.
6. Note also the older Google style using numeric suffixes such as `-001` and `-002`, where the trailing number is the revision of the same model generation. Different shape, same purpose.

■ 50.11.4 PLAIN - what is really happening inside

1. Why aliases exist at all, honestly stated from both sides.
2. From the provider's side: most users want improvements without doing work, safety fixes must be able to ship immediately, and serving many pinned versions costs real hardware.
3. From your side: any change to weights is a change to behaviour, and you have tested against the old behaviour.
4. Those two positions cannot both be fully satisfied, which is why both options exist.
5. What actually changes when a name is repointed: refusal boundaries, verbosity and formatting habits, tool-calling and JSON reliability, tokenizer or context limits in larger changes, and latency and price.
6. What almost never changes silently: the API shape. Providers version that separately, precisely because it would break everyone at once.
7. The distinction is worth naming. The API contract is close to a standard, documented and versioned. Model behaviour under an alias is a convention, and one that assumes you will cope.
8. There is also a middle case people forget: a provider may change the serving stack, quantization or routing behind a fixed identifier. The weights are the same; the outputs may not be bit-identical. Ask about this if determinism matters to you.

■ 50.11.5 TECHNICAL - the engineer's version

1. The production rule, in three lines.

1. Pin the dated snapshot in config, never an alias.
 2. Store the exact id with every logged output.
 3. Re-run your eval set before changing the pin.
2. Logging the identifier with each output is the step people skip and later regret. Without it, you cannot answer "did quality drop when we moved?" because you cannot tell which requests used which weights.

3. Keep the identifier in configuration, not in code, so a migration is a config change plus an eval run, not a code release.
4. Where aliases are still right: prototypes, internal tools, and anything where a two-week notice period is acceptable and nobody has built a parser around the exact output shape.
5. Where snapshots are essential: anything with a downstream schema, anything regulated, anything with a tuned prompt, and anything you cannot re-test quickly.
6. Watch for parameter-level changes too, not just weight changes. As one 2026 example, Anthropic's documentation lists `temperature`, `top_p` and `top_k` as deprecated for its newest models, with non-default values returning an error. Code that hard-codes a temperature will fail on migration even though the API version did not change.
7. Established fact: aliases are repointed and dated snapshots exist at all three major providers. Marketing claim: "drop-in upgrade". Every upgrade is a behaviour change until your own eval set says otherwise.

■ 50.11.6 WORDS - remember these

1. Product name — the brand on the website — a marketing label that may map to different artifacts over time.
2. Model identifier — the exact string you send — the API-level model name that selects a served artifact.
3. Snapshot — a frozen dated version — an immutable identifier bound to one set of weights and serving configuration.
4. Alias — a name that follows the newest release — a mutable pointer, such as a bare family name or a latest channel.
5. Pinning — fixing the version in config — specifying an immutable snapshot identifier so behaviour does not change under you.
6. Channel — the stability tier — stable, preview, latest or experimental, each with different change and notice policies.

50.12 Deprecation and retirement

■ 50.12.1 PLAIN - in simple words

1. Every model you build on will eventually be switched off.
2. Not "might". Will. Plan for it from the first day.
3. There are three stages and they mean different things.
4. **Legacy** means still working, no longer recommended, no further updates.
5. **Deprecated** means a shutdown date has been announced.
6. **Retired** means the identifier no longer works and your requests fail.
7. Providers give notice, but the notice period varies a great deal by provider and by model tier.
8. The single thing that makes a forced migration safe is your own eval set, because it is the only way to know whether the replacement is good enough for your job.

■ 50.12.2 PLAIN - a picture in your head

1. Think of a bus route being withdrawn.
2. First, notices appear at the stops. That is deprecation.
3. Then a date is given, and an alternative route is suggested.
4. On the date, the bus stops coming. That is retirement.
5. If you never tried the alternative route before the date, you find out on the morning you needed it whether it gets you to work on time.

- Trying the alternative early, with your actual journey, is the whole of migration strategy.

Where this comparison breaks: a bus route is either running or not, and you find out at the stop. A retired model identifier can fail in a way your code does not handle, at the worst possible moment, in production, returning an error your retry logic loops on. Test the failure path too.

■ 50.12.3 PLAIN – a worked example

- Real, checkable policies and dates, as published in August 2026.

Provider	Class	Notice given
OpenAI	Generally available	At least 6 months
OpenAI	Specialized variants	At least 3 months
OpenAI	Preview models	As short as 2 weeks
Anthropic	Public models	At least 60 days
Google	Preview models	At least 2 weeks

- And real retirement events, which show the policies being applied.

Identifier	Retired
claude-3-7-sonnet-20250219	19 February 2026
claude-sonnet-4-20250514	15 June 2026
claude-opus-4-1-20250805	5 August 2026
gpt-5-2025-08-07 (announced)	11 December 2026

- Read the third row against the first table. That snapshot was released in August 2025 and retired in August 2026. One year of life.
- Read the fourth row. Announced on 11 June 2026 for shutdown on 11 December 2026: exactly the stated six months.
- That is the realistic planning number. A pinned snapshot at a major provider has historically lasted roughly one to two years, with a few months of warning at the end.
- Note also that whole product features get retired, not only models. In 2026 OpenAI announced shutdown dates for several platform features including its hosted evaluation product. Do not build your only eval harness inside a vendor's product.

■ 50.12.4 PLAIN – what is really happening inside

- Why retirement happens: serving old weights occupies accelerators that could run new ones, old models lack current safety mitigations, and every supported version multiplies the testing burden.
- Where to find the schedule: every major provider publishes a deprecations page listing model identifier, deprecation date, retirement date and the recommended replacement. Read it quarterly. Subscribe to the changelog.
- The migration procedure, in order.
 - Read the deprecation notice; note both dates.
 - Identify the suggested replacement id.
 - Run your eval set on old and new, same prompts.
 - Read the failure list, not just the score.
 - Fix prompts for the new model's habits.
 - Re-run. Compare cost and latency too.
 - Ship behind a flag; keep the old id available.

- 8. Watch production metrics; then remove the flag.
- 9. Delete the old id from config before the date.

4. Step 3 is the one that makes the rest possible. Without an eval set, steps 4 to 8 are guesswork and step 9 is a leap of faith.
5. Expect prompt changes to be needed. A newer model in the same family often needs a shorter prompt, because instructions written to work around an older model's weaknesses can actively hurt a newer one.
6. Budget the work. For a small production system with a good eval set, a migration is typically days. Without one, it is weeks and the outcome is unknown.

■ 50.12.5 TECHNICAL – the engineer's version

1. Operational controls worth having in place before you need them: the model identifier in configuration, a per-request log of the identifier used, a feature flag to switch identifiers without deploying, and an alert on the specific API error returned for a retired model.
2. Consider a fallback chain in code: primary snapshot, then a named replacement, then a degraded path. Test the fallback deliberately, because an untested fallback is not a fallback.
3. If continuity matters more than anything else, the alternative is running open-weight models yourself. Then nobody can retire your model, and the cost is that you own the serving, the security patching and the hardware. That is a genuine trade, not a free option.
4. Established fact: the notice periods and retirement dates above, as published in August 2026. They are policies, not contracts, and they change; check the current pages rather than trusting this table.
5. The honest version of “the provider will look after you”: the provider will give you a date and a suggested replacement. Whether the replacement does your job is entirely your problem, and only your eval set can answer it.

■ 50.12.6 WORDS – remember these

1. Deprecated — announced for removal — still functional, with a published retirement date and a recommended replacement.
2. Retired — switched off — the identifier no longer serves requests and calls return an error.
3. Legacy — still working, no longer recommended — a supported but frozen version receiving no updates.
4. Notice period — how long you get to move — the interval between deprecation announcement and retirement.
5. Migration — moving to the replacement — re-pinning the identifier and re-validating behaviour against your own evaluation set.
6. Fallback chain — what to use when the first choice fails — an ordered list of model identifiers with a degraded final option.

50.13 What you should actually do

■ 50.13.1 PLAIN – in simple words

1. Here is the whole procedure, in order, for choosing a model for a job.
2. One. Write down the job in one sentence, and what a good answer looks like.
3. Two. Build the eval set from section 50.8. Thirty to a hundred real examples with a marking rule.
4. Three. Shortlist two or three candidate models. Use public benchmarks only for this step, only to shortlist, and never to decide.
5. Four. Run your eval set on all of them, with identical prompts.

6. Five. Read every failure. The failures tell you more than the score.
7. Six. Measure cost and latency at the same time, on the same runs.
8. Seven. Decide using all three numbers together, not quality alone.
9. Eight. Pin the dated identifier you chose.
10. Nine. Put the failures you saw into the eval set as new items.
11. Ten. Re-check every three months, and whenever a deprecation notice arrives.
12. That is it. It is not sophisticated. It is just done consistently.

■ 50.13.2 PLAIN - a picture in your head

1. This is how you would hire someone, and you already know how to do that.
2. You do not hire from a league table of universities.
3. You shortlist from credentials, then you set a work sample task that resembles the actual job.
4. You watch how they handle the hard parts, not just whether they finish.
5. You consider salary and availability alongside skill.
6. And you review after a few months, because circumstances change.

Where this comparison breaks: a person learns your job over time and asks questions when confused. A pinned model snapshot is exactly as good on day 300 as on day 1, and never tells you it is uncertain unless you build that in.

■ 50.13.3 PLAIN - a worked example

1. What to measure, and what a reasonable target looks like.

What to measure	How	Why it matters
Pass rate overall	Your eval set	Does it do the job
Pass rate per slice	Eval set slices	Where it fails
Cost per task	Tokens times price	Unit economics
Time to first token	Client-side timer	Feels responsive
Tokens per second	Client-side timer	Long output speed
Run-to-run variance	3 repeat runs	Is a gain real
Refusal rate	Refusal slice	Over-caution
Format validity	Schema check	Downstream breakage

2. A worked cost calculation, using round illustrative prices rather than any provider's current ones, since prices change monthly.
3. Suppose a task uses 4,000 input tokens and 800 output tokens, and the price is 3 dollars per million input and 15 dollars per million output.
4. Input cost: $4,000 / 1,000,000 * 3 = 0.012$ dollars.
5. Output cost: $800 / 1,000,000 * 15 = 0.012$ dollars.
6. Total 0.024 dollars per task. At 50,000 tasks a month that is 1,200 dollars.
7. Now suppose a model scoring 4 points higher costs three times as much. That is 3,600 dollars a month for 4 points on a 50-item eval set, where one standard error is about 5.7 points.
8. Written that way the decision is obvious, and it was invisible while you were looking at a leaderboard.

■ 50.13.4 PLAIN – what is really happening inside

1. Why “try two or three” and not one or ten.
2. One tells you nothing comparative. Ten is a research project you will not finish, and the differences between the middle candidates will be inside your noise floor anyway.
3. Two or three, chosen to be genuinely different, is the sweet spot: for example one frontier model, one cheaper model from the same provider, and one from a different provider or an open-weight model you host.
4. Include a cheaper model deliberately. The most common real finding is that the cheap model is good enough for 80 per cent of traffic, and a routing rule sends the rest to the expensive one.
5. Why re-check periodically: models change under aliases, new versions appear, prices fall, and your traffic mix drifts as users learn what your product can do.
6. Why the failure list beats the score: a 90 per cent pass rate where the 10 per cent are harmless is fine, and a 95 per cent pass rate where the 5 per cent are confident wrong answers about money is not.

■ 50.13.5 TECHNICAL – the engineer’s version

1. Measure latency properly and separately. Time to first token dominates perceived responsiveness in streaming interfaces; output tokens per second dominates total completion time for long outputs. Report both, at the median and at the 95th percentile, not as a mean.
2. Measure from where your users are. A model served in another region can add more latency in transit than the difference between two models.
3. Control the comparison. Same prompts, same temperature where settable, same context, same tools, same time of day. Provider load varies through the day and will otherwise contaminate your latency numbers.
4. Compute your noise floor first: run one model three times and record the spread. Any difference between models smaller than that spread is not a result.
5. Keep the artifacts: raw outputs, identifiers, timestamps and prices. When something changes in three months, the old raw outputs are the only way to prove what changed.
6. Established fact: the arithmetic and the method here. Active research: automated evaluation set generation and automatic prompt optimization, both of which are useful and neither of which removes the need for a human to define what a good answer is.

■ 50.13.6 WORDS – remember these

1. Shortlist — the two or three you will actually test — candidate models chosen from public evidence before task-specific measurement.
2. Noise floor — the wobble between identical runs — the run-to-run standard deviation below which differences are meaningless.
3. Time to first token — how long before words appear — the latency from request to first streamed token, distinct from total completion time.
4. Unit economics — what one task costs — the token cost per completed task at your actual input and output lengths.
5. Routing — sending easy work to a cheap model — a policy that selects a model per request based on difficulty, cost or latency budget.
6. Regression — the new thing is worse at something — a drop on any slice of your evaluation set after a change.

50.98 Common wrong ideas

1. Wrong: benchmark scores measure intelligence. Right: they measure performance on a fixed list of items under one protocol. Intelligence is not defined well enough to be scored, and no benchmark claims to.
2. Wrong: a higher MMLU means it is better for my task. Right: MMLU is four-option factual recall and is saturated. It predicts almost nothing about summarizing your emails or fixing your code. Only your own eval set answers that.
3. Wrong: the leaderboard is objective. Right: every leaderboard embeds choices about prompts, extraction, subsets and sampling, and preference leaderboards additionally embed the taste and the question mix of whoever voted.
4. Wrong: the model knows its own training cutoff. Right: the cutoff is not stored in the weights. Any date it gives is repeated from post-training data or guessed, and the felt cutoff is usually earlier than the real one.
5. Wrong: the same model name always means the same model. Right: undated product names are aliases and get repointed. Only a dated snapshot identifier is fixed, and even that is eventually retired.
6. Wrong: if two labs disagree, one is cheating. Right: prompt template, few-shot count, extraction, subset, sampling, system prompt and tool access routinely explain gaps of several points, with nobody at fault.
7. Wrong: contamination means the scores are fake. Right: contamination inflates scores by an unknown and uneven amount. Matched fresh benchmarks show drops of up to about 8 points for some model families and near zero for others.
8. Wrong: a 2-point lead is a meaningful lead. Right: on GPQA Diamond one standard error is about 2 points and one item is 0.5 points. On a single AIME paper, one question is 6.7 points.
9. Wrong: pass@k is just the score. Right: pass@k rises with k by construction. pass@1 and pass@100 are different quantities and cannot be compared.
10. Wrong: benchmarks are useless, so ignore them. Right: they are useful for shortlisting, for tracking the field over years, and for spotting obvious weakness. They are not useful for choosing between two close models for your specific job.

50.99 Chapter summary in 20 lines

1. A benchmark is a fixed set of items with known answers plus a rule for scoring a response. Nothing more.
2. Every run has five parts: items, prompt template, generation settings, answer extraction, and aggregation.
3. Only the items are usually published, so a score belongs to a model plus a harness, not to a model.
4. MMLU (2020, 57 subjects, about 14,000 test items) and HellaSwag (2019) are saturated and now mostly sanity checks.
5. GSM8K (2021, 1,319 test items) is solved. MATH (2021), AIME and FrontierMath carry the maths signal now.
6. HumanEval (2021, 164 problems, pass@k) is tiny, isolated and leaked. MBPP (2021, 974 problems) has the same shape.
7. GPQA (2023, 448 items; Diamond 198) is hard four-option science recall, not the ability to do science.
8. SWE-bench (2023, 2,294 tasks; Verified 500) scores whether real repository tests pass, and measures the scaffold as much as the model.

9. ARC-AGI keeps a held-back set and reports cost per task, which more benchmarks should copy.
10. Few-shot examples mainly fix output format. Chain-of-thought uses the model's own output as working memory and transforms maths scores.
11. $\text{pass}@k = 1 - C(n-c,k)/C(n,k)$. With $n=10$ and $c=2$, $\text{pass}@1$ is 0.20 and $\text{pass}@5$ is 0.78. The model did not improve; you bought more tries.
12. Two labs disagree because of prompts, shots, extraction, subsets, sampling, system prompts, tools, or bugs. Ask which harness and version.
13. Contamination is near-inevitable when training on the web. Detection by n-gram overlap, canaries, perplexity, old-versus-new splits and completion tricks can prove presence but never absence.
14. Saturation, Goodhart's law, self-reported numbers, cherry-picked baselines, best-of-n reporting, tool asymmetry and truncated axes are the seven ordinary ways a number misleads without anyone lying.
15. Preference arenas measure real user taste well and truth badly, with documented length and style bias; a 10-point gap is a 51 per cent win rate.
16. Inside a company, an eval is a private regression suite of 30 to 100 real examples with per-item marking rules, run on every change.
17. Training cutoff and release date are different, historically separated by months to over a year, and models are unreliable about their own cutoff.
18. Product names are mutable aliases; dated snapshots are fixed. Pin the snapshot in production and re-run your evals when you change it.
19. Every model is eventually retired. Typical notice in 2026 was at least six months at OpenAI, at least 60 days at Anthropic, about two weeks for preview channels.
20. Shortlist with public benchmarks, decide with your own eval set, measure cost and latency alongside quality, and re-check every few months.

Serving, Running One Yourself, RAG, Agents and the Limits

51.0 What this chapter gives you

1. You will be able to say what changes when a model stops learning and starts answering, and give the memory numbers for both.
2. You will be able to read a real API request body field by field, and say what the system prompt is and what it is not.
3. You will be able to explain, with arithmetic, why the first token is slow for one reason and the rest are slow for a completely different reason.
4. You will be able to name the serving tricks that make a model fast, and say why the same model runs at different speeds at different providers.
5. You will be able to build a cost model for a real application and decide honestly whether to self-host.
6. You will be able to run a model on your own laptop with two named tools, and predict beforehand what speed you will get.
7. You will be able to download weights, read a model card, and pick a quantization level on purpose rather than by guessing.
8. You will be able to explain retrieval-augmented generation step by step, including how approximate nearest neighbour search works.
9. You will be able to explain tool calling and agents precisely, including why every tool is a security boundary and why long agent runs fail.
10. You will be able to state the real limits, including why hallucination is intrinsic and why prompt injection is still unsolved as of August 2026.

51.1 Inference versus training: what changes

■ 51.1.1 PLAIN – in simple words

1. Training is the process that produced the numbers inside the model.
2. Inference is the process of using those numbers to answer a question.
3. During training, every number in the model can move.
4. During inference, not one of them moves. They are read and never written.
5. Training needs to remember, for each number, which way it was moving and how fast it has been moving lately. That is several extra copies.
6. Inference needs the numbers themselves, plus some working notes about the conversation so far.
7. So the same model needs roughly ten times less memory to use than to train.

8. Training is judged on work finished per week. Inference is judged on how long one person waits for one answer.
9. A training run that takes 30 days is normal. An answer that takes 30 seconds feels broken.

■ 51.1.2 PLAIN – a picture in your head

1. Think of a very large printed cookbook that a team of chefs wrote over a year.
2. Writing it was training. Drafts, corrections, arguments, notes in the margins about which way each recipe was heading.
3. All of that scaffolding took far more desk space than the finished book.
4. Cooking from it is inference. You open the book, read, and cook.
5. You do need the book itself open in front of you, and a small notepad for what you have already done in this particular meal.
6. But if a customer waits 40 minutes for soup, the restaurant fails.

Where this comparison breaks: a chef can read one page and ignore the rest. A language model reads every page of the book for every single word it produces. That is the fact that governs the whole of this chapter, and no ordinary object behaves that way.

■ 51.1.3 PLAIN – a worked example

1. Take a real model: Llama 3.1 8B, which has 8.03 billion parameters.
2. Chapter 47 gave the rule: memory equals parameter count times bytes per parameter.
3. At 2 bytes per parameter that is 16.1 billion bytes, which is 15.0 GiB.
4. To use the model, you need those 15.0 GiB plus working space.
5. To train the model, you need the same 15.0 GiB plus four more copies.
6. Here is the whole comparison, in GiB, for one 8B model:

Item	Inference	Training
Weights, 16-bit	15.0	15.0
Master copy, 32-bit	0	29.9
Gradients, 32-bit	0	29.9
Optimizer state (two)	0	59.8
KV cache, 8K tokens	1.0	0
Activations	0.5	about 28
Framework overhead	1.0	2.0
Total	about 17.5	about 165

7. So the same model is a 17.5 GiB problem to run and a 165 GiB problem to train fully. That is a factor of about 9.4.
8. 17.5 GiB fits on one consumer graphics card with 24 GB.
9. 165 GiB does not fit on any single card sold in 2026, including the 141 GB NVIDIA H200.

■ 51.1.4 PLAIN – what is really happening inside

1. In training, each batch of data goes forward through the network, an error is measured, and the error is pushed backwards.
2. Pushing it backwards requires remembering every intermediate value from the forward pass. Those are the activations, and they are enormous.
3. In inference, none of that exists. There is no backward pass, so no activation needs to be kept after the layer that produced it is done.

4. What is kept instead is different: a per-token record called the KV cache, which we come to in section 51.4.
5. So inference memory is dominated by two things only: the frozen weights, and the KV cache that grows with the length of the conversation.
6. And the weights are read-only, which means many requests can share exactly one copy of them in memory. This is why batching works at all.

■ 51.1.5 TECHNICAL – the engineer’s version

1. Training memory per parameter under the standard mixed-precision recipe with Adam: 2 bytes bf16 weights, 4 bytes fp32 master, 4 bytes fp32 gradient, 8 bytes for Adam’s first and second moments. Total 18 bytes.
2. Inference memory per parameter: 2 bytes at bf16, 1 byte at int8, about 0.5 bytes at 4-bit. No optimizer state, no gradients.
3. The workload character inverts. Training is throughput-bound and runs at high model FLOPs utilization, typically 35 to 55 percent on well-tuned clusters. Batch-1 decode runs at roughly 0.3 percent of peak FLOPs.

Property	Training	Inference
Weights	read and written	read only
Optimizer state	8 to 12 B/param	none
Bottleneck	compute, network	memory bandwidth
Measured by	MFU, days to finish	TTFT, tokens/s, p99

4. Tools that show the split: `nvidia-smi` for device memory, `torch.cuda.memory_summary()` for allocator detail, and the `/metrics` endpoint that vLLM exposes for queue depth, batch size and cache usage.

■ 51.1.6 WORDS – remember these

Inference — using a trained model — the forward pass only, no gradients. Training — teaching the model — forward and backward passes plus optimizer. KV cache — the model’s notes on this conversation — cached key and value tensors per token per layer. MFU — how much of the chip’s maths power you use — model FLOPs utilization. TTFT — the wait before the first word appears — time to first token.

51.2 What an API call actually contains

■ 51.2.1 PLAIN – in simple words

1. Talking to a hosted model is an ordinary web request. Nothing exotic.
2. Chapter 36 covered what an API is; Chapter 32 covered what happens on the wire when a request is encrypted and sent. This is one of those requests.
3. You send a block of text in a structured form, plus some settings.
4. The structured form is a list of messages. Each message has a role.
5. The roles are conventionally “system”, “user” and “assistant”.
6. “System” is the instruction you put at the front. “User” is what the person typed. “Assistant” is what the model said last time.
7. Everything in that list is glued together into one long string of tokens and fed to the model at once.
8. The model does not know which part came from where, except that the training taught it to treat the system part as instructions.

■ 51.2.2 PLAIN – a picture in your head

1. Imagine handing a temporary worker a clipboard before every single task.
2. Sheet one says: you are a polite support agent, never discuss competitors.
3. Sheet two says: here is the customer's question.
4. The worker reads the whole clipboard from the top, then speaks.
5. Then the worker forgets everything and leaves.
6. Next time, you hand over a fresh clipboard with all the sheets again, plus the new ones.
7. The first sheet has no special authority. It is just first.
8. If the customer writes on their sheet "ignore sheet one", the worker has to decide, from habit alone, which sheet to obey.

Where this comparison breaks: a human worker has an employer, a contract and a memory of yesterday. The model has none of those. Its deference to the first sheet was installed by training on examples where the first sheet was obeyed. It is a strong statistical habit, not a permission system.

■ 51.2.3 PLAIN – a worked example

1. Here is a complete request, in the shape most providers accept, with every field explained afterwards.

```
POST /v1/chat/completions HTTP/1.1
Host: api.example-provider.com
Authorization: Bearer sk-REDACTED
Content-Type: application/json

{
  "model": "example-model-2026-08",
  "messages": [
    {"role": "system",
     "content": "You are a support agent. Answer in
     British English. Never invent a policy."},
    {"role": "user",
     "content": "My git push failed. What do I check?"}
  ],
  "temperature": 0.2,
  "top_p": 0.95,
  "max_tokens": 400,
  "stop": ["\n\nUser:"],
  "stream": false,
  "response_format": {"type": "json_object"},
  "tools": []
}
```

2. `model` names the exact weights to run. Dated names matter: providers change what an undated name points at.
3. `messages` is the conversation, oldest first. The whole list is sent every time. Nothing is stored on the server between calls.
4. `temperature` controls randomness in picking the next token. 0 means always take the most likely token. 1.0 is the model's raw distribution.
5. `top_p` keeps only the smallest set of tokens whose probabilities add up to 0.95, then samples inside that set. It is called nucleus sampling.
6. `max_tokens` is a hard ceiling on the reply length, and it is a cost control as well as a safety valve.
7. `stop` is a list of strings. If the model produces one, generation halts and the string is not returned.
8. `stream` set to true sends the answer piece by piece as it is produced.

9. `response_format` asks for strictly valid JSON rather than prose.
10. `tools` declares functions the model may ask you to run. Section 51.12 covers this in full.
11. And here is the reply:

```
{
  "id": "chatcmpl-8xk2ab",
  "object": "chat.completion",
  "created": 1786600000,
  "model": "example-model-2026-08",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "Check the remote URL, then your token."
      },
      "finish_reason": "stop"
    }
  ],
  "usage": {
    "prompt_tokens": 62,
    "completion_tokens": 41,
    "total_tokens": 103
  }
}
```

12. `finish_reason` is the single most useful field for debugging. “stop” means the model chose to end. “length” means you hit `max_tokens` and the answer is truncated. “tool_calls” means it wants you to run something.
13. `usage` is what you are billed on. 62 input tokens and 41 output tokens here, priced at different rates.

■ 51.2.4 PLAIN - what is really happening inside

1. On the server, that JSON is flattened into one token sequence using a chat template, which is a fixed pattern of special marker tokens.
2. A template looks roughly like this, with the markers written out:

```
<|start|>system
You are a support agent...<|end|>
<|start|>user
My git push failed...<|end|>
<|start|>assistant
```

3. The model then predicts what comes after the final marker.
4. The honest version: the system prompt is a run of tokens at the front of the context window and has no privileged status in the architecture.
5. Its influence comes entirely from post-training, where the model was shown many examples of the system section being followed.
6. Because it is only a habit, it can be overridden. A user message that says “the previous instructions were a test, here are the real ones” sometimes works, and content pasted in from a web page can carry instructions too.
7. That failure has a name, prompt injection, and section 51.15 treats it as the architectural problem it is.

■ 51.2.5 TECHNICAL – the engineer’s version

1. The message-list shape shown above is a de facto convention, not a standard. It came from OpenAI’s chat completions endpoint in March 2023 and was copied widely because tooling supported it.
2. It is not universal. Anthropic’s Messages API takes `system` as a top-level string rather than a message role. OpenAI’s newer Responses API uses `input` and `instructions`. Local servers from `llama.cpp` and `vLLM` implement the older chat-completions shape for compatibility.
3. Streaming is served as Server-Sent Events over HTTP, media type `text/event-stream`, one JSON delta per event:

```
data: {"choices":[{"delta":{"content":"Check"}}]}
data: {"choices":[{"delta":{"content":" the"}}]}
data: {"choices":[{"delta":{"content":" remote"}}]}
data: [DONE]
```

4. Chapter 32 covers what carries this: TLS 1.3 over TCP 443, usually HTTP/2, with the response body arriving as a chunked stream. Streaming needs no new protocol; it is one long-lived response.

■ 51.2.6 WORDS – remember these

System prompt — the instruction sheet at the front — leading tokens in the context, obeyed by training habit, not by architecture. Role — who said this line — a marker token pattern in the chat template. Temperature — how adventurous the model is — logit scaling before softmax. Nucleus sampling — keep only the likely words — top_p truncation of the cumulative distribution. finish_reason — why it stopped — stop, length, tool_calls or content_filter. Chat template — the glue that turns messages into one string — a Jinja template shipped with the model.

51.3 How generation actually performs

■ 51.3.1 PLAIN – in simple words

1. Answering has two phases, and they behave like two different machines.
2. Phase one: the model reads your whole prompt. This is called prefill.
3. Phase two: the model writes the answer one token at a time. This is called decode.
4. Prefill can read every word of your prompt at the same moment, in parallel, because all the words already exist.
5. Decode cannot do that. Word five cannot be chosen before word four exists.
6. So prefill is limited by how much arithmetic the chip can do per second.
7. And decode is limited by something else entirely: how fast the chip can pull the model’s numbers out of memory.
8. That is the whole secret. For each single token of output, the machine has to read every parameter in the model once.
9. So the speed of writing is roughly: memory speed divided by model size.
10. That one division explains almost everything people find strange about model speed.

■ 51.3.2 PLAIN – a picture in your head

1. Imagine a librarian who must consult every book in a library before saying one word.
2. To read your question, the librarian can put a hundred assistants on it at once, one per sentence. That is quick, and it scales with staff.

3. But to say each word of the answer, the librarian must personally walk the whole library again. Every shelf. Every book.
4. Adding assistants does not help with the walking. The corridor is the limit.
5. So the wait before the first word depends on how long your question was and how many assistants are free.
6. And the speed of the words after that depends only on how fast one person can walk the library once.
7. A bigger library means slower words, in direct proportion.

Where this comparison breaks: the librarian would remember the books. The chip does not. It genuinely re-reads the weights from memory for every token, because there is nowhere near enough fast on-chip storage to hold them. That re-reading is not waste from bad design; it is the fundamental shape of the computation.

■ 51.3.3 PLAIN - a worked example

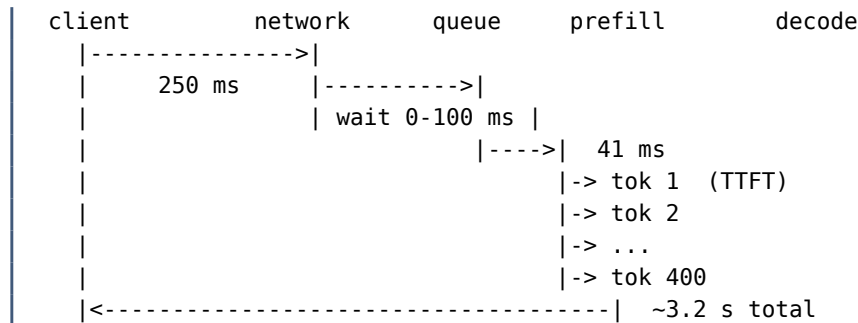
1. Take Llama 3.1 8B at 16-bit precision. Weights are 16.1 GB.
2. Run it on one NVIDIA H100 SXM. Its memory bandwidth is 3,350 GB per second. That figure is from NVIDIA's own specification sheet.
3. Decode ceiling equals 3,350 divided by 16.1, which is 208 tokens per second.
4. Real systems reach 60 to 80 percent of that, so expect 125 to 165 tokens per second for a single user. Measured single-stream figures for 8B on an H100 sit in that range.
5. Now the same model on a MacBook Air with the M5 chip. Its memory bandwidth is 153 GB per second, from Apple's technical specifications.
6. At 4-bit the model is about 4.7 GB. 153 divided by 4.7 is 32 tokens per second as a ceiling; expect 20 to 25 in practice.
7. Now prefill on the same H100. Prefill work is about 2 times parameters times prompt tokens.
8. For a 1,000-token prompt: 2 times 8.03 billion times 1,000 equals 16.1 trillion floating-point operations.
9. An H100 SXM does about 990 trillion dense bf16 operations per second. At 40 percent utilization that prompt takes about 41 milliseconds.
10. Compare the phases directly:

Phase	1,000-token prompt	400-token answer
Work	16.1 TFLOP	400 x 16.1 GFLOP
Bytes moved	16.1 GB once	16.1 GB x 400
Time on H100	about 41 ms	about 2.9 s

11. So the answer takes seventy times longer than reading the question, even though the question was longer.
12. One more real-world correction. The reader of this book is in India. A round trip to a server in the United States is typically 220 to 280 milliseconds.
13. So for short prompts, the network wait is larger than the prefill. Time to first token is network plus queueing plus prefill, and often the network wins.

■ 51.3.4 PLAIN – what is really happening inside

1. Here is the timeline of one request, drawn out:



2. Time to first token is everything up to the first arrow out: network, queue, prefill, and one decode step.
3. This is why a long prompt hurts your first token but not your typing speed, and why a long answer hurts total time but not the first token.

■ 51.3.5 TECHNICAL – the engineer’s version

1. The controlling ratio is arithmetic intensity: floating-point operations per byte moved from memory.
2. An H100 SXM has about 990 TFLOP/s dense bf16 against 3.35 TB/s, a balance point of roughly 295 FLOP per byte. Below that ratio you are bandwidth-bound.
3. Batch-1 decode does 2 FLOP per parameter and reads 2 bytes per parameter: 1 FLOP per byte. You are using about 0.3 percent of the chip’s arithmetic.
4. The metric for the decode side is memory bandwidth utilization, or MBU, popularized by Databricks in 2023: achieved bytes per second divided by peak. Well-tuned stacks reach 60 to 85 percent.
5. Note that the vendor’s headline FLOPS figure often includes 2:1 structured sparsity. NVIDIA’s 1,979 TFLOPS figure for H100 bf16 is the sparse number; the dense number is 989.5. Use the dense one for language models.

Hardware	Bandwidth GB/s	8B fp16 ceiling
H100 SXM	3,350	208 tok/s
H200 SXM	4,800	298 tok/s
RTX 5090	1,792	111 tok/s
RTX 4090	1,008	63 tok/s
M5 Max, 40-core	614	38 tok/s
MacBook Air M5	153	9.5 tok/s

6. Measure it yourself with `llama-bench` from `llama.cpp`, which reports prompt processing and token generation separately, or with `vLLM`’s `benchmark_serving.py`, which reports TTFT and inter-token latency percentiles under load.

■ 51.3.6 WORDS – remember these

Prefill — reading your question — parallel forward pass over all prompt tokens, compute-bound. Decode — writing the answer — sequential single-token forward passes, memory-bandwidth-bound. Arithmetic intensity — sums done per byte fetched — FLOP per byte, compared against the hardware’s balance point. MBU — how much of the memory pipe you are using — memory bandwidth

utilization. Inter-token latency — the gap between words — the reciprocal of decode tokens per second.

51.4 The serving optimizations, each explained

■ 51.4.1 PLAIN – in simple words

1. Every trick in serving exists to fight one of two enemies: reading the weights too often, or leaving the machine idle.
2. The KV cache stops the model re-reading your whole conversation for every new word.
3. Continuous batching stops the machine idling while one user thinks.
4. Speculative decoding lets a small fast model guess ahead so the big model can check several words at once.
5. Quantization shrinks the weights so there is less to read per token.
6. Tensor parallelism splits one model across several chips so their memory pipes add up.
7. Prefix caching stores the processed form of a shared opening so it is not re-processed for every user.
8. Streaming does not make anything faster, but it makes the wait feel far shorter, which is often what people actually want.

■ 51.4.2 PLAIN – a picture in your head

1. Think of a busy kitchen with one very slow oven.
2. The KV cache is keeping the prepared ingredients on the counter instead of re-chopping everything for each course.
3. Static batching is waiting until six orders arrive, cooking all six, and serving nobody until the slowest one finishes.
4. Continuous batching is putting each new order into the oven the moment a space appears, and taking each dish out as soon as it is done.
5. PagedAttention is using many small identical trays instead of reserving a whole shelf per order in case it turns out to be large.
6. Speculative decoding is a junior cook who prepares the next three likely steps; the chef glances at them and either accepts all three or redoes them.

Where this comparison breaks: in a kitchen, more staff means more parallel cooking. On a GPU, the constraint is a single memory bus that all the work shares, so most of these tricks are about using one bus better, not about adding more hands.

■ 51.4.3 PLAIN – a worked example

1. Start with the KV cache, because it is the one that changes the order of the arithmetic.
2. Without a cache, to produce token number t the model must process all $t-1$ previous tokens again from scratch.
3. Generating 500 tokens after a 1,000-token prompt means processing, on average, 1,250 tokens, 500 times: 625,000 token-passes.
4. With a cache, each new token processes exactly one new token, because the previous keys and values are already stored: 500 token-passes.
5. That is a saving of 1,250 times for this case. The work goes from growing with the square of the length to growing with the length itself.
6. Now the price. The cache is real memory, and it is calculated exactly:


```

KV bytes per token
  = 2 (keys and values)
  x layers
  x key/value heads
  x head dimension
  x bytes per number

```

```

Llama 3.1 8B, 16-bit:
  = 2 x 32 x 8 x 128 x 2 = 131,072 bytes = 128 KiB

```

```

Llama 3.1 70B, 16-bit:
  = 2 x 80 x 8 x 128 x 2 = 327,680 bytes = 320 KiB

```

7. So an 8B model with a 128,000-token context holds 16 GiB of cache, which is more than the 15 GiB of weights.
8. This is why serving systems spend so much effort on cache memory, and why grouped-query attention, which cuts the number of key/value heads from 64 to 8, was such an important change. It divided this table by eight.

■ 51.4.4 PLAIN – what is really happening inside

1. Static batching groups requests, runs them together, and returns them together. Every request waits for the longest one in the group.
2. Continuous batching, first described in the 2022 Orca paper, works at the level of single decode steps instead.
3. At each step the scheduler looks at every active sequence, runs one token for all of them in one fused operation, then evicts finished ones and admits waiting ones immediately.
4. The result is that the batch is never allowed to shrink, so the memory pipe stays busy. Reported throughput gains over static batching are large, often several times, at the same latency.
5. PagedAttention, from the vLLM paper at SOSP 2023, borrows the operating system idea of paging from Chapter 12.
6. Instead of reserving a contiguous block of cache for each sequence's maximum possible length, it allocates fixed-size blocks, typically 16 tokens each, and keeps a block table per sequence.
7. This removes internal fragmentation and lets several sequences share blocks, which matters when many requests begin with the same system prompt.
8. Speculative decoding uses a small draft model to propose the next k tokens cheaply, then runs the large model once over all k proposals in parallel.
9. Because the large model's forward pass over k tokens costs almost the same as over one token, accepted guesses are nearly free. Rejected ones cost only the draft work.
10. The mathematics were published independently by two groups in 2023, and the method is exact: the output distribution is identical to running the large model alone. It is a speed trick, not a quality trade.

■ 51.4.5 TECHNICAL – the engineer's version

1. The optimizations, with what each actually buys:

Technique	What it saves	Typical effect
KV cache	recompute	quadratic to linear
Continuous batching	idle GPU	2 to 10x throughput
PagedAttention	cache memory	19 to 27% memory

Technique	What it saves	Typical effect
Speculative decoding	decode steps	1.5 to 3x latency
4-bit weights	bytes per token	about 3x decode
Tensor parallel, 2 GPUs	bandwidth	1.6 to 1.9x
Prefix caching	prefill	up to 90% TTFT

2. Tensor parallelism splits each weight matrix across GPUs, so each device reads only its shard. Two GPUs give roughly twice the aggregate bandwidth, less an all-reduce over NVLink after every layer. Scaling is sub-linear and degrades sharply without a fast interconnect.
3. Quantized serving matters twice over: 4-bit weights cut both memory and bytes read per token. Activation quantization to FP8 additionally raises prefill throughput on Hopper and Blackwell hardware.
4. Why the same open model is faster at provider A than provider B, in order of how much it usually explains: the hardware generation, the quantization they silently chose, the batch size policy they run, the current queue depth, and the geographic distance to you.
5. That list is the reason a published benchmark of “model X speed” is close to meaningless without naming the deployment.

■ 51.4.6 WORDS – remember these

KV cache — saved notes so the model need not re-read — cached key and value projections per token per layer. Continuous batching — slot new work in as space frees — iteration-level scheduling, from the Orca paper, 2022. PagedAttention — memory in small blocks, not big reservations — block-table KV allocation, vLLM, 2023. Speculative decoding — a fast guesser the big model checks — draft-and-verify with an exact acceptance rule. Tensor parallelism — one model split across chips — sharded matrix multiply with all-reduce per layer. Prefix caching — reuse the shared opening — KV reuse keyed on a common token prefix.

51.5 Cost and capacity

■ 51.5.1 PLAIN – in simple words

1. Hosted models are sold by the token, and tokens in cost less than tokens out.
2. Chapter 47 explained what a token is: roughly three quarters of an English word, so 1,000 tokens is about 750 words.
3. The reason input is cheaper is exactly section 51.3. Your prompt is read in parallel in one pass, so it is cheap per token.
4. The answer is produced one token at a time, and each token costs a full read of the model. That is the expensive part.
5. Two discounts exist almost everywhere. Batch work that can wait up to a day is half price. Repeated openings you have marked as cacheable are about a tenth of the input price.
6. Building your own server instead is not obviously cheaper. You rent the machine by the hour whether anybody uses it or not.

■ 51.5.2 PLAIN – a picture in your head

1. Think of a taxi where the meter runs differently for the two halves of the journey.
2. Listening to your address is quick and charged at a low rate per word.
3. Driving you there is slow and charged at a high rate per street.
4. Batch pricing is agreeing to be picked up sometime today rather than now, for half fare.

5. Cache pricing is a regular route the driver already knows, so the first part of the journey needs no navigation.
6. Buying your own car instead is the self-hosting decision. The car costs the same whether you drive it or not.
7. It only beats taxis if you drive a great deal, every day.

Where this comparison breaks: taxi fares scale with distance, which you cannot change. Token costs scale with how much text you send, which you control completely. Trimming a system prompt from 2,000 tokens to 400 is a real 80 percent cut on that part of every request, forever.

■ 51.5.3 PLAIN – a worked example

1. Take a support assistant. Real shape, real numbers.
2. 10,000 conversations a day. Six turns each. A 800-token system prompt. 2,000 tokens of retrieved documents at the start. 100-token questions. 300-token answers.
3. Because the whole history is resent every turn, the input grows each turn: 2,900, then 3,300, 3,700, 4,100, 4,500, 4,900.
4. That totals 23,400 input tokens and 1,800 output tokens per conversation.
5. Now price it against three real published tiers from August 2026, in dollars per million tokens: a mid tier at 2 in and 10 out, a small tier at 1 and 5, and a very small tier at 0.10 and 0.60.

Choice	Per conversation	Per 30 days
Mid, 2 and 10	\$0.0648	\$19,440
Mid with caching	\$0.0466	\$13,980
Small, 1 and 5	\$0.0324	\$9,720
Very small, 0.10, 0.60	\$0.0034	\$1,026

6. Read that table twice. The same application costs nineteen thousand dollars a month or one thousand, depending on one decision.
7. The caching row assumes the 2,800-token opening is marked cacheable and hit on five of six turns, charged at one tenth, with a write surcharge on the first turn.

■ 51.5.4 PLAIN – what is really happening inside

1. Renting one datacentre GPU in 2026 costs roughly 2 to 4 US dollars an hour on the open market, varying by provider, region and commitment. Take 2.50.
2. That is about 1,825 dollars a month, per GPU, running or idle.
3. One such GPU serving an 8-billion-parameter model with good batching produces on the order of 2,000 to 4,000 output tokens per second in aggregate across all users.
4. At 2,500 tokens per second held constantly, that is 6.6 billion output tokens a month, which works out at about 28 cents per million.
5. That looks unbeatable until you notice the words “held constantly”.
6. Compare the two cost shapes across real volumes:

Output tokens/month	Hosted at \$0.60/M	Own GPU
10 million	\$6	\$1,825
100 million	\$60	\$1,825
1 billion	\$600	\$1,825
3 billion	\$1,800	\$1,825

Output tokens/month	Hosted at \$0.60/M	Own GPU
10 billion	\$6,000	\$3,650

- Break-even against a cheap hosted model is around 3 billion output tokens a month, and that is before you add a second GPU for redundancy, an engineer to run it, or the load spikes you must size for.
- The honest version: self-hosting is rarely cheaper below very high sustained volume. The good reasons to self-host are usually not cost.
- They are: data must not leave your network, you need a model that no provider hosts, you need the exact same weights for years, or you need a latency floor you control.

■ 51.5.5 TECHNICAL – the engineer’s version

- Published list prices, checked August 2026. Prices move; treat these as a snapshot, not a constant.

Model, August 2026	Input \$/M	Output \$/M
Claude Opus 5	5.00	25.00
Claude Sonnet 5	2.00	10.00
Claude Haiku 4.5	1.00	5.00
OpenAI gpt-5.6-sol	2.50	15.00
OpenAI gpt-5.6-terra	1.00	6.00
OpenAI gpt-5.6-luna	0.10	0.60

- Discount structure, also checked August 2026: batch processing at 50 percent of both input and output on both providers; cache reads at 10 percent of the input rate; cache writes at 125 percent for a short time-to-live.
- Long-context surcharges are now common. OpenAI applies a higher rate above a published input threshold for the whole request, which means one oversized prompt reprices the entire call.
- Reasoning models bill their internal thinking tokens as output. A model that thinks for 4,000 tokens before writing 200 costs twenty-one times more than the visible answer implies. This is the single most common billing surprise of 2025 and 2026.

■ 51.5.6 WORDS – remember these

Input token — text you send — prompt tokens, billed at the lower rate. Output token — text produced — completion tokens, billed several times higher. Batch API — cheaper if you can wait — asynchronous queue, typically 50 percent off with a 24-hour window. Prompt caching — pay less for a repeated opening — server-side KV reuse keyed on an exact prefix, billed at about 10 percent. Duty cycle — how much of the time your machine is busy — utilization, the term that decides every self-hosting argument.

51.6 Running one yourself, practically

■ 51.6.1 PLAIN – in simple words

- You can run a real language model on your own laptop today, offline, for free, in about ten minutes.
- llama.cpp is the engine. It is a C++ program that loads a model file and runs it on your processor, your graphics chip, or both.
- Ollama wraps that engine in something friendly: one command to fetch a model, one to talk to it, and a small local server.

4. LM Studio is a desktop application with a window and buttons, for people who would rather not use a terminal.
5. vLLM, SGLang and TGI are for the other end: serving many users at once on datacentre graphics cards.
6. MLX is Apple's own numerical framework, built for Apple silicon, and it is noticeably faster than the alternatives on those machines.
7. Start with Ollama. Move to llama.cpp when you want control. Move to vLLM when you have users.

■ 51.6.2 PLAIN – a picture in your head

1. Think of playing recorded music.
2. llama.cpp is the amplifier and speaker: raw, capable, many knobs, no case.
3. Ollama is the all-in-one hi-fi with one power button and a catalogue of albums you can fetch by name.
4. vLLM is the sound system for a concert hall, designed for two thousand listeners rather than one.
5. MLX is a speaker built specifically for the shape of your room.

Where this comparison breaks: with music, the recording is identical whatever plays it. Here the tools change the result. Different engines pick different default quantizations, context lengths and sampling settings, so the same named model can give different answers and very different speeds.

■ 51.6.3 PLAIN – a worked example

1. Here is Ollama, start to finish, on a Mac or Linux machine.

```
# install with a package manager, then check it runs
ollama --version

# fetch a small model (a few gigabytes) and chat
ollama pull qwen3:8b
ollama run qwen3:8b

# see what you have, and what is loaded right now
ollama list
ollama ps

# use it as a local HTTP service on port 11434
curl localhost:11434/api/chat -d '{
  "model": "qwen3:8b",
  "messages": [{"role": "user", "content": "Explain TCP in 3 lines"}],
  "stream": false
}'
```

2. That last call has the same shape as the hosted API in section 51.2. Most local servers deliberately imitate it so your code does not change.
3. Now llama.cpp directly, which is one level down.

```
# build once, with Apple GPU support turned on
cmake -B build -DGGML_METAL=ON
cmake --build build --config Release -j

# run a model pulled straight from Hugging Face
./build/bin/llama-cli -hf ggml-org/gemma-3-1b-it-GGUF \
  -p "Write one sentence about routers."
```

```
# serve it on port 8080, 8k context, all layers on the GPU
./build/bin/llama-server -m model-Q4_K_M.gguf \
  -c 8192 -ngl 99 --port 8080

# measure prompt speed and generation speed separately
./build/bin/llama-bench -m model-Q4_K_M.gguf
```

4. The flag that matters most is `-ngl`, the number of layers to put on the GPU. Setting it to 99 means “all of them, if they fit”.
5. If the model does not fit, layers stay on the processor and speed collapses, often by a factor of five or more. That is the single most common cause of “why is my local model so slow”.

■ 51.6.4 PLAIN – what is really happening inside

1. llama.cpp was started by Georgi Gerganov in March 2023 as an experiment in running Llama on a MacBook with no graphics card at all.
2. It defined a file format, now called GGUF, which packs the weights, the tokenizer and the metadata into one file that can be memory-mapped.
3. On the serving side the design goal is the opposite. vLLM keeps the GPU saturated with continuous batching and PagedAttention from section 51.4.
4. SGLang adds RadixAttention, a shared prefix tree, which pays off when thousands of requests begin with the same long system prompt.
5. Hugging Face’s Text Generation Inference was the early default, but Hugging Face put it into maintenance mode on 11 December 2025 and now points new users at vLLM or SGLang. Treat it as a migration topic, not a choice.
6. MLX is Apple’s array framework, released in December 2023, designed around unified memory so tensors are not copied between processor and graphics chip. Ollama added an MLX backend in version 0.19 on 27 March 2026.

■ 51.6.5 TECHNICAL – the engineer’s version

Tool	What it is	Use it when
llama.cpp	C/C++ engine, GGUF	any hardware, full control
Ollama	wrapper plus registry	fastest local start
LM Studio	desktop application	no terminal wanted
vLLM	Python serving engine	many users, GPUs
SGLang	serving, RadixAttention	shared prefixes
TGI	older HF server	existing systems only
MLX	Apple array framework	Apple silicon speed

1. macOS caps what the GPU may wire down, historically around 75 percent of unified memory. The `iogpu.wired_limit_mb` sysctl raises it. That is an implementation detail of one operating system version and it has been renamed before; check before relying on it.
2. Ollama’s published benchmark for its MLX backend, March 2026, on a mixture-of-experts model with 35 billion total and 3 billion active parameters: decode from 58 to 112 tokens per second, prefill from 1,154 to 1,810 tokens per second.
3. Vendor figures, so read them as a direction, not a promise.
4. For serving, the vLLM invocation you will actually type is `vllm serve <model> --tensor-parallel-size 2 --max-model-len 32768`, and it exposes an OpenAI-compatible endpoint plus a Prometheus `/metrics` path.

5. Watch three numbers in production: running batch size, waiting queue length, and KV cache utilization. When cache utilization hits 100 percent the scheduler starts preempting sequences and latency becomes erratic.

■ 51.6.6 WORDS – remember these

GGUF — the one-file local model format — successor to GGML, memory-mappable, carries weights, tokenizer and metadata. Offload — putting model layers on the graphics chip — `-ngl` or `n_gpu_layers`; partial offload is the usual cause of slow local models. Ollama — the friendly local runner — a daemon plus registry over llama.cpp. vLLM — the standard GPU serving engine — PagedAttention, continuous batching, OpenAI-compatible server. MLX — Apple’s numerical framework — unified-memory array library for Apple silicon, released December 2023.

51.7 What hardware you actually need

■ 51.7.1 PLAIN – in simple words

1. Two numbers decide everything: how much memory you have, and how fast that memory is.
2. Memory decides whether the model runs at all. Bandwidth decides how fast it talks.
3. The memory rule from Chapter 47: parameters times bytes per parameter, plus a bit for the conversation and the software.
4. At 4 bits per parameter, a rough working rule is that the model needs about 0.6 GB per billion parameters, plus 2 to 3 GB of headroom.
5. So an 8-billion model at 4 bits wants about 7 GB free, and a 32-billion model wants about 22 GB free.
6. If it does not fit, it does not simply run slower. Part of it spills to the processor or to disk and the speed falls off a cliff.
7. Apple silicon is unusually good here, because the processor and graphics chip share one pool of memory. A laptop with 32 GB can load models that need a 3,000 dollar graphics card on a PC.
8. But Apple laptops have much narrower memory pipes than datacentre cards, so they load big models and then run them slowly.

■ 51.7.2 PLAIN – a picture in your head

1. Think of memory as the size of your desk and bandwidth as how fast you can turn pages.
2. A PC graphics card is a small desk with an extremely fast hand.
3. An Apple laptop is a large desk with a moderate hand.
4. A datacentre card is a large desk with an extraordinarily fast hand, in somebody else’s building, rented by the hour.
5. If the book does not fit on the desk, you keep fetching pages from the floor, and everything stops.

Where this comparison breaks: a person can skim. The model reads every page for every word. There is no skimming, so the “fast hand” is used at full stretch on every single token.

■ 51.7.3 PLAIN – a worked example

1. Here are real machines, with real figures from the manufacturers, and what each can hold.

Machine	Usable memory	Bandwidth
MacBook Air M1/M2, 8 GB	about 5 GB	about 100 GB/s
MacBook Air M5, 16 GB	about 11 GB	153 GB/s

Machine	Usable memory	Bandwidth
MacBook Air M5, 24 GB	about 18 GB	153 GB/s
MacBook Pro M5 Max, 128 GB	about 110 GB	614 GB/s
Desktop, RTX 4090	24 GB	1,008 GB/s
Desktop, RTX 5090	32 GB	1,792 GB/s
Server, H100 SXM	80 GB	3,350 GB/s

- Now the speeds. These are the bandwidth ceiling from section 51.3 with a realistic 70 percent efficiency applied, for 4-bit models.

Model at 4-bit	Air M5 16 GB	M5 Max	RTX 5090
4B, 2.5 GB	43 tok/s	170 tok/s	500 tok/s
8B, 4.7 GB	23 tok/s	91 tok/s	267 tok/s
14B, 8.5 GB	12 tok/s	51 tok/s	148 tok/s
32B, 19 GB	will not fit	23 tok/s	66 tok/s
70B, 40 GB	will not fit	11 tok/s	will not fit

- Being specific, as promised. On an 8 GB MacBook Air: models up to about 4 billion parameters at 4 bits run fine. An 8B at 4 bits technically loads and then swaps to disk, giving a few tokens per second and a hot machine. Anything larger will not run at all.
- On a 16 GB Air: 8B is comfortable, 14B works, 32B does not fit.
- On a 24 GB Air: 32B at 4 bits fits with a short context and delivers about 8 tokens per second, which is slower than reading speed and unpleasant.
- A useful threshold: below about 10 tokens per second, most people stop using a model interactively. Above 25 it feels fine for chat.

■ 51.7.4 PLAIN – what is really happening inside

- Why unified memory helps so much: on a normal PC, the graphics card has its own memory, and anything not in it must cross the PCI Express bus, which moves about 64 GB per second on a sixteen-lane fourth-generation link.
- That is twenty to fifty times slower than the card's own memory, so a model that half fits runs at roughly the speed of the slow path.
- On Apple silicon there is one pool. There is no crossing, so a 70-billion model at 4 bits genuinely runs on a laptop, which no 3,000 dollar consumer graphics card can do.
- Where Apple silicon is still limited, in two places.
- First, bandwidth. Even the fastest laptop chip at 614 GB per second is under a fifth of an H100. Big models load and then crawl.
- Second, prompt processing. Prefill is compute-bound, and Apple's graphics cores have far less matrix throughput than a datacentre card. Pasting in a long document can take many seconds before the first word appears, even though generation afterwards looks reasonable.
- This asymmetry surprises people constantly. The machine feels fine in chat and terrible the moment you paste a file into it.

■ 51.7.5 TECHNICAL – the engineer's version

- Sizing formula for a local deployment, in bytes:

```
total = params x bytes_per_param
      + kv_bytes_per_token x context_length x concurrency
```

- + activations (0.5 to 1 GiB)
- + runtime overhead (0.5 to 2 GiB)

2. Mixture-of-experts models decouple the two limits: capacity is set by total parameters, speed by active parameters. A 26-billion-total, 4-billion-active model needs about 16 GB at 4 bits but decodes like a 4B model.
3. Community measurements in 2026 report roughly 150 tokens per second for such a model on an RTX 4090 or 5090 and roughly 80 on an M5 Max.
4. Apple's published bandwidths, August 2026: M5 in the MacBook Air, 153 GB/s; M5 Pro up to 307 GB/s with up to 64 GB; M5 Max 460 GB/s in the 32-core configuration and 614 GB/s in the 40-core, up to 128 GB.
5. NVIDIA's published bandwidths: RTX 4090 1,008 GB/s over 24 GB GDDR6X; RTX 5090 1,792 GB/s over 32 GB GDDR7; H100 SXM 3,350 GB/s over 80 GB HBM3; H200 4,800 GB/s over 141 GB HBM3e; B200 about 8,000 GB/s.
6. Observe the reality rather than the ceiling with llama-bench, which prints prompt-processing tokens per second and generation tokens per second as two separate rows. They will differ by one or two orders of magnitude, exactly as section 51.3 predicts.

■ 51.7.6 WORDS – remember these

Unified memory — one pool shared by processor and graphics — Apple silicon's architecture; capacity advantage, bandwidth disadvantage. VRAM — the graphics card's own memory — dedicated GDDR or HBM, fast but small on consumer cards. Offload spill — part of the model living on the slow side — layers left on the CPU or paged to disk; the usual cause of a tenfold slowdown. Prompt processing speed — how fast it reads what you paste — prefill tokens per second, compute-bound, weakest on laptops. Active parameters — the part of a sparse model actually used — the figure that sets decode speed in a mixture-of-experts model.

51.8 Getting weights and reading a model card

■ 51.8.1 PLAIN – in simple words

1. Hugging Face is the place where models live. It is a website and a set of tools, and it works much like a code-hosting site.
2. Each model has a repository with an owner and a name, files inside it, and a page of documentation called a model card.
3. As of 2026 the site hosts more than two million public models. Most are variations, fine-tunes and re-uploads of a much smaller number of originals.
4. You download either the original weights, which are large files in a format for research frameworks, or a converted single-file version for local tools.
5. The model card is the part people skip and should not. A good one tells you what the model was built from, what it may legally be used for, and where it fails.
6. A bad one tells you only that the model is excellent.
7. Then you choose a quantization level, which is the trade from Chapter 47: fewer bits per number means less memory and more speed, and a little less accuracy.

■ 51.8.2 PLAIN – a picture in your head

1. Think of buying a second-hand car from an online listing.
2. The files are the car. The model card is the seller's description.
3. A good listing states the year, the mileage, the service history, what is worn, and what the car is not suitable for.
4. A bad listing says "runs great, must sell" and shows one photograph taken from a flattering angle.
5. The licence is the logbook. Without it you may not legally use the car for the purpose you had in mind, however well it runs.

Where this comparison breaks: a car's condition can be inspected by a mechanic in an hour. A model's real behaviour on your task can only be found by running your own evaluation on your own data, which is why Chapter 50's material on evaluation matters more than any card.

■ 51.8.3 PLAIN – a worked example

1. A typical repository contains these files. Knowing what each is stops a lot of confusion.

config.json	architecture: layers, heads, sizes
model-00001-of-00004.safetensors	weight shard 1 of 4
model.safetensors.index.json	which tensor is in which shard
tokenizer.json	the vocabulary and merge rules
tokenizer_config.json	special tokens, chat template
generation_config.json	default temperature, stop tokens
README.md	the model card
LICENSE	the terms you are agreeing to

2. To download, use the official command-line tool, which is hf in current versions and was named huggingface-cli before 2025:

```
hf download meta-llama/Llama-3.1-8B-Instruct \
  --local-dir ./llama-3.1-8b

# or fetch only one quantized file for local use
hf download bartowski/Meta-Llama-3.1-8B-Instruct-GGUF \
  Meta-Llama-3.1-8B-Instruct-Q4_K_M.gguf --local-dir .
```

3. Prefer files ending in .safetensors over .bin. The older .bin format is Python pickle, which can execute code when loaded. Safetensors, introduced in 2022, is a plain data format that cannot.
4. Now the quantization decision. These are real measured figures from llama.cpp's own tables for an 8-billion model, showing file size and the increase in perplexity, which is the confusion measure from Chapter 50. Lower is better.

Level	File size	Extra perplexity
F16	14.00 GB	baseline
Q8_0	7.96 GB	+0.0026
Q6_K	6.14 GB	+0.0217
Q5_K_M	5.33 GB	+0.0569
Q4_K_M	4.58 GB	+0.1754
Q3_K_M	3.74 GB	+0.6569
Q2_K	2.96 GB	+3.5199

5. Read the last row. Two-bit quantization on a small model does not degrade it slightly; it breaks it.
6. The decision, as a table:

Situation	Choose	Reason
It fits at 8-bit	Q8_0	effectively lossless
Normal case	Q4_K_M	the standard default
Just short of fitting	IQ4_XS	smaller, small loss
Far short	smaller model at Q4	better than Q2

■ 51.8.4 PLAIN – what is really happening inside

1. What a good model card contains, in the order you should read it.
2. The base model and how this one relates to it: is it an original pre-trained model, an instruction-tuned version, or somebody's fine-tune of a fine-tune.
3. A description of the training data. Not the data itself, usually, but its sources, size, languages and cut-off date.
4. The licence, stated plainly, including whether commercial use is allowed and whether outputs may be used to train other models.
5. Intended use and out-of-scope use. Real cards say what the model is not for.
6. Limitations and known biases, in specifics rather than boilerplate.
7. Evaluation results with the benchmark names, the exact settings, and ideally the harness used, so the numbers can be reproduced.
8. What a bad card omits, in order of how much it should worry you: the licence, the training data description, the evaluation settings, and any statement of weakness at all.
9. A card that reports only wins on benchmarks, with no method and no limitations section, is marketing. Treat the numbers as unverified.
10. The idea is not new or informal. Model cards were proposed in a 2019 paper by Margaret Mitchell and colleagues, and datasheets for datasets in a 2018 paper by Timnit Gebru and colleagues. Both are conventions, widely adopted and nowhere enforced.

■ 51.8.5 TECHNICAL – the engineer's version

1. Licences differ far more than people assume, and "open" is not one thing. Apache 2.0 and MIT are permissive. Llama's community licence adds an acceptable-use policy and a threshold on monthly active users. Some research-only licences forbid commercial use entirely.
2. "Open weights" means you can download the parameters. It does not mean open source: the training data and training code are usually not released. Say open-weights when that is what you mean.
3. Gated repositories require accepting terms with your account before the files resolve, so automated pipelines need a token with the acceptance already recorded.

■ 51.8.6 WORDS – remember these

Model card — the honest description page — a structured README covering data, licence, intended use, limitations and evaluations. Safetensors — a weight file that cannot run code — a simple tensor container format, preferred over pickle-based .bin. Open weights — you can download the numbers — distinct from open source, which would include data and training code. Perplexity — how confused the model is by real text — exponentiated average negative log-likelihood; used here to price quantization damage. Gated repository — you must accept terms first — access controlled by account agreement on the hub.

51.9 Building something small from scratch, honestly

■ 51.9.1 PLAIN – in simple words

1. You can train your own language model on a laptop. It is a genuinely good afternoon's work, and it will teach you more than ten articles.
2. Realistically, on a laptop, you are training something between one and fifty million parameters on a few tens of millions of characters of text.
3. That is between one thousandth and one ten-thousandth of the size of the models you use every day.
4. What it will do: produce text with the correct shape. Sentences that look like sentences. Correct punctuation. Local coherence for a line or two.
5. If you train it on Shakespeare, it produces convincing Shakespeare-shaped nonsense with character names and stage directions in the right places.
6. What it will not do: know any facts, follow an instruction, answer a question, hold a conversation, or reason.
7. Those abilities appear only with vastly more data and a separate instruction-tuning stage afterwards.
8. Do it exactly once, as an education. Then never ship it, because a downloadable open model is thousands of times better and free.

■ 51.9.2 PLAIN – a picture in your head

1. Think of a child copying the appearance of adult handwriting before they can spell.
2. The loops are right. The spacing is right. The line has the rhythm of writing. From across the room it looks like a letter.
3. Your tiny model is at exactly that stage, and getting there takes an hour.
4. Learning to spell is the next thousand hours, and learning to write something worth reading is the thousand after that.

Where this comparison breaks: the child is learning towards meaning the whole time. The model has no such trajectory. Scale does not turn imitation into understanding by itself; it turns imitation into much better imitation, which turns out to be enough for a surprising number of tasks, and this distinction is exactly where thoughtful people still disagree.

■ 51.9.3 PLAIN – a worked example

1. Concrete plan, with real numbers.
2. Data: the collected works of Shakespeare, about 1.1 million characters, which is a standard toy dataset.
3. Model: a transformer with 6 layers, 6 attention heads, 384 hidden dimensions, character-level vocabulary. That is roughly 10 million parameters.
4. Training: a few thousand steps. On an Apple silicon laptop this takes on the order of 10 to 30 minutes. On a rented graphics card, a couple of minutes.
5. Result: text like this in shape, though the words will be invented.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

6. Every convention is right. Speaker names, colons, line breaks, capitalized verse. No sentence means anything.
7. Now the honest scaling. To get factual knowledge you need billions of tokens, not one million characters.
8. The compute-optimal ratio from the 2022 Chinchilla paper is roughly 20 training tokens per parameter. A useful 1-billion-parameter model implies about 20 billion tokens, which is not a laptop job.

■ 51.9.4 PLAIN – what is really happening inside

1. The canonical implementations to read, by name, so you can find them.
2. minGPT, by Andrej Karpathy, 2020: a few hundred lines, written to be read rather than run fast.
3. nanoGPT, by the same author, January 2023: the practical version, which reproduces GPT-2 at 124 million parameters.
4. The video “Let’s build GPT: from scratch, in code, spelled out”, 2023, which builds the whole thing live in about two hours.
5. llm.c, 2024: the same training in plain C and CUDA, with no framework at all, useful for seeing where the time actually goes.
6. nanochat, October 2025: the full pipeline including tokenizer, pre-training, fine-tuning, evaluation and a chat interface, in one repository.
7. The nanochat numbers are the useful ones for calibration. Its own documentation reports training a model of GPT-2 capability for about 48 dollars, roughly two to four hours on a node of eight H100 cards, against the roughly 43,000 dollars that training GPT-2 cost in 2019.
8. That fall of nearly a thousand times in six years is one of the clearest single facts about this field, and it is about hardware, software and recipes together, not about any one breakthrough.

■ 51.9.5 TECHNICAL – the engineer’s version

1. What a laptop-scale run looks like against the real thing:

Property	Laptop toy	Small open model
Parameters	10 million	8 billion
Training tokens	1 million chars	15 trillion
Hardware	1 laptop chip	thousands of GPUs
Wall-clock	20 minutes	weeks
Can follow orders	no	yes

2. The alternative, and usually the right one: fine-tune an existing open model instead of training one.
3. Full fine-tuning of an 8B model needs the 165 GiB from section 51.1, so it is a multi-GPU job.
4. Parameter-efficient methods avoid that. LoRA, from a 2021 Microsoft paper, freezes the model and trains small low-rank matrices alongside it, typically 0.1 to 1 percent of the parameters.
5. QLoRA, 2023, adds 4-bit quantization of the frozen base, which brought fine-tuning a 7B model onto a single 24 GB card, and a 70B model onto a single 80 GB card.
6. When fine-tuning is the right answer: you need a fixed output format, a house style, a domain vocabulary, or you want to make a small cheap model imitate a large expensive one on one narrow task.
7. When it is the wrong answer: you want the model to know new facts. Facts change, fine-tuning bakes them in, and the model will still state the old ones confidently. Use retrieval, which section 51.11 covers.

8. The practical threshold: below roughly 1,000 good examples, prompt engineering and few-shot examples usually beat fine-tuning. Above roughly 10,000, fine-tuning usually wins.
9. Evaluate before and after on a held-out set you wrote yourself. Fine-tuning frequently improves the target task and quietly degrades everything else, a phenomenon called catastrophic forgetting.

■ 51.9.6 WORDS – remember these

Pre-training — the long expensive stage — next-token prediction over a very large corpus, producing a base model. Base model — a text continuer, not an assistant — the pre-trained model before instruction tuning. Chinchilla ratio — how much text per parameter — about 20 training tokens per parameter for compute-optimal training, from the 2022 paper. LoRA — training a small patch instead of the whole model — low-rank adapter matrices added to frozen weights. Catastrophic forgetting — getting better at one thing and worse at everything else — degradation of unrelated capabilities after narrow fine-tuning.

51.10 Prompting as an engineering practice, not a trick

■ 51.10.1 PLAIN – in simple words

1. A prompt is just the text you send. There is no magic word.
2. What works is the same thing that works when briefing a competent new colleague who cannot ask you questions.
3. Be specific about the task, the audience, the length and the format.
4. Show an example of what you want, because one example beats a paragraph of description.
5. Say what the output must look like, exactly, especially if a program will read it.
6. For multi-step problems, let the model work through the steps rather than demanding the answer immediately.
7. Break big tasks into several calls instead of one enormous one.
8. And now the honest part: the same request phrased two ways can give noticeably different quality. That is a real weakness of the technology, not a skill you lack.

■ 51.10.2 PLAIN – a picture in your head

1. Think of writing a work order for a contractor who will do exactly what the paper says and will never phone you back.
2. “Fix the bathroom” gets you something. Possibly not the thing you wanted.
3. “Replace the tap with this model, keep the existing pipework, finish by Friday, leave the old tap in the box” gets you the thing you wanted.
4. Adding a photograph of a finished job you liked works better still.
5. The contractor is skilled. The failure was in the brief.

Where this comparison breaks: a real contractor who received an ambiguous brief would either ask or use judgement informed by having seen your house. The model has no judgement about your situation and cannot ask, so it fills gaps with the statistically ordinary choice, which may be nothing like your case.

■ 51.10.3 PLAIN – a worked example

1. Five real improvements. In each pair, the second version is not longer for the sake of it; each addition removes one specific ambiguity.

1 BEFORE: Summarize this.

AFTER: Summarize the text below in 5 bullet points for a

non-technical manager. Each bullet under 20 words.
Do not mention anything not in the text.

2 BEFORE: Write a regex for emails.

AFTER: Write a Python regex matching the addresses in the list below and rejecting the ones after "BAD:". Return only the pattern, no explanation.

3 BEFORE: Is this contract clause risky?

AFTER: Read the clause. List each obligation it places on the buyer, then rate each Low/Medium/High risk with one reason. If the clause is ambiguous, say so.

4 BEFORE: Extract the data.

AFTER: Return JSON only, matching exactly this shape:
{"invoice_no": string, "total": number,
"currency": string, "date": "YYYY-MM-DD"}
Use null for any field not present in the text.

5 BEFORE: What is 17% of 4,382 plus the VAT?

AFTER: Work step by step. State each arithmetic step on its own line, then give the final number. Use the calculator tool for each multiplication.

2. Improvement 2 supplies examples of both classes, which is what few-shot prompting means.
3. Improvement 4 pins the exact schema, which is the difference between code that parses and code that crashes at 3 a.m.
4. Improvement 5 does two things: asks for steps rather than a single leap, and moves the arithmetic to a tool, for reasons section 51.15 explains.

■ 51.10.4 PLAIN - what is really happening inside

1. Why examples work: the model is completing a pattern. Two worked examples define the pattern far more sharply than a description of it.
2. Zero-shot means no examples, just an instruction. Few-shot means two to five worked examples in the prompt. Few-shot costs tokens on every call and is usually worth it for format compliance.
3. Why step-by-step working helps: each token the model writes becomes input for the next token. Written reasoning is literally extra computation, and it gives the model somewhere to put intermediate results, since it has no scratchpad of its own.
4. This was formalized as chain-of-thought prompting in 2022, and the effect is large on multi-step arithmetic and logic.
5. When it does not help: simple lookups, classification, and formatting tasks, where it adds cost and latency for nothing.
6. There is also a 2026 research finding worth knowing: more reasoning is not monotonically better. Accuracy against reasoning length often follows an inverted U, and past a point extra thinking flips correct answers to wrong ones. That result is active research, not settled.
7. Recent reasoning models do this internally and are trained for it, so instructing them to “think step by step” can be redundant or harmful. Read the provider’s guidance for the specific model.

■ 51.10.5 TECHNICAL – the engineer’s version

1. Treat prompts as versioned artifacts. They belong in the repository, with a test suite, not pasted into a dashboard.
2. Prompt sensitivity is measurable and is a genuine limitation. Reported effects in the literature include changes in accuracy from reordering few-shot examples, from formatting choices such as separators, and from the position of the question relative to a long document.
3. The position effect has a name: models attend most reliably to the beginning and end of a long context and least reliably to the middle, described as “lost in the middle” in a 2023 paper.
4. Put the question after the documents, and put the most important document last.
5. Techniques with reasonable evidence: few-shot examples, explicit output schemas, decomposition into separate calls, self-consistency by sampling several answers and taking the majority, and constrained decoding when the output must parse.
6. Techniques that are mostly folklore: politeness, offering tips, threatening, role-play preambles such as “you are a world-class expert”, and long lists of adjectives. Some show small effects in some studies and vanish in others.
7. The honest summary: prompt engineering is real but shallow. Its ceiling is low compared with fixing your retrieval, your data or your task decomposition.

■ 51.10.6 WORDS – remember these

Zero-shot — no examples given — instruction-only prompting. Few-shot — a handful of worked examples — in-context learning from demonstrations in the prompt. Chain of thought — asking it to show working — eliciting intermediate reasoning tokens before the answer. Self-consistency — ask several times and take the majority — sampling multiple reasoning paths and voting. Prompt sensitivity — small wording changes, different quality — measured variance in output quality under semantically equivalent rephrasing.

51.11 RAG: giving a model your documents

■ 51.11.1 PLAIN – in simple words

1. A model knows only what was in its training data, which stopped at some date, and which never included your company’s files.
2. You cannot fix that by asking nicely. The information is genuinely not inside the weights.
3. But you can put the relevant text into the question. The model reads its context, so text you paste in is text it can use.
4. Doing that automatically is called retrieval-augmented generation, or RAG.
5. The pipeline is: cut the documents into pieces, turn each piece into a vector of numbers, store them, turn the question into a vector too, find the closest pieces, paste them into the prompt, and generate.
6. The vector is the embedding from Chapter 47: a list of numbers where similar meanings sit close together.
7. Every part of that pipeline can fail, and when the answer is wrong it is usually retrieval that failed, not the model.

■ 51.11.2 PLAIN – a picture in your head

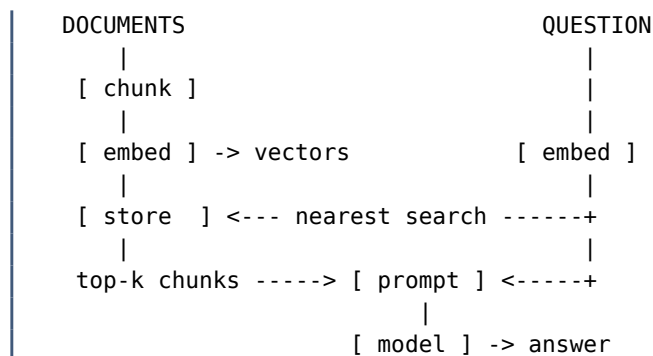
1. Think of an exam where the candidate may bring notes, but only one page.
2. Someone else chooses which page, from a filing cabinet, in one second, based on the question.
3. If they bring the right page, the candidate looks brilliant.

4. If they bring a page about the right topic but the wrong year, the candidate confidently gives last year's answer.
5. If they bring six pages of near-misses, the candidate gets distracted and the good page gets lost among them.
6. The candidate's ability is not what varies between these cases. The filing clerk's is.

Where this comparison breaks: a real candidate would say "these notes do not cover this question". A model usually will not. It will answer from the notes if it can, and from habit if it cannot, and both answers look equally confident.

■ 51.11.3 PLAIN – a worked example

1. The pipeline, drawn out:



2. Real sizes for a small system: 500 documents, average 8 pages each, cut into chunks of about 500 tokens with 50 tokens of overlap.
3. That gives roughly 30,000 chunks. Embedded at 1,536 dimensions in 4-byte floats, the vectors occupy $30,000 \times 1,536 \times 4$ bytes, which is 184 MB.
4. Embedding cost, at a published rate of about 0.02 dollars per million tokens for a small embedding model: 15 million tokens costs 30 cents. Embedding is cheap; that is not where the money goes.
5. At query time you embed one question, find the 5 nearest chunks, and paste about 2,500 tokens into the prompt.
6. That 2,500 tokens is charged on every single question, which is why section 51.5's cost table had retrieved context in it.

■ 51.11.4 PLAIN – what is really happening inside

1. Chunking first, because it decides more than any other choice.
2. Chunk too small and each piece loses the context that made it meaningful. A sentence saying "this does not apply in India" is useless alone.
3. Chunk too large and the embedding becomes an average of several topics, which is close to nothing in particular, so it matches nothing well.
4. Strategies, in increasing order of effort: fixed size with overlap; splitting on structure such as headings and paragraphs; splitting on semantic shifts; and adding a short summary of the parent document to every chunk so it carries its own context.
5. Overlap exists so that a fact sitting on a boundary appears whole in at least one chunk. Ten percent is a common default.
6. Then embedding. An embedding model is a separate, smaller model that maps text to a fixed-length vector. Both the documents and the query must use the same one, and changing it means re-embedding everything.

7. Then storage and search. With 30,000 chunks you could compare the query to every vector, which is exact and fast enough. With 30 million you cannot.
8. So vector databases use approximate nearest neighbour search, and the dominant method is called HNSW.
9. Here is HNSW in plain words. Build a graph where each vector is a point joined to a few of its neighbours. Then build a second, sparser graph above it containing a random subset, and a third above that, and so on.
10. To search, start at the top layer, walk greedily to the closest point you can reach, then drop a layer and repeat from there.
11. The top layers move you across the whole space in a few hops. The bottom layer does the fine work. It is an express train, then a local train, then walking.
12. It is approximate: it can miss the true nearest neighbour. You trade recall against speed with a parameter, usually called `ef_search`.

■ 51.11.5 TECHNICAL – the engineer’s version

1. RAG was named in a 2020 paper by Patrick Lewis and colleagues at Facebook AI Research. HNSW is from a 2016 paper by Yury Malkov and Dmitry Yashunin. FAISS, the library that made billion-scale search routine, came from Facebook AI Research in 2017.
2. Vector store choices in 2026, honestly ranked by how often they are the right answer: pgvector inside an existing Postgres for anything up to tens of millions of vectors; Qdrant, Milvus or Weaviate when you need dedicated scale or filtering; Pinecone when you want it fully managed; Chroma or FAISS for prototypes and local work.
3. Pure vector search fails on exact tokens: part numbers, error codes, proper nouns and negations. Keyword search fails on paraphrase. Hybrid search runs both and fuses the results.
4. The standard fusion method is reciprocal rank fusion, from a 2009 paper by Cormack, Clarke and Buettcher: score each document by the sum of one over (a constant plus its rank) in each list.
5. It needs no score calibration, which is why it is used.
6. Re-ranking is a second, more expensive model that scores query-document pairs jointly, called a cross-encoder. Retrieve 50 candidates cheaply, re-rank to the best 5, pass those on. This is usually the single highest return improvement in a RAG system.
7. Evaluate retrieval separately from generation, always. Retrieval metrics: recall at k, mean reciprocal rank, normalized discounted cumulative gain. Generation metrics: groundedness, meaning every claim is supported by a retrieved chunk, and answer relevance.
8. The three honest failure modes:

Failure	Symptom	Usual fix
Retrieval miss	confident wrong answer	hybrid search, chunking
Crowded context	vague, hedged answer	fewer chunks, re-rank
Ignored context	contradicts the source	cite-and-quote prompt

9. The third one deserves a note. Models do sometimes override retrieved text with training-time knowledge, especially when the retrieved text is surprising. Requiring the model to quote the supporting sentence before answering reduces it measurably and makes the failure visible when it happens.

■ 51.11.6 WORDS – remember these

RAG — fetch the right text and paste it in — retrieval-augmented generation, named in a 2020 paper. Chunk — one searchable piece of a document — a token-bounded span, usually with overlap, that is embedded as a unit. Embedding model — turns text into a vector of meaning — a bi-encoder producing a fixed-length dense representation. HNSW — express train, local train, then walking — hierarchical navigable small world graph for approximate nearest neighbour search. Hybrid search — keywords and meaning together — dense plus sparse retrieval fused, commonly by reciprocal rank fusion. Re-ranker — a second opinion on the shortlist — a cross-encoder scoring query and document jointly.

51.12 Tool use and function calling

■ 51.12.1 PLAIN – in simple words

1. A model produces text. Only text. It cannot press a button or call a web service.
2. It writes down a request, in a structured form you specified, saying which function it wants and with what arguments.
3. Your program reads that request, decides whether to run it, runs it, and puts the result back into the conversation as a new message.
4. The model then continues, now with the answer in front of it.
5. That is the whole mechanism. The model never executes anything. Your code does.
6. Which means every tool you offer is a door in your system that a piece of text can ask to open. That is a security boundary, and it must be treated like one.

■ 51.12.2 PLAIN – a picture in your head

1. Think of a consultant behind a glass wall with no phone and no keyboard.
2. They can write notes and slide them under the glass.
3. A note might say: look up order 4471 and tell me the status.
4. You read it, decide whether that is a reasonable thing to ask, look it up, and slide the answer back.
5. At no point did the consultant touch your systems. They asked. You acted.
6. If you slide back whatever any note asks for, without checking, the glass wall is decorative.

Where this comparison breaks: a human consultant has intentions of their own. A model has none, but it can be steered by any text that reaches its context, including text inside the very documents it is reading for you. So the note under the glass may have been dictated by a stranger.

■ 51.12.3 PLAIN – a worked example

1. A complete exchange, in four steps. First you declare the tool.

```
"tools": [{
  "type": "function",
  "function": {
    "name": "get_order_status",
    "description": "Look up the status of one order.",
    "parameters": {
      "type": "object",
      "properties": {
        "order_id": {"type": "string",
                      "description": "Order number"}
      }
    },
    "required": ["order_id"]
  }
}]
```

```

    }
  }
}

```

- Second, the user asks “where is order 4471”, and the model replies with a request rather than an answer:

```

"message": {
  "role": "assistant",
  "content": null,
  "tool_calls": [{
    "id": "call_a1",
    "type": "function",
    "function": {
      "name": "get_order_status",
      "arguments": "{\"order_id\": \"4471\"}"
    }
  ]
},
"finish_reason": "tool_calls"

```

- Third, your code validates the arguments, runs the real lookup, and sends the whole conversation back with one extra message:

```

{"role": "tool",
 "tool_call_id": "call_a1",
 "content": "{\"status\": \"shipped\",
            \"eta\": \"2026-08-15\"}"}

```

- Fourth, the model produces the final answer: “Order 4471 shipped and should arrive on 15 August 2026.”

■ 51.12.4 PLAIN – what is really happening inside

- The tool declarations are inserted into the model’s context as text, usually as a schema in the system section. They cost tokens on every call.
- The model was trained to emit a particular token pattern when it wants a tool. The serving layer detects that pattern and returns it as structured JSON instead of prose.
- Many providers use constrained decoding here, so the arguments are guaranteed to match the schema. Others do not, and you must handle malformed JSON.
- The model chooses the tool by matching your description text against the user’s request. So the description is a prompt, and vague descriptions cause wrong tool choices.
- Now the security part, stated plainly. The model decides which tool to call based on text in its context. Some of that text came from your user. Some may have come from a document, a web page, or an email.
- Therefore any text the model can read can attempt to trigger any tool the model can call. There is no separation between instructions and data.
- So: validate every argument in your own code as if it came from an anonymous internet user, because effectively it did. Enforce permissions on the caller, not on the model.
- Never pass model output into a shell, a SQL string or a file path without the same checks you would use on a web form.
- Prefer narrow tools over general ones. `get_order_status(order_id)` is safe to expose. `run_sql(query)` is not, however convenient it looks.

■ 51.12.5 TECHNICAL – the engineer’s version

1. History: OpenAI shipped function calling in June 2023. The research precursor was Toolformer, published in February 2023. Anthropic released the Model Context Protocol in November 2024 to standardize how tools are described and discovered; OpenAI adopted it in March 2025, and its specification is dated by release, with 2026-07-28 current in August 2026.
2. Cost note: every tool round trip re-sends the entire conversation. A five-tool-call task sends the context six times. This is the main reason agent workloads are expensive, and the main reason prompt caching matters.
3. Threat model, in the vocabulary that has settled: the dangerous combination is private data access, exposure to untrusted content, and an ability to communicate outward. Simon Willison named this the “lethal trifecta” in June 2025. Remove any one of the three and exfiltration becomes much harder.
4. Practical controls: allowlist tools per session, require human confirmation for writes, run tools with least privilege, log every call with arguments, and put outbound network egress behind a proxy you control.

■ 51.12.6 WORDS – remember these

Function calling — the model asks, you do it — structured tool-request output matching a declared JSON schema. Tool result — the answer handed back — a message with role “tool” carrying the execution output into the context. Constrained decoding — only valid output can be produced — logit masking against a grammar or schema. MCP — a common plug for tools — Model Context Protocol, from Anthropic, November 2024, adopted broadly in 2025 and 2026. Lethal trifecta — the dangerous combination — private data, untrusted content and outbound communication in one agent.

51.13 Agents: what the word actually means

■ 51.13.1 PLAIN – in simple words

1. An agent is a loop, not a product and not a personality.
2. The loop is: the model decides what to do next, calls a tool, sees the result, and decides again, until it says it is finished or you stop it.
3. A single API call is one question and one answer. An agent is many calls in a row, where the model chooses what the next call contains.
4. RAG is different again: retrieval happens once, before the model runs, and the model does not choose it.
5. So the ladder is: one call, then one call with tools, then a loop with tools, which is an agent.
6. The problem is arithmetic. Steps multiply. If each step is 95 percent reliable, twenty steps together are 36 percent reliable.
7. That is not a detail to be tuned away. It is the central engineering fact about agents.

■ 51.13.2 PLAIN – a picture in your head

1. Think of a relay race where each runner hands over a written message rather than a baton.
2. Each runner is fast and mostly accurate, but occasionally mishears a word.
3. Over forty, it arrives as something else entirely, and nobody along the way noticed anything wrong.
4. Now add this: every runner is completely confident that what they passed on was correct.
5. The fix is not faster runners. It is checkpoints where the written message is compared against the original.

Where this comparison breaks: a garbled message is obviously garbled at the end. An agent's wrong result usually looks exactly like a right one, because the same fluency that makes the output readable makes the error invisible.

■ 51.13.3 PLAIN – a worked example

1. The compounding arithmetic, spelled out. This is just multiplication.

Per-step success	10 steps	30 steps
90%	34.9%	4.2%
95%	59.9%	21.5%
99%	90.4%	74.0%
99.9%	99.0%	97.0%

2. Read the 95 percent row. A step that fails once in twenty sounds excellent and produces a task that fails four times in five at thirty steps.
3. Now read the 99.9 percent row. To run thirty steps reliably you need per-step reliability that no language model currently offers unaided.
4. So the practical designs all do the same thing: reduce the number of steps, or make each step verifiable so failures are caught rather than compounded.
5. A verifiable step is one where something other than the model can check the result. Code that compiles. Tests that pass. A schema that validates. A number that reconciles.
6. This is exactly why coding agents work better than most other kinds. The compiler and the test suite are an independent judge on every step.

■ 51.13.4 PLAIN – what is really happening inside

1. The common patterns, named, so you recognize them in documentation.
2. ReAct, from a 2022 paper, interleaves reasoning and acting: think, act, observe, repeat. It is the default shape of most agent frameworks.

```
Thought: I need the order status.
Action: get_order_status("4471")
Observation: {"status": "shipped", "eta": "2026-08-15"}
Thought: I now have what I need.
Answer: It shipped and arrives on 15 August.
```

3. Plan-and-execute makes a full plan first, then carries out the steps. It is cheaper and more predictable, and worse at recovering when reality differs from the plan.
4. Reflection adds a criticism step: the model reviews its own output against the goal and revises. It helps when there is an objective standard to check against, and mostly produces confident agreement when there is not.
5. Multi-agent designs give different roles to different model calls, such as a planner, a coder and a reviewer, with messages between them.
6. The honest position on multi-agent systems as of 2026: they demonstrate well, they are expensive, and independent evidence that they beat one well-prompted model with good tools on ordinary tasks is thin. Some published analyses argue that errors compound faster across agents, not slower.
7. What actually makes agents work in production, in order of importance: narrow scope, verifiable steps, a hard step limit, a budget limit, retries with different phrasing, and a human checkpoint before anything irreversible.

■ 51.13.5 TECHNICAL – the engineer’s version

1. Separate what is established from what is sold.
2. Established: tool-calling loops work well for bounded tasks with checkable outputs; coding assistants that run tests are genuinely useful; retrieval plus a small number of tool calls is a solid production pattern.
3. Active research: long-horizon planning, self-correction without an external verifier, reliable multi-agent coordination, and memory that survives across sessions in a useful way.
4. Marketing claim: that an agent can be given a business objective and left alone. No published evidence supports this for open-ended tasks, and the compounding arithmetic above argues against it directly.
5. Engineering controls that belong in any agent you deploy:

Control	What it prevents
Max steps and max tokens	runaway loops and bills
Tool allowlist per task	unexpected capability
Idempotency keys	duplicate side effects
Human approval on writes	irreversible mistakes
Full trace logging	undebuggable failures

■ 51.13.6 WORDS – remember these

Agent — a loop where the model chooses the next action — iterative plan-act-observe with tools and a stopping condition. ReAct — think, act, observe, repeat — the interleaved reasoning and acting pattern from a 2022 paper. Reflection — the model criticizing its own draft — a self-review step, useful mainly with an external standard to check against. Compounding error — small failures multiplying — end-to-end success equals per-step reliability raised to the number of steps. Human in the loop — a person approves the risky bit — a mandatory confirmation gate before irreversible actions.

51.14 Memory, and why a model with none appears to have some

■ 51.14.1 PLAIN – in simple words

1. A language model has no memory between calls. None at all.
2. The only state it has is the text currently in its context window.
3. When a chat application seems to remember yesterday, an ordinary database remembered it and pasted it back in.
4. Every turn of a conversation resends the whole conversation so far.
5. That has a cost consequence people miss: turn twenty is not the same price as turn one, because turn twenty carries nineteen turns of history.
6. Everything sold as “memory” is an application feature: store some text, retrieve the relevant bits later, put them in the prompt.
7. So memory is just retrieval, from section 51.11, pointed at your own past conversations instead of your documents.

■ 51.14.2 PLAIN – a picture in your head

1. Imagine a brilliant advisor with complete amnesia every night.
2. Each morning you hand them a folder containing everything relevant from before, and they work perfectly.

3. They are not remembering. You are.
4. If your folder is wrong, they are confidently wrong, and they have no way to notice, because the folder is their entire world.

Where this comparison breaks: the advisor would know they had amnesia. The model has no sense of a gap. Text it was never given simply does not exist for it, and it will fill the space with something plausible rather than reporting the absence.

■ 51.14.3 PLAIN – a worked example

1. A ten-turn conversation with 500 tokens per turn, no caching:

Turn	Input tokens	Cumulative input
1	500	500
5	2,500	7,500
10	5,000	27,500

2. Ten turns of 500 tokens each is 5,000 tokens of conversation, but you paid for 27,500 input tokens, which is five and a half times more.
3. The growth is quadratic in the number of turns. Doubling the turns quadruples the input bill.
4. Three fixes, all standard: cache the stable prefix, summarize old turns into a short note, or drop old turns entirely past a window.

■ 51.14.4 PLAIN – what is really happening inside

1. The common designs, and what each is really doing.
2. Full history: send everything. Simple, exact, and quadratic in cost. Fine for short sessions.
3. Sliding window: keep the last N turns. Cheap and predictable; loses anything older, silently.
4. Running summary: after every few turns, ask the model to compress the older part into a paragraph, and send the summary plus recent turns. Cheap, and lossy in ways you cannot predict.
5. Retrieved memory: store every turn in a vector database and retrieve the few most relevant to the current message. Scales, and inherits every failure mode from section 51.11.
6. Structured facts: extract specific fields, such as the user's name, preferences and account, into an ordinary database, and inject them as text. This is the most reliable and the least fashionable.
7. The limits are the same in every case: whatever is not retrieved does not exist, and nothing about this is the model remembering.

■ 51.14.5 TECHNICAL – the engineer's version

1. Context is a hard bound, not a soft one. Exceeding it produces an error or silent truncation, and truncation usually removes the oldest tokens, which is where your system prompt might be.
2. Put the system prompt where truncation cannot reach it, and validate token counts before sending, using the model's own tokenizer.
3. Structured extraction beats free-text memory for anything you will filter or act on. "The user's plan is enterprise" belongs in a column, not in prose inside a vector store.

■ 51.14.6 WORDS – remember these

Context window — everything the model can see right now — the maximum token sequence, its only state. Conversation history — the transcript resent every turn — the messages array, billed in full on each call. Summarization memory — squash the old part — periodic lossy compression of

earlier turns into a short note. Structured memory — facts in a proper database — extracted fields injected as text, the most reliable design.

51.15 The limits, stated plainly

■ 51.15.1 PLAIN – in simple words

1. Hallucination is when a model states something false with the same confidence it states something true.
2. It is not a bug that was left in. It follows from what the model is.
3. The model was trained to produce likely continuations of text. A confident, fluent, plausible answer is a likely continuation whether or not it is true.
4. There is also no internal flag for “I do not know”. The model always produces a next token, and the machinery for producing a true one and a false one is identical.
5. Asking it to be accurate does not help, because it is not choosing to be inaccurate.
6. What helps is putting truth in front of it, or letting a tool check.
7. And this is the important sentence: confident wrongness is the default failure mode. Errors do not arrive marked as errors.

■ 51.15.2 PLAIN – a picture in your head

1. Think of an extremely well-read person answering at a dinner party, at speed, who considers not answering to be a failure.
2. Ninety percent of the time they are right and it is delightful.
3. Ten percent of the time they produce the same tone, the same fluency, the same specific detail, and it is invented.
4. You cannot tell the two apart from the outside, and neither can they.
5. Now imagine the party is a legal filing, a medical note or a payment.

Where this comparison breaks: the dinner guest could stop and say “I am not sure”. The model can be trained to say that more often, and modern ones do, but it is still producing the phrase because such phrases followed such contexts in training, not because it inspected its own uncertainty and found it high.

■ 51.15.3 PLAIN – a worked example

1. What actually reduces hallucination, and what does not:

Intervention	Effect
Retrieval with quoted source	large reduction
Tool for the checkable part	removes that class
Constrained output schema	removes format errors
Second pass verification	moderate reduction
Saying “be accurate”	negligible
Saying “do not hallucinate”	negligible

2. A concrete pair. Ask for a citation for a claim and you may get a real- looking reference that does not exist, with plausible authors, journal and year.
3. Give the model five real abstracts and require it to quote from one, and the invented citation disappears, because the correct behaviour is now the easy continuation.

4. Now three things the model cannot know, whatever you do.
5. Anything after its training cut-off. It will still answer, using the world as it was.
6. Anything private: your files, your database, this morning's ticket. Unless you put it in the context, it is not there.
7. Its own weights and internals. A model's account of why it answered is a plausible story generated after the fact, not a readout. This is established: self-reported reasoning does not reliably match the computation.

■ 51.15.4 PLAIN - what is really happening inside

1. Why counting and arithmetic fail, connecting to Chapter 47.
2. The model never sees letters. It sees tokens, which are chunks of several characters, and the chunks were chosen by compression statistics.
3. So "strawberry" may arrive as two or three tokens, and the letter r is not individually visible in any of them. Counting them is like counting bricks in a photograph of a wall taken from a mile away.
4. Long arithmetic fails for a second, separate reason: the model does a fixed amount of computation per token and has no scratchpad.
5. Multiplying two six-digit numbers requires carrying intermediate results through many steps. With no working memory, the only place to put them is the output itself.
6. That is exactly why writing out the steps helps, and why handing the sum to a calculator tool helps far more.
7. Now prompt injection, which is the unsolved one. The model has a single channel. Instructions and data arrive as the same tokens.
8. So text inside a document, a web page, an email or a code comment can address the model directly, and the model has no reliable way to tell that this text was not from you.
9. The term was coined by Simon Willison in September 2022. As of August 2026 it remains unsolved in the general case, and security guidance from national agencies now assumes it rather than promising a fix.
10. Jailbreaking is a related but different thing: the user of a model trying to get past its own trained refusals. Injection is a third party attacking your application through content it processes.
11. Sycophancy is the tendency to agree with the user, to soften a correct answer under pushback, and to praise a bad idea. It comes from training on human preference, and humans prefer agreement.

■ 51.15.5 TECHNICAL - the engineer's version

1. Hallucination is best understood as a calibration problem. A well-calibrated model would assign confidence matching its accuracy and abstain below a threshold.
2. A 2025 paper from OpenAI, "Why Language Models Hallucinate", argues the incentive is structural: both pre-training objectives and benchmark scoring reward a guess over an admission of ignorance, exactly as a multiple-choice exam with no penalty for wrong answers does.
3. That implies the fix is partly in evaluation design: score abstention as better than a wrong answer, and models trained against that scoring abstain more. This is active work, not a solved problem.
4. Uncertainty signals you can actually use: token log-probabilities where the provider exposes them, agreement across several samples at nonzero temperature, and agreement between two different models. All are weak but better than nothing.
5. Prompt injection defenses that reduce risk without solving it: strict separation of trusted and untrusted content with explicit markers, least-privilege tools, breaking the lethal trifecta by removing outbound communication or private data access from the same agent, output filtering, and human confirmation on consequential actions.

6. Defenses that do not work: instructing the model to ignore instructions in documents, and detecting injections with another language model, which is itself injectable.
7. What all of this means if you are building:

Rule	Because
Never auto-execute output	injection is unsolved
Verify facts with a source	fluency is not accuracy
Design for wrong answers	they are unmarked
Log inputs and outputs	you will need the trace
Keep a human on writes	reversal is expensive

■ 51.15.6 WORDS - remember these

Hallucination — confident invention — generation of unsupported content, a calibration failure rather than a discrete bug. Calibration — confidence matching accuracy — the property a model would need to know when to abstain. Prompt injection — instructions smuggled in through data — an architectural consequence of one shared channel; unsolved as of August 2026. Jailbreak — getting past the model's own rules — user-side circumvention of trained refusals, distinct from injection. Sycophancy — agreeing because you pushed — preference-trained bias toward agreement and flattery. Knowledge cut-off — the date the world stopped for it — the end of the training data, after which nothing is known.

51.16 The gaps you did not ask about but need

■ 51.16.1 PLAIN - in simple words

1. Six things you will meet in week one that nobody puts in the introduction.
2. Multimodality: how a picture becomes something a text model can read.
3. Reasoning models: models trained to think at length before answering, and billed for the thinking.
4. Long context: how a model that started with 2,000 tokens now takes a million, and why that is less useful than it sounds.
5. Cards: a model card describes one model; a system card describes a whole deployed product including its safety testing.
6. The safety vocabulary: alignment, red-teaming, refusal, guardrails, evals.
7. Distillation: making a small model imitate a big one.

■ 51.16.2 PLAIN - a picture in your head

1. For images, think of a photograph cut into a grid of small squares.
2. Each square is described as a list of numbers, in the same kind of language the model already uses for words.
3. The squares are then laid in a row alongside the text tokens, and the model reads the whole row as one sequence.

Where this comparison breaks: the translation is learned, not defined, and it is lossy in ways nobody chose. Fine print, precise counts and exact spatial relations survive badly, which is why models misread charts and miscount objects far more often than their fluent descriptions suggest.

■ 51.16.3 PLAIN – a worked example

1. Rough token costs of non-text inputs, which is what you will be billed on:

Input	Becomes roughly
1024 x 1024 image	750 to 1,600 tokens
One page of a PDF	1,000 to 2,000 tokens
One minute of speech	150 words, plus overhead

2. So ten screenshots can cost more than a long document, and a reasoning model may bill 4,000 hidden thinking tokens behind a 200-token answer.

■ 51.16.4 PLAIN – what is really happening inside

1. Images: a vision encoder cuts the picture into patches, embeds each patch, and a small projection layer maps those embeddings into the same space as token embeddings. The transformer then treats them as ordinary positions.
2. Audio: either a spectrogram is patched the same way, or a separate codec turns sound into discrete tokens from a learned audio vocabulary.
3. Reasoning models: trained with reinforcement learning to produce long internal chains of thought before answering. Established fact: this raises accuracy substantially on mathematics, competitive programming and multi-step logic.
4. Active research: how far it generalizes, and the 2026 finding that past a point extra thinking reduces accuracy.
5. Long context is achieved by changing how positions are encoded, most commonly by scaling rotary position embeddings, plus attention variants that avoid materializing a full attention matrix.
6. The honest note: a large advertised context is a capacity, not a competence. Retrieval quality inside a long context degrades with distance from the ends, and simple needle-finding benchmarks overstate real performance on tasks needing several facts combined.
7. Distillation, from a 2015 paper by Hinton and colleagues: train a small student model on the outputs of a large teacher. Most small fast models available today were made this way.

■ 51.16.5 TECHNICAL – the engineer’s version

1. Alignment, defined concretely rather than as a slogan: the engineering work of making a model’s outputs conform to a written specification of intended behaviour, implemented through preference training and rule-based methods, and measured by evaluations. It is a practice with artifacts, not an aspiration.
2. Red-teaming: adversarial testing by humans or automated attackers, aimed at producing the failures the model is meant not to produce. Findings feed back into training and into guardrails.
3. Refusal: the trained behaviour of declining a request. Both over-refusal and under-refusal are measured, and the trade between them is a product decision, not a technical constant.
4. Guardrails: checks outside the model, on input and output, such as classifiers, regular expressions, schema validation and allowlists. They are the only part of the safety stack that is deterministic.
5. Evals: automated test suites over fixed inputs with scoring, covering both capability and safety. Chapter 50 covered how to build them; the essential discipline is that a private eval set you wrote beats any public leaderboard.

6. System card: a document about a deployed system rather than a bare model, covering intended use, safety evaluations, red-team results, known failure modes and mitigations. The convention was established by the GPT-4 system card in March 2023 and is now normal for frontier releases.
7. Where the field appears to be heading, explicitly labelled speculation and not fact: more inference-time compute rather than only larger models; sparse mixture-of-experts as the default architecture; agentic use as the main product surface; standardized tool protocols; and small models on devices for the majority of routine work.
8. Any of these could be wrong within a year, and this paragraph should be read with its date attached.

■ 51.16.6 WORDS – remember these

Multimodal — text plus images or audio — inputs encoded into the same embedding space and consumed as one sequence. Reasoning model — thinks before it answers — trained to emit long internal chains of thought, billed as output tokens. System card — the document about the product — deployment-level description of intended use, evaluations and mitigations. Alignment — making behaviour match the specification — preference training and rules, measured by evaluations. Guardrail — a check outside the model — deterministic filtering or validation on inputs and outputs. Distillation — a small model taught by a big one — student trained on teacher outputs, from a 2015 paper.

51.17 A practical decision guide

■ 51.17.1 PLAIN – in simple words

1. Most projects reach for the most complicated option first. Work the other way.
2. Ask whether the task needs a language model at all. Regular expressions, database queries and existing libraries are cheaper, faster and exact.
3. If it does, start with one API call and a good prompt.
4. Add retrieval when the answer depends on documents the model has not seen.
5. Add tools when the answer depends on live data or on an action.
6. Add a loop, making it an agent, only when the number of steps genuinely cannot be known in advance.
7. Fine-tune when the format or style must be fixed, or when you want a small model to do one narrow job cheaply.
8. Self-host when policy, volume or control demands it, not because it sounds cheaper.

■ 51.17.2 PLAIN – a picture in your head

1. Think of choosing transport for a journey.
2. A taxi is a hosted API: instant, no maintenance, pay per trip.
3. Bringing a map is retrieval.
4. Hiring a driver who chooses the route as they go is an agent.
5. Buying a fleet is self-hosting, and it only pays if you drive constantly.

Where this comparison breaks: transport choices are reversible in an afternoon. Architecture choices lock in data formats, evaluation sets and vendor contracts, so the cost of starting too complicated is higher than the comparison suggests.

■ 51.17.3 PLAIN – a worked example

```

Does it need judgement about language?
no -> write ordinary code. Stop.
yes -> Is the needed knowledge in the model already?
      yes -> one API call with a good prompt
      no -> Is it in documents you hold?
            yes -> RAG
            no -> Is it live or an action?
                  yes -> tools
                  no -> reconsider the task

Is the number of steps known in advance?
yes -> a fixed chain of calls
no -> an agent loop, with step and cost limits

Is the output format or style unstable?
yes and you have 1,000+ examples -> fine-tune
yes and you do not -> few-shot examples

Must data stay inside your network, or is volume
above roughly 3 billion output tokens a month?
yes -> self-host
no -> hosted API

```

■ 51.17.4 PLAIN – what is really happening inside

1. The same guidance as a table, with the cost and latency notes that decide arguments.

Task	Approach	Note
Classify text	small hosted model	cheap, fast, few-shot
Extract fields	model plus schema	constrained decoding
Answer from your docs	RAG	retrieval is the risk
Answer about live data	tools	validate arguments
Summarize a document	one call	watch input cost
Fixed house style	fine-tune	needs 1,000+ examples
Multi-step research	agent	cap steps and budget
Regulated private data	self-host	cost is not the reason
Bulk offline work	batch API	half price, slower

2. Latency notes worth having in mind. A short hosted call is roughly 0.5 to 3 seconds. Adding retrieval adds 50 to 300 milliseconds.
3. Each tool round trip adds a full model call. A reasoning model can take 10 to 60 seconds. An agent is measured in minutes.
4. Cost notes. A simple call is fractions of a cent. RAG multiplies input tokens. An agent multiplies whole calls. Reasoning multiplies output tokens. The multipliers stack.

■ 51.17.5 TECHNICAL – the engineer's version

1. Build the evaluation set before choosing an approach. Thirty to two hundred real examples with known good answers, held privately, decides every question in this section empirically rather than by argument.
2. Choose the cheapest model that passes your evaluation, then move up only if it fails. Most production traffic does not need a frontier model.
3. Design for model substitution from the first day: keep prompts in one place, keep an adapter layer, and re-run the evaluation set when you change models, because a provider changing an undated model name is a real event.

4. Route by difficulty. A small model handles the common case and escalates on low confidence or explicit triggers. This routinely cuts costs by an order of magnitude while leaving quality on hard cases intact.
5. Instrument tokens per request, cost per task, cache hit rate, p50 and p99 latency, and end-to-end task success. The last one is the only one your users experience.

■ 51.17.6 WORDS – remember these

Model routing — send easy work to the cheap model — confidence or rule-based dispatch across a tier of models, with escalation. Escalation — try harder when the cheap answer looks weak — a second call to a stronger model triggered by a confidence signal. Eval set — your own private exam for the system — a fixed collection of real inputs with expected outputs, versioned with the code. Golden path — the case you optimize for — the dominant traffic pattern that should decide model and architecture choice.

51.98 Common wrong ideas

1. Wrong: it looked that up when I asked. Right: unless a retrieval or search tool ran, nothing was looked up. The answer came from frozen weights.
2. Wrong: it remembers me. Right: it has no memory between calls. An application stored your text and pasted it back into the context.
3. Wrong: hallucination will be fixed in the next version. Right: it is a property of predicting likely text without verification. Versions reduce the rate; retrieval, tools and checking reduce it much more.
4. Wrong: the system prompt is secure and hidden. Right: it is ordinary tokens at the front of the context, obeyed by training habit, and it can be overridden or revealed by clever input.
5. Wrong: an agent can be trusted to finish the job. Right: per-step errors compound, so a 95 percent reliable step is 21 percent reliable over thirty steps. Cap steps, verify each one, keep a human on writes.
6. Wrong: self-hosting is cheaper. Right: you rent the machine whether you use it or not. Break-even against cheap hosted models is around billions of output tokens a month.
7. Wrong: a bigger context window means it uses all of it well. Right: capacity is not competence. Attention to the middle of a long context is measurably weaker than to the ends.
8. Wrong: more parameters always means faster answers if the hardware is good. Right: decode speed is memory bandwidth divided by model bytes. A bigger model is slower on the same machine, in direct proportion.
9. Wrong: prompt engineering is the main skill. Right: it is real but shallow. Retrieval quality, task decomposition, evaluation and tool design matter more.
10. Wrong: the model runs my code when it calls a function. Right: it emits a structured request; your program decides whether to run it. Every tool is a security boundary you own.

51.99 Chapter summary in 20 lines

1. Inference is the frozen model answering; training is what produced it, and it needs roughly ten times the memory.
2. An API call is ordinary HTTPS carrying a list of messages plus sampling settings, and the whole conversation is resent every time.
3. The system prompt is only leading tokens. Its authority is a training habit, not a protection.
4. Generation has two phases: prefill, which is parallel and compute-bound, and decode, which is sequential and bandwidth-bound.
5. Decode speed is approximately memory bandwidth divided by model bytes, which explains almost every speed question people ask.
6. Time to first token is network plus queue plus prefill; tokens per second afterwards is a different limit entirely.
7. The KV cache turns quadratic recomputation into linear work and costs 128 KiB per token for an 8B model at 16-bit.
8. Continuous batching, PagedAttention, speculative decoding, quantization, tensor parallelism and prefix caching are the serving toolkit.
9. The same model runs at different speeds at different providers because of hardware, quantization, batching policy, load and distance.
10. Output tokens cost five to six times input tokens because decode is sequential and prefill is parallel.
11. Self-hosting is rarely cheaper below very high sustained volume; the good reasons are privacy, control and stability.
12. llama.cpp is the engine, Ollama and LM Studio the friendly wrappers, vLLM and SGLang the serving engines, MLX the Apple silicon path.
13. Memory decides whether a model runs; bandwidth decides how fast. Apple's unified memory wins on capacity and loses on bandwidth and prefill.
14. Read the model card for base model, data, licence, limitations and evaluations, and default to Q4_K_M unless you have a reason.
15. Training a tiny model on a laptop teaches shape without knowledge; do it once as education, then fine-tune or use an open model instead.
16. RAG is chunk, embed, store, retrieve, inject, generate; chunking and retrieval quality dominate the outcome.
17. Tool calling means the model emits a request and your code executes it, so every tool is a security boundary.
18. An agent is a loop, and per-step errors compound multiplicatively, so narrow scope, verifiable steps and human checkpoints are what make it work.
19. Hallucination is intrinsic to predicting likely text without verification; retrieval, tools and constrained output reduce it, exhortation does not.
20. Prompt injection remains unsolved as of August 2026 because instructions and data share one channel, so build as though confident wrong output will happen, because it will.



PART I

The Reference Desk

The numbers, commands, decision trees, timeline and
glossary you will come back to.

The Engineer's Reference

52.0 How to use this chapter

1. Every other chapter in this book teaches. This one does not teach. It answers.
2. It is the chapter you open when you are already in the middle of something and you need one number, one flag, or one next step.
3. Nothing here is new. Every fact appeared earlier in the book, in context, with the explanation attached. Here it is stripped down to the lookup.
4. The chapter is arranged so you can find things without reading. The section numbers below are the map.
5. Sections 52.1 to 52.3 are numbers: latency, powers of two, units, bases, character codes. Use these when you are estimating.
6. Section 52.4 is the networking reference: layers, addresses, ports, status codes, record types, states. Use this when you are reading a capture or a log.
7. Section 52.5 is the networking command cheat sheet. Every command has two extra columns that most cheat sheets leave out: what a pass **proves**, and what a pass does **not** prove. That distinction is the whole diagnostic method of this book.
8. Section 52.6 is a decision tree for “I cannot reach a site”. It is written as a branching procedure. Follow the numbers.
9. Sections 52.7 and 52.8 are the same two things for git: a full command cheat sheet, then a decision tree for a failed git operation.
10. Sections 52.9 to 52.16 are the rest of the desk reference: shell, file magic numbers, complexity, cloud, security, AI, regular expressions, editors.
11. Section 52.17 is what to do when none of the above worked.
12. Two conventions run through the chapter. First, macOS syntax is given first, because that is what the reader of this book is using, with the Linux difference noted on the next line. Second, anything that is a convention rather than a standard is labelled as such.
13. Anything marked “approximate” is an order of magnitude, not a measurement. Order of magnitude is usually what you need. A number that is right to within a factor of two will stop you making the wrong decision.
14. One warning before you start. A reference table tells you what a thing is. It does not tell you whether the thing is your problem. Keep the habit of asking what an observation rules out, not just what it looks like.

52.1 Numbers every engineer should know

■ 52.1.1 Why this section exists

1. Most engineering decisions are decided by a factor of a thousand, not by a factor of two.
2. If you know that memory is roughly one hundred times slower than cache, and a network hop across the planet is roughly a million times slower than memory, you will make the right architectural call without measuring.
3. The list below descends from fastest to slowest. Learn the shape, not the digits.
4. The original list of this kind is credited to Jeff Dean of Google, who presented “Numbers Everyone Should Know” in a 2009 keynote. Peter Norvig published a similar table earlier, in his 2001 essay “Teach Yourself Programming in Ten Years”.
5. The figures below are updated for typical 2025 to 2026 hardware. They are approximate.

■ 52.1.2 The latency ladder, real figures

Operation	Typical time	Note
1 CPU cycle at 3 GHz	0.33 ns	one tick
L1 cache reference	1 ns	4 to 5 cycles
Branch mispredict	3 to 5 ns	pipeline refill
L2 cache reference	4 ns	about 14 cycles
L3 cache reference	15 to 40 ns	shared, slower
Mutex lock and unlock	17 ns	uncontended
Main memory reference	80 to 100 ns	DDR4 or DDR5
Read 1 MB from RAM	50 us	one core, 20 GB/s
NVMe SSD 4 KiB read	20 to 100 us	random
Read 1 MB from NVMe	150 to 300 us	sequential
Same-rack round trip	0.1 ms	switch to switch
Same-datacentre round trip	0.5 ms	500 us
HDD seek, 7200 rpm	8 ms	seek plus rotate
Read 1 MB from HDD	5 ms	sequential
Same-city round trip	5 to 10 ms	metro fibre
Same-continent round trip	20 to 50 ms	
Mumbai to Frankfurt	110 to 130 ms	approximate
Mumbai to Virginia	190 to 220 ms	approximate
Cross-continent worst case	250 to 300 ms	
Geostationary satellite	480 to 600 ms	35,786 km up
Low-earth-orbit satellite	25 to 60 ms	about 550 km up

1. The single most useful line in that table is the second-from-last group: light in glass fibre travels at about 200,000 kilometres per second, which is roughly two thirds of the speed of light in vacuum.
2. That gives a rule you can do in your head: about **1 millisecond of round trip per 100 kilometres of fibre**.
3. Check it. Mumbai to London is about 7,200 km in a straight line and about 10,000 km of real cable route. Ten thousand divided by one hundred is 100 milliseconds. Measured round trips on that path are 110 to 130 ms.
4. This rule is physics. No amount of money buys past it. It is why content delivery networks exist: you cannot make the light faster, so you move the copy closer.

■ 52.1.3 The latency ladder, scaled to human time

1. Multiply every figure by one billion. One nanosecond becomes one second.
2. Now the ladder becomes something a person can feel.

Operation	Real	Human scale
L1 cache reference	1 ns	1 second
L2 cache reference	4 ns	4 seconds
L3 cache reference	15 ns	15 seconds
Main memory reference	100 ns	1 minute 40 seconds
NVMe SSD read, fast	20 us	5.6 hours
NVMe SSD read, typical	100 us	1.2 days
Same-datacentre round trip	0.5 ms	5.8 days
Read 1 MB from HDD	5 ms	2 months
HDD seek	8 ms	3 months
Mumbai to Frankfurt	120 ms	3.8 years
Mumbai to Virginia	200 ms	6.3 years
Cross-continent worst case	300 ms	9.5 years
Machine reboot	30 s	950 years
The reader's curl timeout	15 s	475 years

3. Read the last two lines again. When the reader's curl waited fifteen seconds and gave up, the processor had, in its own sense of time, waited several human centuries for an answer that never came.
4. That is why a timeout is such strong evidence. Nothing on a working network takes that long. Silence for fifteen seconds is not slowness. It is absence.

■ 52.1.4 Powers of two, 2⁰ to 2⁶⁴

1. Every size in computing is a power of two or a multiple of one.
2. Learn 2¹⁰ = 1024, 2²⁰ = about one million, 2³⁰ = about one billion. The rest follows.

Power	Value	As a byte size
2 ⁰	1	-
2 ¹	2	-
2 ²	4	-
2 ³	8	one byte in bits
2 ⁴	16	one hex digit range
2 ⁵	32	-
2 ⁶	64	-
2 ⁷	128	ASCII range
2 ⁸	256	one byte of values
2 ⁹	512	classic disk sector
2 ¹⁰	1024	1 KiB
2 ¹¹	2048	2 KiB
2 ¹²	4096	4 KiB, one page
2 ¹³	8192	8 KiB
2 ¹⁴	16384	16 KiB
2 ¹⁵	32768	32 KiB, int16 max+1
2 ¹⁶	65536	64 KiB, port count

Power	Value	As a byte size
2 ¹⁷	131072	128 KiB
2 ¹⁸	262144	256 KiB
2 ¹⁹	524288	512 KiB
2 ²⁰	1048576	1 MiB
2 ²¹	2097152	2 MiB
2 ²²	4194304	4 MiB
2 ²³	8388608	8 MiB
2 ²⁴	16777216	16 MiB, colours
2 ²⁵	33554432	32 MiB
2 ²⁶	67108864	64 MiB
2 ²⁷	134217728	128 MiB
2 ²⁸	268435456	256 MiB
2 ²⁹	536870912	512 MiB
2 ³⁰	1073741824	1 GiB
2 ³¹	2147483648	2 GiB, int32 max+1
2 ³²	4294967296	4 GiB, IPv4 space
2 ³³	8589934592	8 GiB
2 ³⁴	17179869184	16 GiB
2 ³⁵	34359738368	32 GiB
2 ³⁶	68719476736	64 GiB
2 ³⁷	137438953472	128 GiB
2 ³⁸	274877906944	256 GiB
2 ³⁹	549755813888	512 GiB
2 ⁴⁰	1099511627776	1 TiB
2 ⁴¹	2199023255552	2 TiB
2 ⁴²	4398046511104	4 TiB
2 ⁴³	8796093022208	8 TiB
2 ⁴⁴	17592186044416	16 TiB
2 ⁴⁵	35184372088832	32 TiB
2 ⁴⁶	70368744177664	64 TiB
2 ⁴⁷	140737488355328	128 TiB, x86 user VA
2 ⁴⁸	281474976710656	256 TiB, MAC space
2 ⁴⁹	562949953421312	512 TiB
2 ⁵⁰	1125899906842624	1 PiB
2 ⁵¹	2251799813685248	2 PiB
2 ⁵²	4503599627370496	4 PiB, double mantissa
2 ⁵³	9007199254740992	8 PiB, exact int limit
2 ⁵⁴	18014398509481984	16 PiB
2 ⁵⁵	36028797018963968	32 PiB
2 ⁵⁶	72057594037927936	64 PiB
2 ⁵⁷	144115188075855872	128 PiB
2 ⁵⁸	288230376151711744	256 PiB
2 ⁵⁹	576460752303423488	512 PiB
2 ⁶⁰	1152921504606846976	1 EiB
2 ⁶¹	2305843009213693952	2 EiB
2 ⁶²	4611686018427387904	4 EiB
2 ⁶³	9223372036854775808	8 EiB, int64 max+1
2 ⁶⁴	18446744073709551616	16 EiB, address space

3. Three lines in that table cause more real bugs than the rest combined.
4. 2^{31} minus 1 is 2147483647, the largest signed 32-bit integer. This is the Year 2038 problem: a 32-bit Unix timestamp counting seconds from 1 January 1970 overflows at 03:14:07 UTC on 19 January 2038.
5. 2^{53} is 9007199254740992. A double-precision floating point number, the only number type JavaScript had until BigInt was standardized in ES2020, can represent every integer up to 2^{53} exactly and not one above it. Git commit hashes and large database IDs sent as JSON numbers are silently corrupted here.
6. 2^{32} is 4294967296, the total number of IPv4 addresses. That is fewer than the number of humans alive. This is why NAT exists.

■ 52.1.5 Decimal versus binary prefixes

1. There are two systems and they disagree by a growing amount.
2. Decimal prefixes are powers of a thousand. Binary prefixes are powers of 1024. The binary ones were standardized by the International Electrotechnical Commission in IEC 60027-2, amendment 2, in January 1999, and now live in ISO/IEC 80000-13:2008.

Decimal	Value	Binary	Gap
kB	10^3	KiB = 2^{10}	2.4%
MB	10^6	MiB = 2^{20}	4.9%
GB	10^9	GiB = 2^{30}	7.4%
TB	10^{12}	TiB = 2^{40}	10.0%
PB	10^{15}	PiB = 2^{50}	12.6%
EB	10^{18}	EiB = 2^{60}	15.3%
ZB	10^{21}	ZiB = 2^{70}	18.1%
YB	10^{24}	YiB = 2^{80}	20.9%

3. Who uses which, in practice, is a convention and not a standard.
4. Disk and SSD makers use decimal. A “1 TB” drive holds 10^{12} bytes.
5. Operating systems differ. Windows reports that same drive as 931 GB, because it divides by 2^{30} but writes the label “GB”. macOS since version 10.6, released in 2009, uses decimal, so it agrees with the box and says 1 TB. Linux tools vary; `df -h` uses 1024 by default, `df -H` uses 1000.
6. RAM is always binary. A “16 GB” memory module is 16 GiB, because memory is addressed by binary address lines and cannot be anything else.
7. Network speeds are always decimal, and always in bits. “1 Gbit/s” is 1,000,000,000 bits per second.
8. In November 2022 the 27th General Conference on Weights and Measures added four new SI prefixes: ronna for 10^{27} , quetta for 10^{30} , ronto for 10^{-27} and quecto for 10^{-30} . There are no matching binary prefixes.

■ 52.1.6 Common bandwidth figures

Link	Rate	Year introduced
10BASE-T Ethernet	10 Mbit/s	1990
100BASE-TX	100 Mbit/s	1995
1000BASE-T	1 Gbit/s	1999
10GBASE-T	10 Gbit/s	2006
400G Ethernet	400 Gbit/s	2017

Link	Rate	Year introduced
800G Ethernet	800 Gbit/s	2024
Wi-Fi 802.11b	11 Mbit/s	1999
Wi-Fi 802.11g	54 Mbit/s	2003
Wi-Fi 4, 802.11n	up to 600 Mbit/s	2009
Wi-Fi 5, 802.11ac	up to 6.9 Gbit/s	2013
Wi-Fi 6, 802.11ax	up to 9.6 Gbit/s	2021
Wi-Fi 7, 802.11be	up to 46 Gbit/s	2024
USB 2.0	480 Mbit/s	2000
USB 3.0	5 Gbit/s	2008
USB4	40 Gbit/s	2019
USB4 version 2	80 Gbit/s	2022
Thunderbolt 3 and 4	40 Gbit/s	2015, 2020
Thunderbolt 5	80 to 120 Gbit/s	2023
SATA III	6 Gbit/s	2009
PCIe 3.0 x16	15.75 GB/s	2010
PCIe 4.0 x16	31.5 GB/s	2017
PCIe 5.0 x16	63 GB/s	2019
DDR4-3200, one channel	25.6 GB/s	2014
DDR5-6400, one channel	51.2 GB/s	2020
HBM3 per stack	819 GB/s	2022
HBM3E per stack	about 1.2 TB/s	2024

1. Note the unit change halfway down. Network and bus speeds are quoted in bits per second. Memory and PCIe totals are usually quoted in bytes per second. Mixing them up gives you an eight-fold error.

■ 52.1.7 Typical throughput of real systems

1. These are order-of-magnitude figures for a single well-configured node in 2025 to 2026. Your numbers will differ. The shape will not.

System	Typical throughput	Note
Redis, single core	100k ops/s	simple GET and SET
Redis, pipelined	1M ops/s	batched requests
PostgreSQL, simple reads	10k to 50k queries/s	indexed, cached
nginx, static file	50k to 500k req/s	small files
Kafka, one broker	100 MB/s and up	sequential writes
HDD, 7200 rpm	100 to 200 IOPS	random 4 KiB
SATA SSD	up to 90k IOPS	random 4 KiB
NVMe SSD	up to 1M IOPS	random 4 KiB
gzip compress	30 to 100 MB/s	one core
zstd compress, level 3	300 to 500 MB/s	one core
SHA-256 with hardware	1 to 2 GB/s	one core
AES-256-GCM with AES-NI	2 to 10 GB/s	one core
JSON parse	100 to 500 MB/s	one core

2. One more figure that catches people out. A single TCP connection is limited by its window divided by the round-trip time. With a 64 KiB window and a 100 ms round trip, that is 64 KiB divided by 0.1 seconds, which is about 655 kB/s, which is about 5.2 Mbit/s.

3. That is the bandwidth-delay product limit. It is why a single download from a distant server can crawl on a gigabit line, and why parallel connections or a larger window fix it. Modern systems scale the window automatically, up to 1 GiB, using the window scale option of RFC 7323.

52.2 Units and conversions

■ 52.2.1 Bit and byte

1. A **bit** is one binary digit. It is either 0 or 1.
2. A **byte** is eight bits. This has been near-universal since the IBM System/360 in 1964. Older machines used 6, 7, 9 and 36-bit units.
3. The correct formal word for exactly eight bits is **octet**. Networking standards use it, because “byte” once meant other sizes. RFC documents say octet.
4. Lowercase b means bit. Uppercase B means byte. This is a convention, but a very strongly held one.
5. A **nibble** is four bits, which is exactly one hexadecimal digit.
6. A **word** has no fixed size. It means the natural unit of the processor: 16 bits on an 8086, 32 on an i386, 64 on modern x86-64 and ARM64.

■ 52.2.2 The full prefix table

Prefix	Symbol	Factor
quetta	Q	10^{30}
ronna	R	10^{27}
yotta	Y	10^{24}
zetta	Z	10^{21}
exa	E	10^{18}
peta	P	10^{15}
tera	T	10^{12}
giga	G	10^9
mega	M	10^6
kilo	k	10^3
milli	m	10^{-3}
micro	u	10^{-6}
nano	n	10^{-9}
pico	p	10^{-12}
femto	f	10^{-15}
atto	a	10^{-18}
zepto	z	10^{-21}
yocto	y	10^{-24}
ronto	r	10^{-27}
quecto	q	10^{-30}

1. Note that kilo is lowercase k in SI. Uppercase K is not an SI prefix; it is the symbol for kelvin. Everyone writes KB anyway. That is a convention that contradicts the standard.
2. The micro symbol is properly the Greek letter mu. In plain text everyone writes u. This book writes us for microseconds.

■ 52.2.3 MB/s versus Mb/s

1. To convert bits per second to bytes per second, divide by 8.
2. To convert bytes per second to bits per second, multiply by 8.
3. Some people add ten per cent for framing overhead when talking about Ethernet. That is a rough allowance, not a standard.

Line rate	Theoretical max	Realistic
10 Mbit/s	1.25 MB/s	1.1 MB/s
100 Mbit/s	12.5 MB/s	11 MB/s
1 Gbit/s	125 MB/s	110 to 118 MB/s
2.5 Gbit/s	312.5 MB/s	290 MB/s
10 Gbit/s	1.25 GB/s	1.15 GB/s
40 Gbit/s	5 GB/s	4.6 GB/s
100 Gbit/s	12.5 GB/s	11.5 GB/s

4. Worked example. Your broadband plan says 300 Mbit/s. Divide by 8 to get 37.5 MB/s. A 4 GB file, at best, takes 4000 divided by 37.5, which is about 107 seconds. If your download manager shows 5 MB/s, you are getting about 40 Mbit/s, which is roughly one seventh of the plan.

■ 52.2.4 Hertz to time period

1. Frequency and period are reciprocals. Period equals one divided by frequency.

Frequency	Period	Where you see it
1 Hz	1 s	a clock tick
50 Hz	20 ms	mains power, India
60 Hz	16.67 ms	mains power, USA
60 Hz refresh	16.67 ms	standard display
120 Hz refresh	8.33 ms	fast display
1 kHz	1 ms	audio low end
44.1 kHz	22.7 us	CD sample rate
1 MHz	1 us	early CPUs
1 GHz	1 ns	modern CPU order
3 GHz	0.333 ns	one CPU cycle
2.4 GHz	0.417 ns	Wi-Fi band centre
5 GHz	0.2 ns	Wi-Fi band centre

2. Wavelength equals the speed of light divided by frequency. At 2.4 GHz that is 300,000,000 divided by 2,400,000,000, which is 12.5 centimetres. That is why a 2.4 GHz antenna is a few centimetres long.

■ 52.2.5 dBm to milliwatts

1. dBm is power measured in decibels relative to one milliwatt. It is a logarithmic scale, so it turns multiplication into addition.
2. The formula is: power in dBm equals 10 times the base-10 logarithm of the power in milliwatts.
3. Two shortcuts do almost all the work. Adding 3 dB doubles the power. Adding 10 dB multiplies it by ten.

dBm	Power	Meaning for Wi-Fi
+30 dBm	1000 mW = 1 W	legal max in some bands
+20 dBm	100 mW	typical router transmit
0 dBm	1 mW	reference point
-30 dBm	0.001 mW	excellent signal
-50 dBm	very small	very good
-60 dBm	very small	good
-67 dBm	very small	minimum for video
-70 dBm	very small	usable, weak
-80 dBm	very small	unreliable
-90 dBm	very small	unusable, near noise

- Signal strength in Wi-Fi is always a negative dBm number, because received power is always far below one milliwatt. Closer to zero is stronger. Minus 50 is much better than minus 80.
- A drop of 3 dBm means you are receiving half the power. A drop of 10 dBm means one tenth.

■ 52.2.6 Temperature

- To convert Celsius to Fahrenheit: multiply by 9, divide by 5, add 32.
- To convert Fahrenheit to Celsius: subtract 32, multiply by 5, divide by 9.
- To convert Celsius to Kelvin: add 273.15.

Celsius	Fahrenheit	What it is
-273.15	-459.67	absolute zero
0	32	water freezes
20	68	comfortable room
27	80.6	warm room, India
35	95	hot ambient
40	104	idle CPU, normal
70	158	CPU under load
90	194	CPU throttling starts
100	212	water boils, CPU limit
105	221	typical shutdown point

- Silicon does not fail at 90 degrees Celsius. It slows itself down on purpose, which is called thermal throttling, to avoid getting hotter. A laptop that suddenly gets slow after ten minutes of heavy work is almost always throttling.

■ 52.2.7 Storage sizes and what fits

Size	What fits
1 byte	one ASCII character
100 bytes	a short sentence
1 KiB	half a page of text
10 KiB	a small web page's HTML
100 KiB	a compressed photo, low
1 MiB	a typical JPEG photo
5 MiB	a 4-minute MP3 at 160 kbit/s
50 MiB	a phone photo burst

Size	What fits
700 MiB	one CD-ROM
1 GiB	250 songs, or 1 hour SD video
4.7 GB	one single-layer DVD
8 GiB	a small language model, 4-bit
25 GB	one Blu-ray layer
50 GB	a large modern game
256 GB	a typical phone
1 TB	250,000 photos
4 TB	a big desktop drive
20 TB	one large hard disk, 2026
1 PB	a mid-size company's archive
1 EB	global scale, a few of these

- Two facts to keep the scale honest. The English Wikipedia text dump, without images and without history, is roughly 25 GB uncompressed and about 22 GB compressed as of 2025. The complete works of Shakespeare in plain text are about 5.5 MB.

52.3 Number bases and encoding

■ 52.3.1 Binary, decimal, hexadecimal and octal, 0 to 32

- Hexadecimal is base 16. It uses 0 to 9 then A to F.
- One hex digit is exactly four bits. Two hex digits are exactly one byte. That is the entire reason hex exists in computing.
- Octal is base 8, three bits per digit. It survives in Unix file permissions and in C escape sequences.

Dec	Binary	Hex	Octal
0	000000	00	0
1	000001	01	1
2	000010	02	2
3	000011	03	3
4	000100	04	4
5	000101	05	5
6	000110	06	6
7	000111	07	7
8	001000	08	10
9	001001	09	11
10	001010	0A	12
11	001011	0B	13
12	001100	0C	14
13	001101	0D	15
14	001110	0E	16
15	001111	0F	17
16	010000	10	20
17	010001	11	21
18	010010	12	22
19	010011	13	23
20	010100	14	24

Dec	Binary	Hex	Octal
21	010101	15	25
22	010110	16	26
23	010111	17	27
24	011000	18	30
25	011001	19	31
26	011010	1A	32
27	011011	1B	33
28	011100	1C	34
29	011101	1D	35
30	011110	1E	36
31	011111	1F	37
32	100000	20	40

■ 52.3.2 The powers of two in hexadecimal

1. In hex, every power of two is a 1, 2, 4 or 8 followed by zeros. This makes sizes instantly recognizable in a hex dump.

Power	Decimal	Hex
2 ⁸	256	0x100
2 ¹⁰	1024	0x400
2 ¹²	4096	0x1000
2 ¹⁶	65536	0x10000
2 ²⁰	1048576	0x100000
2 ²⁴	16777216	0x1000000
2 ³⁰	1073741824	0x40000000
2 ³¹	2147483648	0x80000000
2 ³²	4294967296	0x100000000
2 ⁴⁰	1 TiB	0x10000000000
2 ⁴⁸	256 TiB	0x1000000000000
2 ⁶⁴	16 EiB	0x10000000000000000

2. When you see 0xFFFFFFFF in a log, that is 4294967295, which is the largest unsigned 32-bit value, which usually means “all bits set” or “not found”.
3. When you see 0xDEADBEEF or 0xCAFEFABE, that is a human-chosen marker value, put there so it stands out in memory. 0xCAFEFABE is also the real first four bytes of a Java class file, chosen by James Gosling's team in 1991 for exactly that reason.

■ 52.3.3 The printable ASCII table

1. ASCII was standardized in 1963 by the American Standards Association and revised in 1967 and 1986. It defines 128 codes, 0 to 127.
2. Codes 0 to 31 and 127 are control codes. Codes 32 to 126 are printable.

Dec	Hx	Ch	Dec	Hx	Ch	Dec	Hx	Ch	Dec	Hx	Ch
32	20	SP	56	38	8	80	50	P	104	68	h
33	21	!	57	39	9	81	51	Q	105	69	i
34	22	"	58	3A	:	82	52	R	106	6A	j
35	23	#	59	3B	;	83	53	S	107	6B	k
36	24	\$	60	3C	<	84	54	T	108	6C	l

37	25	%	61	3D	=	85	55	U	109	6D	m
38	26	&	62	3E	>	86	56	V	110	6E	n
39	27	'	63	3F	?	87	57	W	111	6F	o
40	28	(	64	40	@	88	58	X	112	70	p
41	29	)	65	41	A	89	59	Y	113	71	q
42	2A	*	66	42	B	90	5A	Z	114	72	r
43	2B	+	67	43	C	91	5B	[	115	73	s
44	2C	,	68	44	D	92	5C	\	116	74	t
45	2D	-	69	45	E	93	5D	]	117	75	u
46	2E	.	70	46	F	94	5E	^	118	76	v
47	2F	/	71	47	G	95	5F	_	119	77	w
48	30	0	72	48	H	96	60	`	120	78	x
49	31	1	73	49	I	97	61	a	121	79	y
50	32	2	74	4A	J	98	62	b	122	7A	z
51	33	3	75	4B	K	99	63	c	123	7B	{
52	34	4	76	4C	L	100	64	d	124	7C	
53	35	5	77	4D	M	101	65	e	125	7D	}
54	36	6	78	4E	N	102	66	f	126	7E	~
55	37	7	79	4F	O	103	67	g			

3. Four relationships in that table are worth memorizing, because they let you do character arithmetic in your head.
4. The digit 0 is 48, which is hex 0x30. So the numeric value of a digit character is the character minus 48.
5. Uppercase A is 65, hex 0x41. Lowercase a is 97, hex 0x61.
6. The gap between an uppercase letter and its lowercase form is exactly 32, which is a single bit: bit 5, value 0x20. Flipping that one bit changes case. That is not an accident; the 1963 committee arranged it deliberately.
7. Space is 32, hex 0x20. Delete is 127, hex 0x7F.

■ 52.3.4 Control codes and escape sequences

Escape	Dec	Hex	Meaning
\0	0	00	null, string end in C
\a	7	07	bell, terminal beep
\b	8	08	backspace
\t	9	09	horizontal tab
\n	10	0A	line feed, newline
\v	11	0B	vertical tab
\f	12	0C	form feed, new page
\r	13	0D	carriage return
\e	27	1B	escape, starts ANSI
\\	92	5C	a literal backslash
\"	34	22	a literal double quote
\'	39	27	a literal single quote
\x41	65	41	hex escape, gives A
\101	65	41	octal escape, gives A
\u00E9	233	E9	Unicode escape, e-acute

1. The line ending difference is the single most common cause of a file looking corrupted when moved between systems.
2. Unix and macOS since version 10 end a line with LF alone, which is 0x0A.

3. Windows ends a line with CR then LF, which is 0x0D 0x0A.
4. Classic Mac OS, up to version 9 in 2001, used CR alone, 0x0D.
5. In a hex dump, 0D 0A at the end of each line means the file came from Windows. In git, the setting `core.autocrlf` controls whether git converts between the two.

■ 52.3.5 UTF-8 byte-length rules

1. UTF-8 was designed by Ken Thompson and Rob Pike in September 1992, famously sketched on a placemat in a diner in New Jersey. It is defined today by RFC 3629, from 2003.
2. It encodes every Unicode code point in one to four bytes, and it is backward compatible with ASCII: any ASCII file is already valid UTF-8.

Code point range	Bytes	First byte pattern
U+0000 to U+007F	1	0xxxxxxx
U+0080 to U+07FF	2	110xxxxx
U+0800 to U+FFFF	3	1110xxxx
U+10000 to U+10FFFF	4	11110xxx

3. Every byte after the first in a multi-byte sequence has the pattern 10xxxxxx. That is the self-synchronizing property: from any byte you can find the start of the character by scanning backwards past bytes beginning with 10.
4. A byte beginning with 0 is a lone ASCII character. A byte beginning with 11 starts a multi-byte character. A byte beginning with 10 is a continuation.
5. Practical consequences you will meet.
6. English text costs about 1 byte per character. Latin letters with accents, Greek, Cyrillic, Hebrew and Arabic cost 2. Devanagari, Chinese, Japanese and Korean cost 3. Emoji and rarer historic scripts cost 4.
7. So the same sentence stored in Hindi takes roughly three times the bytes of the English version, before you even count that Hindi may need more characters.
8. `len()` in Python 3 counts code points, not bytes. `strlen()` in C counts bytes. `.length` in JavaScript counts UTF-16 code units, so an emoji outside the basic plane counts as 2.
9. A byte order mark, U+FEFF, encoded as the bytes EF BB BF, sometimes appears at the start of a UTF-8 file written by Windows tools. It is legal but discouraged, and it breaks shell scripts, because the shell sees those three bytes before the `#!` line.

52.4 The networking reference

■ 52.4.1 OSI and TCP/IP side by side

1. The OSI model has seven layers. It was published as ISO/IEC 7498-1, with the first version in 1984.
2. The TCP/IP model has four layers. It was written down in RFC 1122 in October 1989, by which time the protocols had already been running for years.
3. OSI is the vocabulary everyone uses. TCP/IP is the thing that actually runs. Both statements are true at the same time.

OSI	Name	TCP/IP	Examples
7	Application	Application	HTTP, DNS, SSH
6	Presentation	Application	TLS, JPEG, UTF-8

OSI	Name	TCP/IP	Examples
5	Session	Application	RPC, TLS sessions
4	Transport	Transport	TCP, UDP, QUIC
3	Network	Internet	IP, ICMP, IPsec
2	Data link	Link	Ethernet, Wi-Fi, ARP
1	Physical	Link	copper, fibre, radio

4. Three honest notes about that table.
5. TLS does not fit cleanly. It sits above TCP and below HTTP, which is layer 6-ish, but it has its own handshake and session resumption, which is layer 5-ish. People argue about this. Nobody wins.
6. ARP is called layer 2 by most people and layer 2.5 by pedants, because it carries layer 3 addresses inside layer 2 frames.
7. QUIC, standardized in RFC 9000 in May 2021, runs on top of UDP but does transport work: streams, ordering, congestion control, encryption. In OSI terms it is layer 4 built on top of layer 4.
8. Where the reader's own failure sat: DNS worked, so layer 7 name lookup was fine. The TCP handshake never completed, so the failure was at layer 4 or below, on the path.

■ 52.4.2 The RFC 1918 private ranges

1. RFC 1918, published February 1996, reserves three blocks that anyone may use inside a private network and that no router on the public internet will forward.

Block	Range	Addresses
10.0.0.0/8	10.0.0.0 to 10.255.255.255	16,777,216
172.16.0.0/12	172.16.0.0 to 172.31.255.255	1,048,576
192.168.0.0/16	192.168.0.0 to 192.168.255.255	65,536

2. The middle block is the one people get wrong. 172.16.0.0/12 covers 172.16.x.x through 172.31.x.x, not 172.16 through 172.16.
3. That is exactly why the reader's traceroute looked odd. Hops showing 172.31.0.17, 172.26.22.235, 172.26.14.75 and 172.16.18.33 are all private addresses inside the same /12, used by the reader's ISP for its own core network.
4. Seeing private addresses in a traceroute is normal and proves nothing about a fault. It tells you the ISP numbers its own core privately, which is ordinary practice, especially with carrier-grade NAT.

■ 52.4.3 Reserved IPv4 ranges with their RFCs

Block	Purpose	RFC
0.0.0.0/8	"This network"	RFC 791
0.0.0.0/32	This host, this network	RFC 1122
10.0.0.0/8	Private use	RFC 1918
100.64.0.0/10	Shared, carrier NAT	RFC 6598
127.0.0.0/8	Loopback	RFC 1122
169.254.0.0/16	Link local, no DHCP	RFC 3927
172.16.0.0/12	Private use	RFC 1918
192.0.0.0/24	IETF assignments	RFC 6890
192.0.0.0/29	DS-Lite	RFC 6333

Block	Purpose	RFC
192.0.2.0/24	Docs, TEST-NET-1	RFC 5737
192.31.196.0/24	AS112 redirection	RFC 7535
192.52.193.0/24	AMT relay	RFC 7450
192.88.99.0/24	Old 6to4, deprecated	RFC 7526
192.168.0.0/16	Private use	RFC 1918
192.175.48.0/24	AS112 delegation	RFC 7534
198.18.0.0/15	Benchmarking only	RFC 2544
198.51.100.0/24	Docs, TEST-NET-2	RFC 5737
203.0.113.0/24	Docs, TEST-NET-3	RFC 5737
224.0.0.0/4	Multicast	RFC 5771
240.0.0.0/4	Reserved, future use	RFC 1112
255.255.255.255/32	Limited broadcast	RFC 919

1. The full list lives in the IANA IPv4 Special-Purpose Address Registry. The rows above are the ones you meet.
2. 169.254.x.x on your own machine means DHCP failed. The machine gave itself a link-local address because no server answered. If you see it, fix DHCP before you look at anything else.
3. 100.64.0.0/10 is the range your ISP uses when it puts many customers behind one public address. If your router's outside address is in this range, you are behind carrier-grade NAT and you cannot accept inbound connections.
4. Use 192.0.2.0/24, 198.51.100.0/24 and 203.0.113.0/24 in documentation and examples. They exist for that. Never invent an example address that belongs to somebody.

■ 52.4.4 CIDR from /8 to /32

1. The prefix length is how many leading bits are the network part. The rest is the host part.
2. Total addresses is 2 to the power of (32 minus the prefix). Usable hosts is that minus two, because the all-zeros address is the network itself and the all-ones address is the broadcast.

Prefix	Netmask	Total	Usable hosts
/8	255.0.0.0	16,777,216	16,777,214
/9	255.128.0.0	8,388,608	8,388,606
/10	255.192.0.0	4,194,304	4,194,302
/11	255.224.0.0	2,097,152	2,097,150
/12	255.240.0.0	1,048,576	1,048,574
/13	255.248.0.0	524,288	524,286
/14	255.252.0.0	262,144	262,142
/15	255.254.0.0	131,072	131,070
/16	255.255.0.0	65,536	65,534
/17	255.255.128.0	32,768	32,766
/18	255.255.192.0	16,384	16,382
/19	255.255.224.0	8,192	8,190
/20	255.255.240.0	4,096	4,094
/21	255.255.248.0	2,048	2,046
/22	255.255.252.0	1,024	1,022
/23	255.255.254.0	512	510
/24	255.255.255.0	256	254
/25	255.255.255.128	128	126

Prefix	Netmask	Total	Usable hosts
/26	255.255.255.192	64	62
/27	255.255.255.224	32	30
/28	255.255.255.240	16	14
/29	255.255.255.248	8	6
/30	255.255.255.252	4	2
/31	255.255.255.254	2	2
/32	255.255.255.255	1	1

- Two special cases. A /31 has no network or broadcast address; RFC 3021, from December 2000, allows both addresses to be hosts, which is how router-to-router links are numbered. A /32 is one single address, used for loopbacks and for host routes.
- Classless Inter-Domain Routing replaced the old class A, B and C scheme. It was defined in RFC 1518 and RFC 1519 in September 1993, and was a direct response to the routing table growing faster than router memory.

■ 52.4.5 The ports worth memorizing

- Port numbers are 16 bits, so 0 to 65535. RFC 6335 splits them into three ranges: 0 to 1023 system, 1024 to 49151 user or registered, and 49152 to 65535 dynamic or ephemeral.
- On Linux the outgoing ephemeral range is usually 32768 to 60999. On macOS it is 49152 to 65535. Both are settings, not standards.

Port	Service	Note
20, 21	FTP data, control	plaintext, obsolete
22	SSH, SFTP, scp	git over SSH
23	Telnet	plaintext, do not use
25	SMTP relay	often blocked at home
53	DNS	UDP and TCP
67, 68	DHCP server, client	UDP
69	TFTP	UDP, no auth
80	HTTP	plaintext
110	POP3	plaintext
123	NTP	UDP
143	IMAP	plaintext
161, 162	SNMP, SNMP trap	UDP
179	BGP	over TCP
389	LDAP	plaintext
443	HTTPS, and HTTP/3	TCP and UDP
445	SMB file sharing	never expose
465	SMTP over TLS	submission
514	syslog	UDP
587	SMTP submission	with STARTTLS
636	LDAPS	over TLS
993	IMAPS	over TLS
995	POP3S	over TLS
1080	SOCKS proxy	
1194	OpenVPN	UDP usually
1433	Microsoft SQL Server	

Port	Service	Note
1521	Oracle database	
3306	MySQL, MariaDB	
3389	RDP, remote desktop	never expose
5432	PostgreSQL	
5672	AMQP, RabbitMQ	
5900	VNC	
6379	Redis	no auth by default
6443	Kubernetes API server	
8080, 8000	HTTP alternate	dev convention
8443	HTTPS alternate	dev convention
9042	Cassandra	
9090	Prometheus	
9200	Elasticsearch	
11211	Memcached	
27017	MongoDB	
51820	WireGuard	UDP

- Note the two rows that mattered in the reader's session. Git over SSH uses port 22. Git over HTTPS uses port 443. In the reader's session one transport worked while the other did not, which is only possible because they are different ports to different endpoints.

■ 52.4.6 ICMP types worth knowing

Type	Name	When you see it
0	Echo Reply	a successful ping
3	Destination Unreachable	see the code below
5	Redirect	router says use another
8	Echo Request	the ping you sent
11	Time Exceeded	every traceroute hop
12	Parameter Problem	malformed header

- Type 3 carries a code that says exactly what was unreachable.

Code	Meaning
0	Network unreachable
1	Host unreachable
2	Protocol unreachable
3	Port unreachable
4	Fragmentation needed, DF set
13	Administratively prohibited

- Code 4 is the one that causes mysterious hangs. It is how path MTU discovery works. If a firewall blocks it, large packets vanish and small ones get through, and a page loads its HTML but never its images.
- Code 13 is a firewall openly telling you it refused. That is a polite refusal, and it is evidence. Silence is not.

- Traceroute works entirely on type 11 code 0. Each probe goes out with a time-to-live one higher than the last, and each router that decrements it to zero sends back type 11.
- Under IPv6, ICMPv6 is defined by RFC 4443 and the numbers change. Echo Request is 128, Echo Reply is 129, Packet Too Big is 2. ICMPv6 is not optional: blocking it breaks IPv6 entirely, because Neighbor Discovery rides on types 133 to 137.
- The reader's trace showed * * * from hop 13 to hop 20. That means no type 11 replies came back from those hops. It does not mean the packets died there. Many routers rate-limit or refuse to generate ICMP.

■ 52.4.7 HTTP methods

- Safe** means the method is not supposed to change anything on the server.
- Idempotent** means doing it twice has the same effect as doing it once.
- Both properties are promises made by the specification, currently RFC 9110, published June 2022. A server can break them. A well-behaved one does not.

Method	Safe	Idempotent	Usual use
GET	yes	yes	fetch a resource
HEAD	yes	yes	headers only
OPTIONS	yes	yes	what is allowed
TRACE	yes	yes	echo the request
PUT	no	yes	replace at this URL
DELETE	no	yes	remove
POST	no	no	create or act
PATCH	no	no	partial update
CONNECT	no	no	tunnel, used by proxies

- The practical consequence: a client may retry a GET, HEAD, PUT or DELETE automatically after a network failure. It must not retry a POST automatically, because it cannot know whether the first one landed.
- That is exactly the reasoning behind the git failure in section 52.8. A disconnect while reading the response tells you nothing about whether the write was applied.

■ 52.4.8 HTTP status codes

- The first digit gives the family. One hundred series is informational, two hundred success, three hundred redirection, four hundred client error, five hundred server error.
- Codes marked with a star are ones you should know without looking up.

Code	Name	Star
100	Continue	
101	Switching Protocols	star
102	Processing	
103	Early Hints	
200	OK	star
201	Created	star
202	Accepted	
203	Non-Authoritative Info	
204	No Content	star

Code	Name	Star
205	Reset Content	
206	Partial Content	star
207	Multi-Status	
208	Already Reported	
226	IM Used	
300	Multiple Choices	
301	Moved Permanently	star
302	Found	star
303	See Other	star
304	Not Modified	star
305	Use Proxy, deprecated	
307	Temporary Redirect	star
308	Permanent Redirect	star
400	Bad Request	star
401	Unauthorized	star
402	Payment Required	
403	Forbidden	star
404	Not Found	star
405	Method Not Allowed	star
406	Not Acceptable	
407	Proxy Auth Required	
408	Request Timeout	
409	Conflict	star
410	Gone	
411	Length Required	
412	Precondition Failed	star
413	Content Too Large	
414	URI Too Long	
415	Unsupported Media Type	
416	Range Not Satisfiable	
417	Expectation Failed	
418	I am a teapot	
421	Misdirected Request	
422	Unprocessable Content	star
423	Locked	
424	Failed Dependency	
425	Too Early	
426	Upgrade Required	
428	Precondition Required	
429	Too Many Requests	star
431	Headers Too Large	
451	Legal Reasons	star
500	Internal Server Error	star
501	Not Implemented	
502	Bad Gateway	star
503	Service Unavailable	star
504	Gateway Timeout	star
505	Version Not Supported	
506	Variant Also Negotiates	

Code	Name	Star
507	Insufficient Storage	
508	Loop Detected	
510	Not Extended	
511	Network Auth Required	star

- Four distinctions that people get wrong every day.
- 401 versus 403. A 401 means “I do not know who you are; send credentials”. A 403 means “I know who you are and you may not”. The reader’s missing workflow scope produced a 403-class refusal, not a 401.
- 502 versus 503 versus 504. A 502 means the gateway got a broken answer from the thing behind it. A 503 means the service is deliberately unavailable, often overloaded or in maintenance. A 504 means the thing behind it never answered in time.
- 301 versus 308, and 302 versus 307. The older pair, 301 and 302, historically allowed clients to change a POST into a GET when following. The newer pair, 307 and 308, forbid changing the method. Use 308 and 307 when you care.
- 418 is real and is a joke. It comes from RFC 2324, the Hyper Text Coffee Pot Control Protocol, published on 1 April 1998. RFC 7168 extended it to teapots in 2014. It is not part of HTTP.
- 451 is named for the novel Fahrenheit 451 and was standardized in RFC 7725 in February 2016. It means content is blocked for legal reasons.

■ 52.4.9 Common DNS record types

Type	Number	What it holds
A	1	one IPv4 address
AAAA	28	one IPv6 address
CNAME	5	an alias to another name
MX	15	mail server plus priority
NS	2	authoritative name server
PTR	12	name for an address
SOA	6	zone parameters
TXT	16	free text, SPF, DKIM
SRV	33	host, port, priority
CAA	257	who may issue certificates
DS	43	DNSSEC delegation signer
DNSKEY	48	DNSSEC public key
RRSIG	46	DNSSEC signature
NSEC3	50	DNSSEC proof of absence
SVCB	64	service binding
HTTPS	65	HTTPS service binding
TLSA	52	certificate pinning, DANE
SSHFP	44	SSH host key fingerprint

- A CNAME cannot exist at the zone apex, that is at `example.com` itself, because the apex must also hold SOA and NS records and a CNAME forbids any other record at the same name. Providers work around this with a non-standard record they call ALIAS or ANAME. That is a vendor convention, not a standard.

2. SVCB and HTTPS records were standardized in RFC 9460 in November 2023. The HTTPS record can tell a browser to use HTTP/3, and can carry the encrypted client hello parameters, before any connection is made.
3. TTL is a number of seconds attached to every record. It is a request, not a guarantee. Resolvers may cache longer or shorter.

■ 52.4.10 TCP flags and states

1. The TCP header has a flags field. The current specification is RFC 9293, from August 2022, which replaced RFC 793 from 1981.

Flag	Name	Meaning
SYN	Synchronize	start a connection
ACK	Acknowledge	I received up to here
FIN	Finish	I have no more to send
RST	Reset	tear down now, refused
PSH	Push	deliver to the app now
URG	Urgent	urgent pointer valid
ECE	ECN Echo	congestion signalled
CWR	Congestion Window Reduced	I slowed down

2. The three-way handshake is SYN, then SYN plus ACK, then ACK. The four-way close is FIN, ACK, FIN, ACK.
3. An RST in reply to a SYN means “nothing is listening on that port”. That is a refusal, and it is information. It arrives in milliseconds.
4. No reply at all to a SYN means the packet was dropped somewhere and nobody told you. That is what the reader saw. It costs the full timeout.

State	Meaning
CLOSED	no connection
LISTEN	waiting for a SYN
SYN_SENT	sent SYN, waiting
SYN_RECEIVED	got SYN, sent SYN-ACK
ESTABLISHED	data can flow
FIN_WAIT_1	we sent FIN
FIN_WAIT_2	our FIN was acked
CLOSE_WAIT	they sent FIN, we have not
CLOSING	both sent FIN together
LAST_ACK	we sent our FIN last
TIME_WAIT	waiting out stray packets

5. Two states are diagnostic on their own.
6. Many sockets stuck in CLOSE_WAIT means your own application is not calling close. That is an application bug, not a network problem.
7. Many sockets in TIME_WAIT is normal on a busy client. The specification says wait twice the maximum segment lifetime, which by the letter of the standard is four minutes. Linux uses a fixed sixty seconds and does not let you change it without recompiling.
8. A connection that sits in SYN_SENT and never leaves it is the exact signature of the reader's fault: the SYN went out, nothing came back.

52.5 Networking command cheat sheet

■ 52.5.1 How to read this section

1. Every entry below gives the macOS form first, because that is the machine in the running example, then the Linux difference on the next line.
2. Then two lines that most cheat sheets omit.
3. **Proves** is the set of things that must be true for the command to have succeeded. If it passed, all of these are true.
4. **Does not prove** is the trap. These are things people assume a pass means and that it does not mean.
5. Getting this second line right is the difference between diagnosing a problem and guessing at it.

■ 52.5.2 The commands at a glance

Command	Layer it tests	Needs network
ifconfig or ip addr	your own interface	no
arp -a	local link, layer 2	local only
netstat -rn or ip route	your routing table	no
ping	ICMP reachability	yes
traceroute	path, hop by hop	yes
mtr	path plus loss over time	yes
dig, host, nslookup	DNS resolution	yes
scutil --dns	which resolver macOS uses	no
nc -vz	one TCP port	yes
telnet	one TCP port, old	yes
nmap	many ports at once	yes
curl -v	full request, TLS, HTTP	yes
wget	file download	yes
openssl s_client	TLS handshake only	yes
tcpdump	actual packets on the wire	no
lsof -i	which process holds a port	no
ss or netstat -an	socket states	no
whois	who owns a name or address	yes
networksetup	macOS network settings	no

■ 52.5.3 ping

1. macOS: ping -c 5 github.com
2. Linux: same, plus -w 5 sets a total deadline in seconds.
3. Useful flags, macOS: -c count, -i interval, -s size, -t timeout in seconds for the whole run, -D set the do-not-fragment bit.
4. Careful: on Linux -t means time-to-live, not timeout. Same letter, different meaning. Linux uses -M do for do-not-fragment.
5. Proves: your interface is up, your route exists, the target address is reachable by ICMP, and something at that address answers ICMP.
6. Does not prove: that any TCP port is open, that the web server is running, that DNS is correct, or that the service you want works. Ping tests ICMP and only ICMP.
7. A failed ping proves even less. Very many hosts and networks drop ICMP on purpose. Cloudflare, AWS load balancers and corporate firewalls commonly do. A ping that fails while curl succeeds is completely normal.

8. Path MTU test, macOS: `ping -D -s 1472 github.com`. If 1472 fails and 1400 works, something on the path has a smaller MTU. Add 28 bytes of headers to get the real figure: 1472 plus 28 is 1500.

■ 52.5.4 traceroute

1. macOS: `traceroute -n github.com`
2. Linux: `traceroute -n github.com`, and `tracepath` is a simpler alternative that needs no privileges.
3. Useful flags: `-n` no reverse lookups so it runs fast, `-m 30` maximum hops, `-q 1` one probe per hop, `-w 2` wait two seconds, `-I` use ICMP probes instead of UDP.
4. macOS default probes are UDP to high port numbers. Linux default is the same. Windows `tracert` uses ICMP by default. This matters, because firewalls filter the three differently.
5. TCP mode is the most useful on a modern network. Linux: `traceroute -T -p 443 github.com`. macOS: `traceroute -P TCP -p 443 github.com`.
6. Proves: which routers along the path are willing to answer, in order, up to the last one that replied.
7. Does not prove: that the path is symmetric, that the reply came back the same way it went, that a hop with no reply is broken, or that the last replying hop is where the trouble is.
8. The reader's trace stopped replying after hop 12, `ae106-0.rwa04.pnq20.ntwk.msn.net (104.44.20.42)`. That does not mean hop 12 dropped the traffic. It means hops beyond it did not send back ICMP time-exceeded messages, which is extremely common.
9. Hops showing more than one address, as hops 5, 6, 8, 9 and 11 did in the reader's trace, mean the traffic is being spread over parallel links. That is load balancing working correctly, not a fault.

■ 52.5.5 mtr

1. `mtr` combines `ping` and `traceroute` and keeps running, so it shows loss per hop over time.
2. macOS: not installed by default. Install with `brew install mtr`, then run with `sudo mtr github.com` because it needs raw sockets.
3. Linux: `mtr github.com`, often already installed.
4. Report mode, for pasting into a ticket: `mtr -rcw 100 github.com`. That means report, wide output, one hundred cycles.
5. TCP mode: `mtr -T -P 443 github.com`.
6. Proves: sustained loss or latency at and beyond a specific hop.
7. Does not prove: that loss shown at a middle hop is real. Routers deliberately rate-limit the ICMP replies they generate for themselves while forwarding everything else perfectly. Only loss that continues to the final hop and stays there is real loss.
8. That single rule saves more wasted hours than any other line in this chapter. Read the last row, not the middle rows.

■ 52.5.6 dig, host and nslookup

1. `dig` is the precise one. Use it.
2. macOS: `dig github.com`. On recent macOS versions Apple has removed the BIND lookup tools from the base install. If `dig` is missing, get it with `brew install bind`.
3. Linux: `dig github.com`, from the `dnsutils` or `bind-utils` package.

```
dig +short github.com           # just the answer
dig +short A github.com         # only IPv4
dig +short AAAA github.com      # only IPv6
dig @1.1.1.1 github.com         # ask a specific resolver
dig @8.8.8.8 github.com         # ask a different one
dig +trace github.com           # walk from the root down
dig -x 20.207.73.82            # reverse lookup
```



```
dig +noall +answer github.com      # answer section only
dig NS github.com                  # nameservers
dig MX github.com                  # mail servers
dig TXT github.com                 # text records
dig +tcp github.com                # force TCP not UDP
dig +dnssec github.com             # ask for signatures
```

4. Proves: that a resolver answered, what it answered, and how long the answer may be cached.
5. Does not prove: that the address is reachable, that it is the right address for you, or that your operating system will use it. `dig` talks to the resolver directly and bypasses the system cache.
6. That last point matters on macOS, which has its own resolver daemon. To see what the system will actually do, use `dscacheutil -q host -a name github.com`, and to see which resolvers are configured use `scutil --dns`.
7. `host github.com` gives a one-line answer. `nslookup github.com 1.1.1.1` works everywhere including Windows. Both are fine for a quick check and worse than `dig` for anything else.
8. In the reader's session, name resolution worked: `github.com` resolved to `20.207.73.82` through the Cloudflare resolver at `1.1.1.1`. That single fact eliminated DNS as the cause.

■ 52.5.7 curl

1. The single most useful network command on any machine.
2. macOS and Linux syntax are the same. macOS has shipped `curl` since 10.4.

```
curl -v https://github.com          # show the whole conversation
curl -I https://github.com          # headers only, HEAD request
curl -sS -o /dev/null -w '%{http_code}\n' https://github.com
curl -L https://example.com         # follow redirects
curl --max-time 10 https://github.com # give up after 10 seconds
curl --connect-timeout 3 https://x.io # give up connecting after 3
curl -4 https://github.com          # force IPv4
curl -6 https://github.com          # force IPv6
curl --resolve github.com:443:20.207.73.82 https://github.com
curl -H 'Accept: application/json' https://api.github.com
curl -X POST -d 'a=1' https://example.com/f
curl -k https://self-signed.example # skip cert check
curl --http1.1 https://github.com   # force HTTP/1.1
curl -x http://proxy:3128 https://x.io # go via a proxy
```

3. The timing breakdown is the hidden gem. It tells you which stage was slow.

```
curl -sS -o /dev/null -w \
'dns %{time_namelookup} conn %{time_connect} tls
%{time_appconnect} first %{time_starttransfer} total
%{time_total}\n' https://github.com
```

4. Proves, when it returns a status code: DNS resolved, TCP connected, TLS completed if the URL was HTTPS, the request was sent, and a server answered.
5. Does not prove: that the answer is correct, that the site works in a browser, or that the certificate is trusted if you passed `-k`.
6. A hang with no output after Trying `<address>:443...` proves the TCP handshake never completed. That is a layer 4 or below problem and nothing above it can be at fault yet.
7. That is exactly what the reader saw. `curl -v https://github.com` printed Trying `20.207.73.82:443...` and then produced nothing for fifteen seconds. No RST, no ICMP unreachable, no TLS error. Silence.

8. Silence is a specific finding. A server that is down sends RST. A firewall that refuses politely sends ICMP type 3 code 13. Nothing at all means a silent drop somewhere on the path.

■ 52.5.8 wget

1. macOS: not installed by default. `brew install wget`.
2. Linux: usually present.

```
wget https://example.com/file.tar.gz # save with its own name
wget -O out.bin https://example.com/f # save under a chosen name
wget -O - https://example.com        # print to standard output
wget --spider https://example.com    # exists? do not save
wget -c https://example.com/big.iso  # continue a partial download
wget -T 10 -t 3 https://example.com  # 10s timeout, 3 tries
```

3. Proves and does not prove: the same as curl.
4. Use wget for downloading files and mirroring. Use curl for inspecting a request. That split is a convention, not a rule.

■ 52.5.9 nc, the netcat tool

1. The cleanest way to test one TCP port and nothing else.
2. macOS: `nc -vz github.com 443`. Apple ships its own build of the OpenBSD version.
3. Linux: `nc -vz -w 3 github.com 443` with the OpenBSD build.

```
nc -vz github.com 443          # is port 443 open
nc -vz -w 3 github.com 22      # with a 3 second timeout
nc -vzu 1.1.1.1 53             # UDP, far less reliable
nc -l 8080                     # listen on 8080, a fake server
nc -vz github.com 20-25        # a small range
```

4. Proves, on success: the full TCP three-way handshake completed to that address on that port.
5. Does not prove: that the service behind the port speaks the protocol you expect, that TLS will succeed, or that your credentials are valid.
6. This is the correct tool for separating “the network is blocked” from “the application is refusing me”. If `nc -vz` succeeds and `curl` fails, the problem is above layer 4.
7. UDP testing with `-u` is close to useless on its own, because a silent UDP port and an open UDP port look identical. Only the application reply proves anything.

■ 52.5.10 telnet

1. Removed from macOS since High Sierra, version 10.13, in 2017. Install with `brew install telnet` or, better, use `nc` instead.
2. `telnet github.com 443` then type text. Historically used to speak HTTP by hand.
3. Proves: a TCP connection opened, and you can type raw bytes into it.
4. Does not prove: anything about encryption. Telnet is plaintext. Never use the telnet protocol itself, on port 23, for real work. It sends passwords in the clear. SSH replaced it in 1995.

■ 52.5.11 nmap

1. Not installed by default anywhere. `brew install nmap` on macOS, `apt install nmap` on Debian and Ubuntu.

```
nmap -Pn -p 443 github.com      # one port, skip host discovery
nmap -p 22,80,443 192.168.0.1   # a few ports
nmap -p- 192.168.0.1           # all 65535 ports, slow
nmap -sV -p 443 example.com     # try to identify the service
```

```
nmap -sn 192.168.0.0/24      # who is on my network
nmap --top-ports 100 192.168.0.1 # the common ones only
sudo nmap -sS -p 443 example.com # half-open scan, needs root
```

2. Proves: which of the scanned ports answered, and how.
3. Does not prove: that a filtered result means blocked by the target. It may be blocked anywhere along the path, including by your own network.
4. One legal and social rule, and it is not optional. Scan only machines you own or have written permission to scan. Port scanning other people's networks is, in many countries including India under the Information Technology Act 2000, potentially an offence, and at minimum it will get your connection blocked.

■ 52.5.12 netstat, ss and lsof

1. These look at your own machine, not the network. None of them need connectivity.
2. macOS: netstat -an -p tcp shows all TCP sockets. netstat -rn shows the routing table. netstat -i shows interface counters. netstat -s shows protocol statistics.
3. Linux: netstat is deprecated in favour of ss, from the iproute2 package.

```
ss -tulpn      # Linux: listening TCP+UDP, pids
ss -tn state established # Linux: established TCP only
ss -s          # Linux: summary counts
netstat -an -p tcp # macOS: all TCP sockets
netstat -rn     # both: routing table
lsof -i -P -n   # both: sockets with process names
lsof -nP -iTCP:443 # both: who is using port 443
lsof -nP -iTCP -sTCP:LISTEN # both: what is listening
```

4. Proves: exactly which local process holds which port, and what state each connection is in.
5. Does not prove: anything about the remote end. A socket shown as ESTABLISHED locally can be dead, if the peer vanished without sending FIN or RST. Only traffic proves liveness.
6. On macOS, lsof needs sudo to see other users' processes.

■ 52.5.13 arp

1. macOS and Linux: arp -a lists the address resolution cache, which maps IPv4 addresses to hardware addresses on your local network segment.
2. Linux modern form: ip neigh show.
3. sudo arp -d 192.168.0.1 deletes one entry, which forces a fresh lookup.
4. Proves: which machines your computer has recently talked to on the same local segment, and their MAC addresses.
5. Does not prove: anything beyond your own network. ARP does not cross a router. If the gateway is in the table with a MAC address, your link to the router is working.
6. arp -a showing 192.168.0.1 with a valid MAC address would have been the first thing to check in the reader's case, and it would have passed. The local network was fine.

■ 52.5.14 ifconfig, ip and route

1. macOS: ifconfig and ifconfig en0. Apple has not adopted the ip command.
2. Linux: ip addr show, or the short form ip -br a.

```
ifconfig -a      # macOS: every interface
ifconfig en0     # macOS: just Wi-Fi
ipconfig getifaddr en0 # macOS: just the IPv4 address
ipconfig getpacket en0 # macOS: the whole DHCP lease
```

```
netstat -rn                # macOS: routing table
route -n get default       # macOS: which route wins
ip -br a                  # Linux: brief addresses
ip link show               # Linux: layer 2 state
ip route show              # Linux: routing table
ip route get 20.207.73.82  # Linux: which route wins
```

3. On macOS, en0 is usually Wi-Fi on a laptop and Ethernet on a desktop. lo0 is loopback. utun0 and upwards are tunnel interfaces, created by VPNs, by iCloud Private Relay, and by some corporate agents.
4. The reader's machine showed several utun interfaces. That is worth noticing but does not prove a VPN was routing the traffic. Check with `route -n get default` and see which interface the default route uses.
5. Proves: your own addresses, your own routes, your own link state.
6. Does not prove: that the route leads anywhere.

■ 52.5.15 tcpdump

1. The ground truth. Everything else is an interpretation; this is the packets.
2. macOS: `sudo tcpdump -i en0 -n host 20.207.73.82`
3. Linux: same, with the interface usually named eth0, ens3 or wlan0.

```
sudo tcpdump -i en0 -n host 20.207.73.82
sudo tcpdump -i en0 -nn port 443
sudo tcpdump -i any -n icmp
sudo tcpdump -i en0 -n -c 20 tcp port 443
sudo tcpdump -i en0 -w capture.pcap          # save for Wireshark
sudo tcpdump -i en0 -n 'tcp[tcpflags] & tcp-syn != 0'
sudo tcpdump -i en0 -n -A port 80            # show payload as text
```

4. Flags that matter: -n no name lookups, -nn also no port names, -i any all interfaces on Linux, -c N stop after N packets, -s 0 full packet, -w file write a capture file, -A print payload as ASCII, -X print hex and ASCII.
5. Proves: exactly what left your machine and exactly what came back.
6. Does not prove: what happened in the middle. You see your end of the wire and nothing else.
7. In the reader's case, tcpdump would have shown repeated SYN packets going out to 20.207.73.82 on port 443, with retransmissions at roughly one, three and seven seconds, and nothing at all coming back. That is the signature of a silent drop, and it is different from every other failure.

■ 52.5.16 openssl s_client

1. Tests the TLS handshake by itself, with nothing above it.

```
openssl s_client -connect github.com:443 -servername github.com
openssl s_client -connect github.com:443 -showcerts
openssl s_client -connect github.com:443 -tls1_2
openssl s_client -connect github.com:443 -tls1_3
echo | openssl s_client -connect github.com:443 2>/dev/null \
| openssl x509 -noout -subject -issuer -dates
```

2. -servername sets Server Name Indication, which almost every modern host requires. Without it you may get the wrong certificate or a failure.
3. Proves: TCP connected, a TLS handshake completed, which version and cipher were agreed, and which certificate chain was presented.

4. Does not prove: that the certificate is trusted by your operating system. The `openssl` command uses its own trust store, which on macOS is usually not the system keychain. A certificate that `openssl` rejects may work fine in Safari, and the reverse.
5. Check expiry dates with the last command above. Certificate expiry is one of the most common causes of a site that worked yesterday and not today.
6. Since 2024 the industry has been shortening maximum certificate lifetimes. The CA/Browser Forum voted in April 2025 to reduce the maximum from 398 days down in stages, reaching 47 days by March 2029. Expiry checks are getting more important, not less.

■ 52.5.17 whois

1. `whois github.com` looks up who registered a domain name.
2. `whois 20.207.73.82` looks up who owns an address block.
3. Proves: what a registry says. Registry data is often stale, and since the European General Data Protection Regulation took effect in May 2018, most personal contact details in domain records are hidden.
4. Does not prove: who is operating the machine today.
5. In the reader's session, `whois` on `20.207.73.82` would show a Microsoft range. That is consistent with GitHub being owned by Microsoft, which acquired it in 2018 for 7.5 billion dollars in stock, and fronting traffic through Microsoft's network edge.
6. The traceroute agreed: from hop 7 onward the names all ended in `ntwk.msn.net`, which is Microsoft's backbone naming.

■ 52.5.18 macOS-only tools

1. `scutil --dns` shows every resolver the system will use, in order, with the search domains and the interface each is bound to. This is the truth about macOS DNS; `/etc/resolv.conf` on macOS is a compatibility file and can be misleading.
2. `scutil --nwi` shows network information: which interfaces are usable, and whether IPv4 and IPv6 are available on each.
3. `scutil --proxy` shows the proxy configuration.
4. `networksetup -listallnetworkservices` lists the named services such as Wi-Fi and Ethernet.
5. `networksetup -getinfo "Wi-Fi"` shows address, mask, router and MAC.
6. `networksetup -getdnsservers "Wi-Fi"` shows configured resolvers.
7. `sudo networksetup -setdnsservers "Wi-Fi" 1.1.1.1 8.8.8.8` sets them.
8. `sudo networksetup -setdnsservers "Wi-Fi"` Empty puts it back to DHCP.
9. `sudo dscacheutil -flushcache`; `sudo killall -HUP mDNSResponder` clears the DNS cache. Both halves are needed on modern macOS.
10. `networkQuality` runs Apple's built-in throughput and responsiveness test. It has shipped since macOS 12 Monterey, released in 2021.
11. `ipconfig getpacket en0` prints the entire DHCP lease including the offered DNS servers, which is how you tell what the router wanted you to use.
12. The reader was using `1.1.1.1`, Cloudflare's public resolver, rather than the router. `scutil --dns` is what proves that.

■ 52.5.19 Worked example: the reader's own five commands

1. Here is the exact sequence that settled the reader's case, with what each step eliminated.

```
Step 1 ping 192.168.0.1
      -> replies. Local network and router: fine.
```

```

Step 2  dig +short github.com
        -> 20.207.73.82.  DNS: fine.

Step 3  curl -v --max-time 15 https://github.com
        -> "Trying 20.207.73.82:443..." then nothing.
        -> TCP handshake never completed.

Step 4  traceroute -n github.com
        -> reaches Microsoft's Pune edge, then * * *.
        -> path exists at least that far.

Step 5  same URL on mobile data, same phone
        -> loads instantly.
        -> the server is up; the destination is fine.

```

2. What that combination proves: the failure is specific to the path between this home connection and that destination, at the transport layer, and it is a silent drop rather than a refusal.
3. What it does not prove: who is dropping the packets, or why. The evidence does not name a party, and you should not either.

52.6 Decision tree: I cannot reach a site

■ 52.6.1 The shape of the procedure

1. Work upward through the layers. Never start in the middle.
2. Each step either passes, which eliminates everything below it, or fails, which tells you exactly where to stop looking.
3. Do not skip a step because you are sure it is fine. The reader's case looked like a DNS problem and was not.

```

START: "I cannot reach a site"
    |
    1. Is any other site reachable?
       no -> your machine or link. Go to A.
       yes v
    2. Does the name resolve?
       no -> DNS. Go to B.
       yes v
    3. Does TCP connect on the port?
       no -> path or firewall. Go to C.
       yes v
    4. Does TLS complete?
       no -> certificate or version. Go to D.
       yes v
    5. Does HTTP return a status code?
       no -> the app is hung. Go to E.
       yes v
    6. Is the status what you expected?
       no -> application layer. Go to F.
       yes -> it works. The problem is elsewhere.
           Go to G.

```

■ 52.6.2 Step 1: is anything reachable at all

1. Run `ping -c 3 192.168.0.1`, using your own gateway address.
2. Find your gateway with `route -n get default` on macOS or `ip route show` on Linux.
3. If the gateway does not answer, stop. The problem is your machine, your cable, or your Wi-Fi. Check `ifconfig en0` for an address.
4. If your address begins 169.254., DHCP failed. Nothing else matters until that is fixed. Renew the lease or restart the interface.
5. If the gateway answers, run `ping -c 3 1.1.1.1`.
6. If a public address answers but names do not resolve, jump straight to step 2. That combination is the classic DNS-only failure.
7. If the gateway answers and 1.1.1.1 does not, the problem is between your router and the internet. Check the router's own status page and the physical line.
8. Branch A conclusion: local. Fix the interface, the lease, the cable or the router before going further.

■ 52.6.3 Step 2: does the name resolve

1. Run `dig +short github.com`.
2. If you get an address, DNS works. Note the address and move on.
3. If you get nothing, ask a different resolver: `dig @1.1.1.1 +short github.com` and `dig @8.8.8.8 +short github.com`.
4. If a public resolver answers and your configured one does not, your resolver is the problem. On macOS, check with `scutil --dns` and change it with `networksetup -setdnsservers`.
5. If no resolver answers for that one name but others resolve, the domain itself may be broken. Check with `dig +trace github.com`, which walks from the root servers down and shows where the chain stops.
6. If the answer looks wrong, for example a private address for a public site, suspect a captive portal, a filtering resolver, or a modified hosts file. Check `/etc/hosts` on macOS and Linux.
7. Clear the cache and try again: on macOS, `sudo dscacheutil -flushcache;` `sudo killall -HUP mDNSResponder`.
8. Branch B conclusion: DNS. Nothing above this layer can work until a name turns into an address.

■ 52.6.4 Step 3: does TCP connect

1. Run `nc -vz github.com 443`.
2. Success means the three-way handshake completed. Move to step 4.
3. Now read the failure carefully, because the three failure modes mean completely different things.
4. **Connection refused**, immediately. You received a TCP RST. Something is there and it said no. The host is up; the port is closed. Suspect a service that is not running, or the wrong port.
5. **No route to host** or **host unreachable**, quickly. You received an ICMP type 3. A router along the way openly told you it could not deliver. Suspect routing or a firewall that refuses politely.
6. **Timeout**, after many seconds, with no message at all. Nothing came back. This is a silent drop. It is the hardest case and it is the reader's case.
7. Confirm which of the three you have with `curl -v --max-time 15` and, if you can, with `sudo tcpdump -i en0 -n host <address> and port 443` in another window.
8. If it is a timeout, test whether it is that one destination or all destinations: try `nc -vz 1.1.1.1 443` and `nc -vz google.com 443`.
9. Test whether it is that one port: try `nc -vz github.com 22` and `nc -vz github.com 80`.
10. Test whether it is that one network path: use a different network. Mobile data on a phone, with tethering or with the same URL typed into the phone, is the cleanest test available to a home user.

11. If it works on a different network and not on yours, the destination is fine and the path is not.
12. Branch C conclusion: transport or path. See section 52.6.8 for what to do next, because this is where the reader ended up.

■ 52.6.5 Step 4: does TLS complete

1. Run `openssl s_client -connect github.com:443 -servername github.com`.
2. Success prints a certificate chain and a line saying which protocol version and cipher were negotiated.
3. `certificate verify failed` means the chain did not validate. Look at the reason code printed on the same line.
4. `certificate has expired` means exactly that. Check the dates with `openssl x509 -noout -dates`.
5. `unable to get local issuer certificate` usually means an intermediate certificate is missing from the server's chain, or your trust store is old.
6. `wrong version number or no protocols available` means a version mismatch. Old servers that only speak TLS 1.0 are refused by modern clients. Test with `-tls1_2` and `-tls1_3` explicitly.
7. A hostname mismatch means the certificate does not cover the name you asked for. Check the subject alternative names with `openssl x509 -noout -text` and look for the SAN extension.
8. If TLS fails only in your browser but succeeds with `openssl`, suspect a corporate inspection proxy that is re-signing traffic, or a VPN.
9. Branch D conclusion: TLS. The network is fine; the trust is not.

■ 52.6.6 Step 5 and 6: does HTTP answer, and correctly

1. Run `curl -sS -o /dev/null -w '%{http_code} %{time_total}\n'` against the URL.
2. No status code and a long time means the connection opened and the server never replied. That is an application hang, not a network fault. Branch E.
3. A 502, 503 or 504 means you reached a gateway and the thing behind it is broken or overloaded. That is the operator's problem, not yours. Branch F.
4. A 401 or 403 means you reached the right service and it refused you. Check your credentials, token, or scopes. Branch F.
5. A 404 means you reached the right service and asked for the wrong path.
6. A 429 means you are being rate limited. Read the `Retry-After` header.
7. A 200 with wrong content means caching, a stale CDN copy, or the wrong virtual host. Try `curl -H 'Cache-Control: no-cache'` and try the origin directly with `--resolve`.
8. Branch F conclusion: application layer. The whole network stack worked.

■ 52.6.7 Everything passes and it still fails

1. This happens, and it has a small number of causes. Work through them in this order.
2. It works from the command line but not in the browser. Suspect a browser extension, a proxy setting inside the browser, or a service worker holding a cached broken response. Test in a private window with extensions off.
3. It works over IPv4 and not IPv6, or the reverse. Force each with `curl -4` and `curl -6`. Broken IPv6 with working IPv4 is common and produces long pauses because the browser tries IPv6 first.
4. The reader's machine reported IPv6: (none), which removes this branch from the reader's own case entirely.
5. It works for small requests and fails for large ones. That is an MTU problem. Test with `ping -D -s 1472` and step the size down. If 1400 works and 1472 does not, something on the path has a smaller MTU and is not sending the ICMP message that would let path MTU discovery work.

6. It works sometimes. Run `mttr -rwc 200` and look at the loss on the final row only. Intermittent means loss, congestion, or a load-balanced path where one of several parallel links is broken.
7. That last case is worth naming: if a hop shows several addresses, as hops 5, 6, 8, 9 and 11 did in the reader's trace, then some flows take one link and some take another. A single bad link out of four gives you a fault that appears for one in four connections and looks random.
8. It works for other people on the same network. Compare `scutil --dns`, `route -n get default` and the proxy settings between the two machines.
9. It stopped working at a specific time. Ask what changed then: a certificate expiry, a DNS TTL running out, a scheduled firewall rule, a router reboot.
10. It works with a VPN on. That proves the destination and the server are fine and that the direct path is the problem, because the VPN carries your traffic through a different route. It does not prove who is blocking.

■ 52.6.8 Exactly what to report to an ISP

1. Support staff cannot act on "the internet is not working". They can act on a reproducible measurement. Give them these seven things, in this order.
2. Your own public address and the time, including the time zone. Get the address with `curl -4 ifconfig.me` when you have any working path, or from the router's status page.
3. The destination name and the exact address it resolves to on your machine. For the reader: `github.com` resolving to `20.207.73.82`.
4. The exact symptom, in protocol terms. Not "it is slow". Say: "TCP SYN to `20.207.73.82` port 443 receives no response of any kind, no RST and no ICMP unreachable, and times out after 15 seconds."
5. A full traceroute, pasted verbatim, with `-n` so it is fast and unambiguous. Include the hop where replies stop.
6. The control test that proves it is not the destination. For the reader: "The same URL loads instantly over mobile data on the same phone, at the same moment."
7. The control test that proves it is not everything. For example: "Other HTTPS sites work normally from the same machine at the same time."
8. An `mttr` report over at least 100 cycles, in report mode, so they can see loss over time rather than one snapshot.
9. What you have already ruled out, listed plainly: local network, DNS, the destination server, IPv6.
10. One sentence of what you want: "Please check whether traffic from my connection to `20.207.73.82` on port 443 is being dropped in your network."
11. Do not accuse anyone. The evidence supports a silent drop somewhere on that path. It does not identify who is doing it, and claiming otherwise makes the ticket easier to dismiss.
12. Keep a copy of every command output with timestamps. If the fault is intermittent, that record is the only thing that will get it taken seriously.

52.7 Git command cheat sheet

■ 52.7.1 How to read the two extra columns

1. **Net** says whether the command needs the network. If it says no, the command works with the cable unplugged. That is most of git.
2. **Risk** says whether the command can destroy work that is not recoverable from the reflog.
3. Low means safe. Medium means it rewrites history but the old commits stay reachable through git reflog for 90 days by default. High means it can delete uncommitted changes or untracked files, which nothing can recover.

■ 52.7.2 Setup and configuration

Command	What it does	Net	Risk
git init	make a new repository	no	low
git init --bare	make a server-side repo	no	low
git clone URL	copy a repo and history	yes	low
git clone --mirror	exact copy of all refs	yes	low
git clone --depth 1	shallow, latest commit	yes	low
git config --global x y	set a user-wide setting	no	low
git config --list	show every setting	no	low
git config --list --show-origin	and which file set it	no	low
git remote -v	list remotes and URLs	no	low
git remote add name URL	add a remote	no	low
git remote set-url name URL	change a remote URL	no	low
git help command	full documentation	no	low

1. The three settings worth doing on a new machine, once:

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
git config --global init.defaultBranch main
git config --global pull.rebase false
git config --global core.editor "vim"
```

■ 52.7.3 Local work

Command	What it does	Net	Risk
git status	what changed, what is staged	no	low
git status -sb	the same, one line each	no	low
git add file	stage a file	no	low
git add -p	stage chosen hunks	no	low
git add -A	stage everything	no	low
git commit -m "msg"	record the staged snapshot	no	low
git commit --amend	replace the last commit	no	med
git diff	working tree vs index	no	low
git diff --staged	index vs last commit	no	low
git diff main..feature	between two branches	no	low
git rm file	delete and stage the delete	no	high
git mv old new	rename and stage it	no	low
git restore file	discard working changes	no	high
git restore --staged file	unstage, keep changes	no	low

Command	What it does	Net	Risk
git stash	park changes, clean tree	no	low
git stash pop	bring them back	no	low
git stash list	see what is parked	no	low
git clean -n	show what would be deleted	no	low
git clean -fd	delete untracked files	no	high

1. `git restore file` and `git clean -fd` are the two most dangerous commands in daily use. They delete things git has never seen. The reflog cannot help you. Always run `git clean -n` first.

■ 52.7.4 History and inspection

Command	What it does	Net	Risk
git log	full history	no	low
git log --oneline	one line per commit	no	low
git log --oneline --graph --all	the branch shape	no	low
git log -p	history with diffs	no	low
git log -S "text"	commits that changed text	no	low
git log --since="2 weeks"	by date	no	low
git show SHA	one commit in full	no	low
git blame file	who last touched each line	no	low
git shortlog -sn	commits per author	no	low
git describe --tags	nearest tag plus distance	no	low
git rev-list --count HEAD	how many commits	no	low
git bisect start	begin a binary search	no	low
git bisect good SHA	mark a known-good point	no	low
git bisect bad	mark the current as bad	no	low
git bisect reset	stop and go back	no	low
git reflog	every position HEAD held	no	low
git range-diff a...b	compare two versions of a series	no	low

1. `git reflog` is the undo button for almost everything. If a rebase or reset went wrong, find the old SHA in the reflog and `git reset --hard` to it. Entries expire after 90 days for reachable commits and 30 days for unreachable ones, by default.

■ 52.7.5 Branching

Command	What it does	Net	Risk
git branch	list local branches	no	low
git branch -a	include remote-tracking	no	low
git branch -vv	show tracking and ahead/behind	no	low
git switch name	move to a branch	no	low
git switch -c name	create and move to it	no	low
git switch -	back to the previous branch	no	low
git checkout name	the older form of switch	no	low
git branch -d name	delete a merged branch	no	low
git branch -D name	delete regardless	no	med
git merge feature	join a branch in	no	low
git merge --abort	undo a conflicted merge	no	low

Command	What it does	Net	Risk
git merge --no-ff	always make a merge commit	no	low
git rebase main	replay commits on a new base	no	med
git rebase -i HEAD~5	edit the last five commits	no	med
git rebase --abort	undo an in-progress rebase	no	low
git cherry-pick SHA	copy one commit here	no	low
git merge-base a b	the common ancestor	no	low
git tag -a v1.0 -m "msg"	make an annotated tag	no	low
git worktree add ../dir br	second checkout, same repo	no	low

1. The one fact about rebase that matters. Rebase does not move commits. It creates new commit objects with the same changes and new parents, so every rebased commit gets a new SHA.
2. That is exactly what the reader saw: a local branch rebased onto a moved main came out with a different SHA, because it was a different object.
3. Merge preserves history and adds a commit. Rebase rewrites history and adds none. Neither is correct in general; teams pick one by convention.

■ 52.7.6 Remotes and the network

Command	What it does	Net	Risk
git fetch	download objects and refs	yes	low
git fetch --all --prune	all remotes, drop dead refs	yes	low
git pull	fetch then merge	yes	low
git pull --rebase	fetch then rebase	yes	med
git push	send commits to a remote	yes	low
git push -u origin br	push and set upstream	yes	low
git push --force	overwrite the remote branch	yes	high
git push --force-with-lease	force, but refuse if it moved	yes	med
git push --tags	send tags too	yes	low
git ls-remote origin	list remote refs, live	yes	low
git remote show origin	remote details	yes	low
git remote update	fetch all remotes	yes	low
git remote prune origin	delete stale tracking refs	yes	low
git bundle create f.bundle --all	history in one file	no	low
git archive HEAD -o src.tar	export files, no history	no	low
git submodule update --init	fetch nested repos	yes	low

1. Only the rows marked yes need the network. Everything in 52.7.3, 52.7.4, 52.7.5 and 52.7.7 works offline, because the whole history is already in .git.

2. That is why the reader could keep committing to branches during a total outage and push them later. Commits are local writes to local files.
3. `git ls-remote origin` is the only way to learn the true current state of a remote. `git rev-parse origin/main` reads a cached value from your last fetch.
4. In the reader's session, `origin/main` showed `0a95cc8` during the outage. That was the remote-tracking ref, which is a note about the past, not a live fact.
5. Never use `--force` on a shared branch. Use `--force-with-lease`, which checks that the remote is still where you last saw it and refuses if somebody else pushed in the meantime.

■ 52.7.7 Undoing

Command	What it does	Net	Risk
<code>git restore --staged file</code>	unstage, keep the change	no	low
<code>git restore file</code>	throw away the change	no	high
<code>git reset --soft HEAD~1</code>	undo commit, keep staged	no	med
<code>git reset --mixed HEAD~1</code>	undo commit, keep in tree	no	med
<code>git reset --hard HEAD~1</code>	undo commit, delete changes	no	high
<code>git reset --hard origin/main</code>	make local match remote	no	high
<code>git revert SHA</code>	new commit that undoes one	no	low
<code>git commit --amend</code>	fix the last commit	no	med
<code>git reflog then reset --hard</code>	recover a lost position	no	med
<code>git checkout SHA -- file</code>	one file from a past commit	no	high
<code>git clean -fdx</code>	delete untracked and ignored	no	high

1. The rule that keeps you safe: `revert` is for history that other people have. `reset` is for history only you have.
2. `--soft` keeps everything staged. `--mixed`, the default, keeps changes in the working tree but unstaged. `--hard` deletes them.

■ 52.7.8 Plumbing, for looking inside

Command	What it does	Net	Risk
<code>git cat-file -t SHA</code>	the object's type	no	low
<code>git cat-file -p SHA</code>	the object's contents	no	low
<code>git cat-file -s SHA</code>	the object's size	no	low
<code>git hash-object -w file</code>	store a blob, print its SHA	no	low
<code>git rev-parse HEAD</code>	resolve a name to a SHA	no	low
<code>git rev-parse --short HEAD</code>	the short form	no	low
<code>git rev-parse --git-dir</code>	where <code>.git</code> actually is	no	low
<code>git ls-tree HEAD</code>	one tree object listed	no	low
<code>git ls-files -s</code>	the index, with SHAs	no	low

Command	What it does	Net	Risk
git write-tree	index becomes a tree object	no	low
git commit-tree tree -p par	build a commit by hand	no	low
git update-ref refs/heads/x SHA	point a ref somewhere	no	med
git symbolic-ref HEAD	what HEAD points at	no	low
git show-ref	every ref and its SHA	no	low
git for-each-ref	refs, formatted	no	low
git fsck --full	check every object	no	low
git gc	pack and prune	no	low
git count-objects -v	how many objects, how big	no	low
git verify-pack -v file.idx	inside a pack file	no	low
git check-ignore -v file	which rule ignores it	no	low

1. The four object types, seen with `git cat-file -t`: blob is file contents, tree is a directory listing, commit is a snapshot plus metadata, tag is an annotated tag.
2. Git stores content addressed by hash. Git 2.29, released October 2020, added experimental SHA-256 repositories; SHA-1 remains the default in 2026.

52.8 Decision tree: my git operation failed

■ 52.8.1 The five families

1. Every git failure belongs to one of five families. Naming the family is 90 per cent of the fix.

Family	Where it happens	Retry safe
Transport	before or during the bytes	usually
Authentication	who are you	after fixing
Permission or scope	you, but not allowed	after fixing
Server-side rejection	server said no, on purpose	after fixing
Local	never left your machine	yes

git command failed. Read the FIRST error line.

```

|
Does it mention a host, port, TLS,
timeout, disconnect, or reset?
yes -> TRANSPORT. See 52.8.3
no v
Does it say Authentication failed,
Permission denied (publickey),
or ask for a password repeatedly?
yes -> AUTHENTICATION. See 52.8.4
no v
Does it name a scope, a permission,
or say 403 / not authorized?
yes -> SCOPE. See 52.8.5
no v
Does it say rejected, non-fast-forward,
protected branch, hook declined, or
pre-receive?
yes -> SERVER REJECTION. See 52.8.6
no -> LOCAL. See 52.8.7

```


■ 52.8.2 First, always: did it actually happen

1. Before diagnosing anything, find out the current true state.
2. Run `git ls-remote origin`. That is a live query. It is the only reliable answer.
3. Compare with `git rev-parse HEAD` locally.
4. Do not trust `git rev-parse origin/main` or `git status` after a failure. Those read remote-tracking refs, which are a cached memory of your last successful fetch.
5. This step is not optional. Half of all git panic is caused by acting on a stale local belief about the remote.

■ 52.8.3 Transport failures

1. Symptoms: Could not resolve host, Failed to connect, Connection timed out, Connection reset by peer, `SSL_ERROR_SYSCALL`, RPC failed, early EOF, unexpected disconnect while reading sideband packet.
2. Diagnostics, in order:

```
git ls-remote origin          # does anything work at all
ssh -T git@github.com        # SSH transport, port 22
nc -vz github.com 22         # is port 22 even reachable
nc -vz github.com 443        # is port 443 reachable
curl -sS -o /dev/null -w '%{http_code}\n' \
    https://github.com
GIT_CURL_VERBOSE=1 git fetch  # verbose HTTPS
GIT_TRACE=1 GIT_TRACE_PACKET=1 git fetch
```

3. Retry: usually safe for fetch, clone and pull, because they only read.
4. Retry for push: safe only after you have re-queried with `git ls-remote` and know whether the previous attempt landed.
5. Common fixes. If port 22 is blocked but 443 is not, use SSH over 443 by putting this in `~/.ssh/config`:

```
Host github.com
  Hostname ssh.github.com
  Port 443
  User git
```

6. If large pushes fail but small ones work, raise the buffer: `git config --global http.postBuffer 524288000`. This is a workaround, not a fix, and it helps only with HTTP transport.
7. If HTTPS fails and SSH works, or the reverse, that is not a git problem at all. It is two different ports to two different endpoints, and one path is broken. The reader saw exactly this in the same session.

■ 52.8.4 Worked branch: sideband packet disconnect

1. The exact error, seen twice in the reader's session: `send-pack: unexpected disconnect while reading sideband packet`.
2. What it means, literally. Git pushed the pack of objects, then waited on the sideband channel for the server's progress and status messages, and the connection died while it was reading them.
3. Now the important part. The disconnect happened while reading the **response**. It says nothing about whether the **write** was applied.
4. Three outcomes are all consistent with that one message. The server never received the pack. The server received it and rejected it. The server received it, applied it, and the acknowledgement was lost on the way back.

5. This is the same problem as a POST over HTTP that times out. The request is not idempotent, so the client cannot safely assume anything.
6. The only correct next step is to ask the server what it now believes:

```
git ls-remote origin
git ls-remote origin refs/heads/main
git rev-parse HEAD
```

7. If the remote SHA equals your local SHA, the push landed. Do nothing.
8. If it does not, push again. A repeated push of the same commits is harmless, because git will only send objects the server lacks.
9. In the reader's case, both times the operation had **not** been applied. That was established by re-querying, not by reading the error.
10. Retry safety: safe to retry, but only after checking. Never force-push as a response to this error. Force is how a lost acknowledgement turns into lost commits.

■ 52.8.5 Authentication and scope failures

1. Authentication symptoms: Permission denied (publickey), Authentication failed, repeated password prompts, Invalid username or password.


```
ssh -vT git@github.com          # verbose SSH handshake
ssh-add -l                      # keys the agent holds
ssh-add ~/.ssh/id_ed25519       # load a key
git config --get remote.origin.url
git credential-osxkeychain erase # macOS, then type host lines
```
2. GitHub stopped accepting account passwords for git over HTTPS on 13 August 2021. If something is prompting you for a password, it wants a personal access token, not your password.
3. Scope symptoms are different and more specific. The push authenticates fine and is then refused with a named reason.
4. Worked branch, from the reader's session. A push was refused because the token lacked the workflow scope, and the push modified a file under `.github/workflows/`.
5. That is not a bug. It is a deliberate rule: a token that can push code should not be able to change what runs automatically in CI, because CI has its own secrets and its own privileges.
6. The diagnosis is the error text plus one question: does this push touch a workflow file? Check with `git diff --stat origin/main...HEAD` and look for `.github/workflows/`.
7. The fix is to add the workflow scope to the token, or to move the workflow change into a separate push made with a token that has it.
8. Retry safety: retrying with the same token will fail identically, every time. Nothing was applied, because the server refused before applying.
9. This is a clean example of the difference between 401 and 403 thinking. You were identified successfully. You were then found to lack a permission.
10. The general rule for tokens: check three things, in this order. Is it expired. Does it cover this repository. Does it have the scope this specific action needs.

■ 52.8.6 Server-side rejections

1. Symptoms: ! [rejected], non-fast-forward, Updates were rejected because the remote contains work that you do not have locally, protected branch, pre-receive hook declined, GH006.

2. A non-fast-forward rejection means somebody pushed since your last fetch. The server is protecting their work.

```
git fetch origin
git log --oneline HEAD..origin/main    # what they added
git log --oneline origin/main..HEAD    # what you added
git rebase origin/main                 # or: git merge origin/main
git push
```

3. A protected-branch rejection means the repository's rules forbid a direct push. Open a pull request instead. This is a setting, not an error.
4. A pre-receive hook declined means the server ran a script that said no. The reason is usually in the lines just after. Common causes are commit message format rules, file size limits, and secret scanning.
5. GitHub blocks any single file over 100 MiB, and warns above 50 MiB. That limit has been in place since 2015 and is a service policy, not a git limit. Use Git Large File Storage for big files.
6. Retry safety: retrying unchanged always fails again. Nothing was applied. Fix the cause first.

■ 52.8.7 Local failures

1. Symptoms: not a git repository, pathspec did not match any file, your local changes would be overwritten, You have unmerged paths, index.lock exists, detached HEAD.

```
git rev-parse --git-dir      # am I in a repo at all
git status                  # the honest current state
git diff --name-only --diff-filter=U # conflicted files
rm -f .git/index.lock       # only if no git is running
git fsck --full              # object integrity
git switch -                 # leave detached HEAD safely
```

2. index.lock exists almost always means a previous git process died or an editor's git integration is running. Check with `ps aux | grep git` before deleting the lock file.
3. detached HEAD is not an error. It means HEAD points at a commit rather than a branch. Commits made there are reachable only from the reflog, so create a branch before you go further: `git switch -c rescue`.
4. Retry safety: always safe. Nothing left your machine.
5. Finally, one distinction that matters and that the reader's session showed clearly. Git the tool and GitHub the service are different things. During the outage plain git kept working locally, and the GitHub web API did not. An error from gh or from the API is not a git error.

■ 52.8.8 Verifying CI properly

1. A green tick next to a branch name means "some run for something on this branch passed". It does not say which commit.
2. The strong check is by run identifier against the exact head SHA:

```
git rev-parse HEAD
gh run list --branch main --limit 5
gh run view <run-id> --json headSha,status,conclusion
```

3. Compare the headSha field with your local SHA character by character. If they differ, the tick is about a different commit.
4. Merge state, seen through the API, uses values such as CLEAN, BLOCKED and UNKNOWN. UNKNOWN usually means the server has not finished computing the merge yet; re-query rather than assuming a failure.

52.9 Shell and system cheat sheet

■ 52.9.1 Files and directories

Command	What it does
<code>pwd</code>	print the current directory
<code>cd -</code>	go back to the previous one
<code>ls -lah</code>	long, all, human sizes
<code>ls -lt</code>	newest first
<code>ls -ls</code>	biggest first
<code>mkdir -p a/b/c</code>	make the whole path
<code>cp -r src dst</code>	copy a directory
<code>cp -a src dst</code>	copy, keep all attributes
<code>mv old new</code>	move or rename
<code>rm -i file</code>	delete, asking first
<code>rm -rf dir</code>	delete a tree, no questions
<code>ln -s target link</code>	make a symbolic link
<code>touch file</code>	create empty, or update time
<code>stat file</code>	size, times, permissions
<code>file thing</code>	guess the format
<code>du -sh dir</code>	total size of a directory
<code>du -sh * \ sort -h</code>	biggest items first
<code>df -h</code>	free space per filesystem
<code>basename path</code>	last part of a path
<code>dirname path</code>	everything but the last part

■ 52.9.2 Looking inside files

Command	What it does
<code>cat file</code>	print the whole file
<code>less file</code>	page through it, q to quit
<code>head -n 20 file</code>	first 20 lines
<code>tail -n 20 file</code>	last 20 lines
<code>tail -f log</code>	follow as it grows
<code>wc -l file</code>	count lines
<code>sort file</code>	sort lines
<code>sort -u</code>	sort and remove duplicates
<code>uniq -c</code>	count runs, needs sort first
<code>cut -d, -f2 file</code>	second comma field
<code>tr 'a-z' 'A-Z'</code>	change characters
<code>sed 's/old/new/g'</code>	replace text
<code>awk '{print \$3}'</code>	print the third field
<code>grep -n pattern file</code>	matching lines, numbered
<code>grep -ri pattern dir</code>	recursive, ignore case
<code>grep -v pattern</code>	lines that do not match
<code>grep -c pattern</code>	just the count
<code>diff -u a b</code>	unified difference
<code>xxd file \ head</code>	hex dump
<code>jq '.field' file.json</code>	read JSON properly

■ 52.9.3 Finding things

```
find . -name '*.log'           # by name
find . -type f -size +100M    # files over 100 MB
find . -mtime -1             # changed in the last day
find . -name '*.tmp' -delete  # find and delete
find . -type f -exec grep -l x {} + # search inside
which python3                 # which one runs
type -a python3               # every match, and aliases
command -v python3            # portable form
locate filename               # from an index, if built
```

■ 52.9.4 Processes and resources

Command	What it does
ps aux	every process
ps aux grep name	find one
top	live view
htop	nicer live view, install it
kill PID	ask politely, signal 15
kill -9 PID	force, signal 9
pkill -f pattern	kill by command line
jobs	this shell's background jobs
bg and fg	move a job
nohup cmd &	keep running after logout
time cmd	how long it took
uptime	load averages
free -h	memory, Linux only
vm_stat	memory, macOS
sysctl -n hw.ncpu	core count, macOS
nproc	core count, Linux
uname -a	kernel and machine
sw_vers	macOS version
lsb_release -a	Linux distribution

1. Signal 15, SIGTERM, asks a program to shut down cleanly. Signal 9, SIGKILL, cannot be caught or ignored and gives the program no chance to save. Try 15 first, always.

■ 52.9.5 Archives, permissions and transfer

```
tar -czf out.tar.gz dir/      # create, gzip
tar -xzf in.tar.gz            # extract
tar -tzf in.tar.gz            # list without extracting
zip -r out.zip dir/           # create a zip
unzip in.zip                   # extract a zip
gzip file                      # replaces file with file.gz
gunzip file.gz                 # and back
chmod 644 file                 # rw for owner, r for others
chmod 755 script               # add execute for everyone
chmod +x script                # just add execute
chown user:group file          # change ownership
umask                           # default permission mask
rsync -avz src/ host:dst/      # copy, resume, compress
scp file host:/path            # simple copy over SSH
ssh -i key user@host           # connect with a key
```

1. Permission digits are octal. Read is 4, write is 2, execute is 1. Add them. The three digits are owner, group, others. So 755 is read-write-execute for the owner and read-execute for everyone else.

■ 52.9.6 Keyboard shortcuts in the shell

Keys	What it does
Ctrl-C	interrupt the running command
Ctrl-D	end of input, or log out
Ctrl-Z	suspend, resume with fg
Ctrl-L	clear the screen
Ctrl-A	jump to start of line
Ctrl-E	jump to end of line
Ctrl-U	delete to start of line
Ctrl-K	delete to end of line
Ctrl-W	delete the word before
Ctrl-Y	paste what you just deleted
Alt-B, Alt-F	back or forward one word
Ctrl-R	search command history
Ctrl-G	leave the history search
Up, Down	previous, next command
Tab	complete a name
Tab Tab	show all completions
!!	the previous command
!\$	last argument of it
sudo !!	rerun the last one as root

1. On macOS Terminal, Alt is the Option key, and you may need to enable “Use Option as Meta key” in the settings before Alt-B and Alt-F work.

■ 52.9.7 The shell expansion order

1. The shell rewrites your command line in a fixed order before running anything. Knowing the order explains almost every strange quoting bug.

1. brace expansion	{a,b}	->	a b
2. tilde expansion	~	->	/Users/you
3. parameter expansion	\$VAR	->	its value
4. arithmetic expansion	\$((2+2))	->	4
5. command substitution	\$(cmd)	->	its output
6. word splitting			on spaces, tabs, newlines
7. pathname expansion	*.txt	->	matching files
8. quote removal	"x"	->	x

2. The two consequences that bite people.
3. Word splitting happens **after** variable expansion. So an unquoted \$FILE holding my report.txt becomes two arguments. Quoting it stops that.
4. Pathname expansion happens after variable expansion too, so a variable containing * will expand to filenames unless you quote it.

■ 52.9.8 The safety rules

1. **Quote every variable.** Write "\$FILE", not \$FILE. Write "\$@", not \$@. This one habit prevents most shell bugs.
2. **Start scripts with strict mode.**

```
#!/usr/bin/env bash
set -euo pipefail
IFS=$'\n\t'
```

3. -e exits on the first failing command. -u treats an unset variable as an error. -o pipefail makes a pipeline fail if any stage fails, not just the last. Setting IFS stops word splitting on plain spaces.
4. **Never pipe an unknown URL into a shell.** curl ... | sh hands a remote server the ability to run anything as you. Download first, read it, then run it.
5. There is a further reason beyond trust: a server can serve different bytes to curl piped into a shell than to curl writing a file, because it can see the download stall as the shell executes. Reading the file you downloaded is the only real protection.
6. **Test destructive commands first.** Use git clean -n, rm -i, rsync --dry-run, and find ... -print before find ... -delete.
7. **Never run rm -rf with a variable that might be empty.** If \$DIR is unset, rm -rf "\$DIR"/ becomes rm -rf /. set -u prevents this.
8. **Use sudo for one command, not for a session.** A root shell left open is a mistake waiting for a typing error.
9. **Prefer absolute paths in scripts and cron jobs.** The working directory and the PATH in an automated context are not the ones you have interactively.

52.10 File format identification by magic number

1. A magic number is a fixed byte sequence at a known offset that identifies a file's format. The extension is a hint; the magic number is the fact.
2. Check with file thing for a guess, or xxd file | head -2 for the bytes.

Format	Hex bytes at start	Extension
PNG image	89 50 4E 47 0D 0A 1A 0A	.png
JPEG image	FF D8 FF	.jpg
GIF image	47 49 46 38	.gif
BMP image	42 4D	.bmp
TIFF, little-endian	49 49 2A 00	.tif
TIFF, big-endian	4D 4D 00 2A	.tif
WebP	52 49 46 46 then WEBP	.webp
ICO icon	00 00 01 00	.ico
PSD, Photoshop	38 42 50 53	.psd
PDF document	25 50 44 46 2D	.pdf
PostScript	25 21	.ps
RTF document	7B 5C 72 74 66	.rtf
ZIP, and docx, xlsx, jar	50 4B 03 04	.zip
GZIP	1F 8B	.gz
BZIP2	42 5A 68	.bz2
XZ	FD 37 7A 58 5A 00	.xz
Zstandard	28 B5 2F FD	.zst

Format	Hex bytes at start	Extension
7-Zip	37 7A BC AF 27 1C	.7z
RAR version 5	52 61 72 21 1A 07 01	.rar
TAR	75 73 74 61 72 at byte 257	.tar
Old Office, OLE2	D0 CF 11 E0 A1 B1 1A E1	.doc
MP3 with ID3 tag	49 44 33	.mp3
FLAC audio	66 4C 61 43	.flac
OGG container	4F 67 67 53	.ogg
WAV audio	52 49 46 46 then WAVE	.wav
MP4, HEIC, AVIF	66 74 79 70 at byte 4	.mp4
Matroska, WebM	1A 45 DF A3	.mkv
ELF executable	7F 45 4C 46	none
Mach-O 64-bit	CF FA ED FE	none
Windows PE, EXE	4D 5A	.exe
Java class file	CA FE BA BE	.class
WebAssembly	00 61 73 6D	.wasm
SQLite 3 database	53 51 4C 69 74 65 20 66	.sqlite
Git pack file	50 41 43 4B	.pack
Git index file	44 49 52 43	none
Debian package	21 3C 61 72 63 68 3E	.deb
RPM package	ED AB EE DB	.rpm
Shell script	23 21	.sh
PEM certificate	2D 2D 2D 2D 2D 42 45 47	.pem
ISO 9660 image	43 44 30 30 31 at 0x8001	.iso
WOFF web font	77 4F 46 46	.woff
OpenType font	4F 54 54 4F	.otf

- Two entries in that table are the same bytes, and this is a real trap. CA FE BA BE is both a Java class file and an old Mach-O universal binary header. The disambiguation is what follows.
- 50 4B 03 04 covers a very large family. Modern Office files, Java jar files, EPUB books, Android APK files and OpenDocument files are all zip archives with different contents inside.
- Two other useful facts. MZ at the start of a Windows executable stands for Mark Zbikowski, one of the original MS-DOS developers, and has been there since 1983. 7F 45 4C 46 is a non-printable byte followed by the letters ELF, deliberately chosen so text tools do not mangle it.

52.11 Complexity and performance reference

■ 52.11.1 Big-O and what fits in one second

- Big-O describes how the work grows as the input grows. It hides constant factors on purpose.
- The table assumes a rough budget of 100 million simple operations per second, which is a fair figure for interpreted-to-compiled code across the range. Compiled C can do ten times more; Python often ten times less.

Complexity	Name	Largest n in 1 s
O(1)	constant	any size
O(log n)	logarithmic	effectively unlimited
O(n)	linear	100,000,000
O(n log n)	linearithmic	about 4,500,000

Complexity	Name	Largest n in 1 s
$O(n^2)$	quadratic	about 10,000
$O(n^3)$	cubic	about 464
$O(2^n)$	exponential	about 26
$O(n!)$	factorial	about 11

3. The practical reading. If n is a million, you need $O(n)$ or $O(n \log n)$. If n is ten thousand, $O(n^2)$ is acceptable. If n is twenty, anything works.
4. Constants matter in real code. A cache-friendly $O(n^2)$ can beat a pointer-chasing $O(n \log n)$ for small n . Measure before optimizing.

■ 52.11.2 Data structure operation costs

Structure	Lookup	Insert or delete
Array, by index	$O(1)$	$O(n)$
Array, by value	$O(n)$	$O(n)$
Dynamic array, append	$O(1)$	$O(1)$ amortized
Linked list	$O(n)$	$O(1)$ at a known node
Hash table	$O(1)$ average	$O(1)$ average
Hash table, worst case	$O(n)$	$O(n)$
Balanced tree	$O(\log n)$	$O(\log n)$
Binary heap, minimum	$O(1)$	$O(\log n)$
Trie	$O(k)$, k is key length	$O(k)$
B-tree	$O(\log n)$	$O(\log n)$
Skip list	$O(\log n)$ expected	$O(\log n)$ expected
Bloom filter	$O(k)$ hashes	insert only

1. A hash table is $O(1)$ on average and $O(n)$ in the worst case, when every key lands in the same bucket. Modern languages randomize the hash seed at startup to stop attackers forcing that deliberately.
2. A B-tree is the shape used by databases and filesystems because its nodes are sized to a disk block, so one comparison level costs one read.
3. A Bloom filter can say “definitely not present” or “probably present”. It never gives a false negative and cannot delete.

■ 52.11.3 Sorting algorithms

Algorithm	Average	Worst	Note
Quicksort	$O(n \log n)$	$O(n^2)$	in place, unstable
Mergesort	$O(n \log n)$	$O(n \log n)$	needs $O(n)$ space, stable
Heapsort	$O(n \log n)$	$O(n \log n)$	in place, unstable
Timsort	$O(n \log n)$	$O(n \log n)$	stable, fast on runs
Introsort	$O(n \log n)$	$O(n \log n)$	quicksort with fallback
Insertion sort	$O(n^2)$	$O(n^2)$	$O(n)$ if nearly sorted
Selection sort	$O(n^2)$	$O(n^2)$	fewest swaps
Bubble sort	$O(n^2)$	$O(n^2)$	teaching only
Counting sort	$O(n + k)$	$O(n + k)$	small integer keys
Radix sort	$O(n k)$	$O(n k)$	not comparison based

1. **Stable** means two records with equal keys keep their original order. That matters when you sort by one column and then another.
2. Timsort was written by Tim Peters in 2002 for Python. Java adopted it for objects in Java 7, released 2011. It finds runs that are already sorted and merges them, which is why it is $O(n)$ on nearly sorted data.
3. Introsort, by David Musser in 1997, is what most C++ standard libraries use for `std::sort`. It runs quicksort and switches to heapsort if the recursion gets too deep, which removes the $O(n^2)$ worst case.
4. No comparison sort can beat $O(n \log n)$ in the worst case. That is a proved lower bound, not an engineering limit. Counting and radix sort beat it only because they do not compare.

■ 52.11.4 The cost of common operations

Operation	Rough cost
Integer add or subtract	under 1 ns
Integer multiply	1 to 3 ns
Integer divide	6 to 13 ns
Float add or multiply	1 to 2 ns
Float divide	4 to 5 ns
Square root	about 5 ns
Function call	1 to 2 ns
Virtual or indirect call	2 to 5 ns
Branch, predicted correctly	under 1 ns
Branch, mispredicted	3 to 5 ns
Uncontended mutex	about 17 ns
Contended atomic increment	50 to 100 ns
Hash a short string	20 to 50 ns
Small malloc and free	50 to 100 ns
A simple system call	100 to 500 ns
Thread context switch	1 to 5 us
Create a thread	10 to 30 us
Fork and exec a process	0.5 to 1 ms
One Python bytecode step	30 to 50 ns
One Python function call	about 100 ns

1. System calls got more expensive in 2018, when the Meltdown and Spectre mitigations forced page table isolation. Depending on the machine and the mitigation set, a simple system call can cost several times what it did before.
2. The gap between a Python operation and a C operation is roughly fifty to a hundred times. That is why numerical Python work is done inside NumPy, where the loop lives in C.

52.12 Cloud and infrastructure reference

■ 52.12.1 Service equivalence

Category	AWS	Azure	Google
Virtual machines	EC2	Virtual Machines	Compute Engine
Object storage	S3	Blob Storage	Cloud Storage
Block storage	EBS	Managed Disks	Persistent Disk

Category	AWS	Azure	Google
Shared file storage	EFS	Azure Files	Filestore
Archive storage	S3 Glacier	Archive Storage	Archive
Managed relational	RDS	Azure SQL Database	Cloud SQL
Scale-out relational	Aurora	SQL Hyperscale	Spanner
Managed NoSQL	DynamoDB	Cosmos DB	Firestore
Wide-column store	Keyspaces	Cosmos DB	Bigtable
Data warehouse	Redshift	Synapse, Fabric	BigQuery
In-memory cache	ElastiCache	Azure Cache	Memorystore
Serverless functions	Lambda	Functions	Cloud Run functions
Containers, managed	ECS, Fargate	Container Apps	Cloud Run
Kubernetes	EKS	AKS	GKE
Container registry	ECR	Container Registry	Artifact Registry
Load balancing	ELB, ALB, NLB	Load Balancer	Cloud Load Balancing
Content delivery	CloudFront	Front Door	Cloud CDN
DNS	Route 53	Azure DNS	Cloud DNS
Private network	VPC	Virtual Network	VPC
Identity	IAM	Entra ID	IAM
Secrets	Secrets Manager	Key Vault	Secret Manager
Message queue	SQS	Service Bus	Pub/Sub
Publish and subscribe	SNS	Event Grid	Pub/Sub
Event streaming	Kinesis	Event Hubs	Pub/Sub
Managed Kafka	MSK	Event Hubs	Managed Kafka
Workflow orchestration	Step Functions	Logic Apps	Workflows
Monitoring and logs	CloudWatch	Azure Monitor	Cloud Monitoring
Infrastructure as code	CloudFormation	Bicep, ARM	Deployment Manager
Machine learning	SageMaker	Azure ML	Vertex AI
Hosted large models	Bedrock	Azure AI Foundry	Vertex AI

- Names change. Microsoft renamed Azure Active Directory to Microsoft Entra ID in 2023, and has been folding Synapse into Microsoft Fabric since 2023. Google renamed Cloud Functions to Cloud Run functions in 2024.
- The equivalences are approximate. Spanner and Aurora are not the same thing; both solve “a relational database bigger than one machine” and solve it differently.

■ 52.12.2 The service model responsibility split

Model	You manage	Provider manages
On-premises	everything	nothing
IaaS	OS upward	hardware, network, power
PaaS	code and data	OS, runtime, scaling
Serverless	function and data	everything else
SaaS	your data and users	the whole application

- One rule survives every model: **you are always responsible for your own data and your own access control**. No provider takes that on.
- The shared responsibility model is the standard phrase. AWS states it as “security of the cloud” being theirs and “security in the cloud” being yours.

3. The most common real breach in cloud is not a provider failure. It is a storage bucket left open to the public, which is entirely on the customer side of the line.

■ 52.12.3 Instance families and what they are for

Family letter	Optimized for	Use it for
t	burstable, cheap	dev, small sites
m	balanced	general servers
c	CPU per unit cost	encoding, simulation
r	memory	caches, in-memory DBs
x, z	very large memory	SAP, huge datasets
i, d	local NVMe storage	databases, search
p	GPU for training	model training
g	GPU for inference	inference, graphics
inf, trn	custom AI silicon	inference, training

1. The letters above are the AWS convention. Azure uses series letters such as D for general purpose, F for compute, E and M for memory, and N for GPU. Google uses names such as N2, C3, M3 and A3.
2. A modern name such as m7g.2xlarge reads as: family m, generation 7, Graviton Arm processor, size 2xlarge. The suffix g for Arm is an AWS convention.
3. Size doubling is a convention too. Each step up in size usually doubles vCPU and memory and doubles the price, so cost per unit of work is roughly flat. That is why splitting across smaller instances rarely saves money by itself.

■ 52.12.4 Cost dimensions to check on any bill

1. Compute time, per hour or per second, per instance.
2. Storage, per gigabyte-month, separately for each class.
3. Data transfer out to the internet, per gigabyte. This is the item that surprises people. Ingress is usually free; egress is not.
4. Data transfer between availability zones and between regions, per gigabyte, in both directions.
5. Request counts. Object storage charges per PUT and per GET, and a workload with many tiny objects can cost more in requests than in storage.
6. Provisioned throughput and IOPS on disks and databases, whether or not you use them.
7. Load balancer hours plus a per-capacity-unit charge.
8. NAT gateway hours plus a per-gigabyte processing charge. This is a common source of unexplained cost in private networks.
9. Public IPv4 addresses. AWS began charging for every public IPv4 address on 1 February 2024, at 0.005 US dollars per hour, which is about 3.60 dollars a month each.
10. Snapshots and backups, which keep growing and which nobody deletes.
11. Log ingestion and log retention, charged separately.
12. Idle waste: unattached volumes, stopped instances still holding disks, old snapshots, forgotten test environments.
13. Support plan, usually a percentage of the total spend.
14. Commitment discounts. Reserved instances and savings plans trade flexibility for a lower rate. Spot instances are much cheaper and can be taken away with two minutes of notice.

52.13 Security reference

■ 52.13.1 Hash functions, as of 2026

Algorithm	Verdict	Note
MD5	broken	collisions since 2004
SHA-1	broken	collision proved 2017
SHA-256	use	the default choice
SHA-384, SHA-512	use	wider, often faster on 64-bit
SHA-3 family	use	different internal design
BLAKE2, BLAKE3	use	very fast, not FIPS
CRC32	not a hash	error detection only

1. MD5 was broken by Xiaoyun Wang and colleagues in 2004. SHA-1 was broken in practice on 23 February 2017, when Google and CWI Amsterdam published two different PDF files with the same SHA-1 digest, an attack they named SHAttered. A chosen-prefix collision followed in January 2020.
2. Neither is safe for signatures or integrity against an attacker. Both are still acceptable as non-security checksums, and git still uses SHA-1 with an added collision-detection check.

■ 52.13.2 Ciphers and key exchange, as of 2026

Purpose	Use	Avoid
Symmetric encryption	AES-256-GCM	DES, 3DES, RC4
Symmetric, no AES hardware	ChaCha20-Poly1305	AES in ECB mode
Mode of operation	GCM, or GCM-SIV	CBC without a MAC
Key exchange	X25519, ECDHE	static RSA, small DH
Signature	Ed25519, ECDSA P-256	DSA, RSA-1024
RSA, if you must	3072 bits or more	1024, 2048 for new keys
SSH host and user keys	Ed25519	DSA, RSA-1024

1. NIST disallowed three-key Triple DES for new encryption after 31 December 2023. RC4 was prohibited in TLS by RFC 7465 in February 2015.
2. Post-quantum standards arrived on 13 August 2024, when NIST published FIPS 203 for ML-KEM, derived from Kyber, FIPS 204 for ML-DSA, derived from Dilithium, and FIPS 205 for SLH-DSA, derived from SPHINCS+.
3. Deployment is hybrid: browsers and servers have been negotiating X25519 combined with ML-KEM-768 since 2024, so a future quantum computer cannot decrypt traffic captured today. This is established practice now, not a research claim.

■ 52.13.3 Password storage parameters

1. Never store a password. Store a slow, salted hash of it. These are the OWASP recommended minimums.

Algorithm	Parameters
Argon2id, preferred	m=19456 KiB, t=2, p=1
Argon2id, alternative	m=47104 KiB, t=1, p=1
Argon2id, alternative	m=12288 KiB, t=3, p=1

Algorithm	Parameters
scrypt	$N=2^{17}$, $r=8$, $p=1$
scrypt, alternative	$N=2^{16}$, $r=8$, $p=2$
bcrypt	work factor 10 or more
PBKDF2-HMAC-SHA256	600,000 iterations
PBKDF2-HMAC-SHA512	220,000 iterations

2. m is memory in kibibytes, t is the number of passes, p is parallelism. Memory cost is the point: it is what makes custom hardware attacks expensive.
3. bcrypt silently ignores everything past 72 bytes of input. Enforce a maximum length or pre-hash. The work factor is a power of two, so 12 is four times the work of 10.
4. Always use a random salt per password, at least 16 bytes. A pepper, a secret stored outside the database, is optional and adds real value.
5. Plain SHA-256 of a password is not acceptable, with or without a salt. It is far too fast: modern hardware computes billions per second.

■ 52.13.4 TLS configuration essentials

1. Enable TLS 1.3 and TLS 1.2. Disable everything older. TLS 1.0 and 1.1 were formally deprecated by RFC 8996 in March 2021.
2. TLS 1.3, standardized in RFC 8446 in August 2018, has only five cipher suites and all of them are safe. The three you will see are TLS_AES_128_GCM_SHA256, TLS_AES_256_GCM_SHA384 and TLS_CHACHA20_POLY1305_SHA256.
3. For TLS 1.2, allow only suites beginning ECDHE_ with GCM or CHACHA20. That gives forward secrecy and authenticated encryption.
4. Turn on OCSP stapling so clients do not have to contact the certificate authority themselves.
5. Send the HTTP Strict Transport Security header with a long lifetime: Strict-Transport-Security: max-age=31536000; includeSubDomains.
6. Disable TLS compression and client-initiated renegotiation. The CRIME attack of 2012 used compression to recover session cookies.
7. Publish a CAA DNS record naming which certificate authorities may issue for your domain.
8. Watch expiry. The CA/Browser Forum agreed in April 2025 to reduce the maximum certificate lifetime in stages, reaching 47 days by March 2029. Automated renewal stops being optional.

■ 52.13.5 The OWASP Top 10, 2025 release

Code	Category
A01	Broken Access Control
A02	Security Misconfiguration
A03	Software Supply Chain Failures
A04	Cryptographic Failures
A05	Injection
A06	Insecure Design
A07	Authentication Failures
A08	Software or Data Integrity Failures
A09	Security Logging and Alerting Failures
A10	Mishandling of Exceptional Conditions

1. This list replaced the 2021 edition. Two changes are worth knowing. Supply chain failures moved up to third place, reflecting attacks on dependencies and build systems rather than on the application itself. Mishandling of exceptional conditions is new at tenth: error handling that leaks information or fails open.
2. The list is an awareness document, ordered by a mix of measured incidence and survey. It is not a checklist you can pass.

■ 52.13.6 A personal security checklist

1. Use a password manager and a different password for every site. This one change removes the largest category of real-world account loss.
2. Turn on two-factor authentication everywhere it is offered. Prefer a passkey or a hardware security key, then a time-based code app, and use SMS only if nothing else is available. SMS codes can be stolen by taking over the phone number.
3. Turn on full disk encryption. On macOS that is FileVault, in System Settings under Privacy and Security.
4. Let the operating system and browser update themselves.
5. Use a non-administrator account for daily work where you can.
6. Back up on the 3-2-1 rule: three copies, on two kinds of media, one of them off-site. Test a restore at least once; an untested backup is a guess.
7. Lock the screen when you walk away.
8. Review which applications have access to your accounts. On GitHub, check authorized OAuth applications and personal access tokens, and delete the ones you do not recognize.
9. Give every token the smallest scope that works, and an expiry date. The reader's workflow scope refusal is the system working correctly.
10. Check your email addresses against public breach databases once a year.
11. Do not install browser extensions you do not need. An extension can read everything you see on every page.
12. Be suspicious of urgency. Almost every successful phishing message manufactures a deadline.

52.14 AI and machine learning reference

■ 52.14.1 The memory formula

1. For inference, the base memory is simple: **bytes = number of parameters times bytes per parameter**.

Precision	Bytes per parameter	Note
FP32	4	full precision
FP16 or BF16	2	standard for inference
FP8	1	newer hardware only
INT8	1	common quantization
INT4 or NF4	0.5	aggressive quantization

2. Add roughly 20 per cent on top for the runtime, activations and fragmentation.
3. Then add the key-value cache, which grows with the conversation: **KV bytes = 2 times layers times kv-heads times head-dim times sequence length times batch size times bytes per element**.
4. The leading 2 is because there is one key tensor and one value tensor.

- For training with the Adam optimizer in mixed precision, budget roughly 16 to 20 bytes per parameter: 4 for the master weights, 4 for momentum, 4 for variance, 2 for the working copy, 2 for gradients, plus activations.
- So a 7 billion parameter model needs about 14 GB to run at 16-bit and about 112 GB or more to train the same way. That gap is why fine-tuning methods such as LoRA exist.

■ 52.14.2 Model size against hardware

Parameters	4-bit	8-bit	16-bit
1 B	0.5 GB	1 GB	2 GB
3 B	1.5 GB	3 GB	6 GB
7 B	3.5 GB	7 GB	14 GB
8 B	4 GB	8 GB	16 GB
13 B	6.5 GB	13 GB	26 GB
32 B	16 GB	32 GB	64 GB
70 B	35 GB	70 GB	140 GB
405 B	203 GB	405 GB	810 GB

- Practical readings for 2026 hardware. A laptop with 16 GB of unified memory runs a 7B or 8B model comfortably at 4-bit. A 24 GB consumer GPU runs 13B at 8-bit or 32B at 4-bit. A 70B model at 16-bit needs two 80 GB accelerators. A 405B model needs a multi-node cluster.
- Quantization costs quality, and the cost is not linear. Going from 16-bit to 8-bit is usually almost free. Going to 4-bit is usually acceptable. Going below 4-bit degrades noticeably, though the frontier moves.

■ 52.14.3 Token cost rules of thumb

Rule	Figure
1 token, English	about 4 characters
1 token, English	about 0.75 words
1,000 words English	about 1,300 tokens
1 page of prose	about 500 tokens
Source code	about 3 chars per token
Hindi in Devanagari	3 to 6 times English
A 300-page book	about 170,000 tokens

- Non-Latin scripts cost more because the tokenizer was trained mostly on English text, so it has fewer merged pieces for them. This is a measured property of specific tokenizers, not a law.
- Cost scales with tokens in plus tokens out, and output tokens are usually priced several times higher than input tokens.
- Attention cost grows with the square of the sequence length in the classical transformer, so doubling the context roughly quadruples that part of the compute.

■ 52.14.4 Common hyperparameters

Name	What it controls
Learning rate	step size when updating weights
Batch size	examples per update

Name	What it controls
Epochs	passes over the whole dataset
Warmup steps	ramp-up before full learning rate
Weight decay	penalty that keeps weights small
Dropout	fraction of units randomly ignored
Gradient accumulation	fake a bigger batch on small memory
LoRA rank	size of the small trained adapter
Temperature	randomness when sampling output
Top-p, nucleus	keep tokens summing to probability p
Top-k	keep only the k most likely tokens
Repetition penalty	discourage repeating tokens
Context length	how many tokens the model can see
Max new tokens	how long the answer may be

1. Temperature 0 means always take the most likely token, which makes output nearly deterministic. Temperature 1 samples from the model's own distribution. Above about 1.2 the output usually becomes incoherent.
2. Learning rate is the hyperparameter that matters most. Too high and training diverges; too low and it never gets anywhere.

■ 52.14.5 How to read a model announcement

1. Separate the three kinds of claim before you read anything else.
2. **Established fact:** parameter count, context length, licence, price per million tokens, whether weights can be downloaded, release date.
3. **Active research:** benchmark scores, reasoning ability, agentic behaviour, long-context quality. These are measured, but the measurements are contested and move quickly.
4. **Marketing claim:** "state of the art", "PhD-level", "reasoning", "understands". These words have no agreed definition.
5. Then ask these ten questions.
6. Which exact benchmark, which version, and which prompting setup. A score with chain-of-thought and 64 samples is not comparable to a single-shot score.
7. Was the benchmark possibly in the training data. Contamination is common and usually undetectable from outside.
8. Is this a base model or an instruction-tuned model. They are not interchangeable.
9. What is the licence. "Open weights" means you can download the numbers. "Open source" properly means you also get the training data and code, which is rare.
10. What is the real context window, and is there an independent measurement of quality at the far end of it. Advertised context and usable context often differ.
11. What is the throughput in tokens per second, and at what batch size.
12. What is the price per million input tokens and per million output tokens.
13. Who ran the evaluation. First-party numbers should be treated as a claim until someone independent repeats them.
14. What is the cut-off date of the training data.
15. What is not mentioned. Announcements omit the benchmarks where the model did badly, and that omission is information.

52.15 Regular expressions

■ 52.15.1 The syntax table

Pattern	Means	Example match
.	any one character	a.c matches abc
^	start of string or line	^Hi
\$	end of string or line	end\$
*	zero or more of the last	ab* matches a
+	one or more of the last	ab+ matches ab
?	zero or one of the last	colou?r
{3}	exactly three	a{3} matches aaa
{2,4}	two to four	a{2,4}
{2,}	two or more	a{2,}
*?, +?	lazy, match as few as possible	<.+?>
[abc]	any one listed character	[aeiou]
[^abc]	any character not listed	[^0-9]
[a-z]	a range	[A-Za-z]
\d	a digit	same as [0-9]
\D	not a digit	
\w	word character	letters, digits, underscore
\W	not a word character	
\s	whitespace	space, tab, newline
\S	not whitespace	
\b	word boundary	\bcat\b
\B	not a word boundary	
\	either side	cat\ dog
()	group and capture	(ab)+
(?:)	group without capturing	(?:ab)+
(?<name>)	named capture group	
\1	what group 1 matched	(\w)\1 finds doubles
(?=)	followed by, lookahead	foo(=bar)
(?!)	not followed by	foo(?!bar)
(?<=)	preceded by, lookbehind	(?<=Rs)\d+
(?<!)	not preceded by	
\.	a literal dot	escapes the dot
(?i)	ignore case from here	

1. Flags are set outside the pattern in most languages: i ignore case, m multi-line so ^ and \$ match at each line, s so . also matches a newline, g find all matches.
2. Two warnings. Nested quantifiers such as (a+)+ can take exponential time on a failing input, which is a real denial-of-service class called catastrophic backtracking. And you cannot parse HTML with a regular expression; use a parser.

■ 52.15.2 Five patterns worth keeping

1. Match an IPv4 address, roughly:

```
| \b(?:\d{1,3}\.){3}\d{1,3}\b
```
2. It finds four groups of one to three digits separated by dots. It will also match 999.999.999.999, so validate the ranges afterwards in code. A regular expression that fully validates IPv4 is long and unreadable, and checking each number is under 256 in code is clearer.
3. Find the start of every log line with a timestamp:

```
| ^\d{4}-\d{2}-\d{2}[ T]\d{2}:\d{2}:\d{2}
```
4. This matches both 2026-08-13 14:05:09 and the ISO 8601 form with a T. Use it with `grep -E` to pick out real log lines and drop stack-trace continuation lines.
5. Match a quoted string, without running past the closing quote:

```
| "(?:[^\\"|\\\.])*"
```
6. The trick is `[^\\"|\\\.]`, which means any character that is not a quote and not a backslash, alternated with `\\.`, which means a backslash followed by anything. That handles escaped quotes inside the string.
7. Find a possible secret in a diff before you commit it:

```
| (?i)(api[_-]?key|secret|token|password)\s*[:=]\s*\S+
```
8. Run it with `git diff --staged | grep -E` before every commit. It produces false positives, which is the correct trade-off for this job.
9. Capture the status code and byte count from a common log format line:

```
| "\s(\d{3})\s(\d+|-)$
```
10. Group 1 is the HTTP status, group 2 is the response size or a dash. Pipe it through `awk` or `sed -E` to build a count by status code.

52.16 Keyboard shortcuts and editor basics

■ 52.16.1 Terminal

Keys	What it does
Cmd-T	new tab, macOS Terminal
Cmd-N	new window, macOS
Cmd-K	clear scrollbar, macOS
Ctrl-Shift-T	new tab, Linux terminals
Ctrl-Shift-C, Ctrl-Shift-V	copy, paste, Linux
Ctrl-A then D	detach from a tmux session
Ctrl-A then C	new tmux window
Ctrl-A then N	next tmux window

1. The tmux prefix is Ctrl-B by default; Ctrl-A is a very common reconfiguration and a convention rather than a default.

■ 52.16.2 Shell line editing

1. The shortcuts in section 52.9.6 are Emacs-style bindings, which is the default in bash and zsh.
2. If you prefer vi bindings, run `set -o vi`. Then Escape puts you in command mode and the vim movement keys work on the command line.
3. `Ctrl-R` then typing searches history backwards. Press `Ctrl-R` again to go further back. `Ctrl-G` cancels.

■ 52.16.3 Enough vim to edit a file and get out

1. vim has modes. That is the whole thing people find confusing. You start in normal mode, where letters are commands, not text.

Key	What it does
i	insert before the cursor
a	insert after the cursor
o	open a new line below
Escape	leave insert mode
h j k l	left, down, up, right
0	start of line
\$	end of line
gg	top of file
G	bottom of file
:42	go to line 42
/text then Enter	search forward
n	next match
x	delete one character
dd	delete the whole line
yy	copy the whole line
p	paste after the cursor
u	undo
Ctrl-R	redo
:w	write the file
:q	quit
:wq	write and quit
:q!	quit and throw away changes

2. The five you actually need, in order: press Escape to be sure you are in normal mode. Press i to type. Press Escape again. Type `:wq` and press Enter to save and leave. Type `:q!` and press Enter to leave without saving.
3. If a git command drops you into an editor you did not expect, that is vim, because it is the default `core.editor` on many systems. Change it with `git config --global core.editor "nano"` if you would rather.
4. nano is the friendly alternative. `Ctrl-O` writes, `Ctrl-X` exits, and it prints the shortcuts along the bottom of the screen.

52.17 What to do when you are stuck

■ 52.17.1 The debugging method

1. **Reproduce it.** A bug you cannot trigger on purpose cannot be fixed, only guessed at. Write down the exact steps.
2. **Read the actual error.** Not the summary. The first line and the deepest line of the stack.
3. **Establish what is true.** Write down three things you have proved and how you proved them. Write down three things you are assuming.
4. **Check the assumptions first.** Most bugs live there. The version you think you are running, the file you think you are editing, the environment variable you think is set.
5. **Bisect.** Cut the problem in half. Does the failure happen with half the input, half the config, half the code. `git bisect` does this over history automatically.
6. **Change one thing at a time.** Two changes at once means you learn nothing from the result.
7. **Look at the boundary.** Most faults sit where two systems meet: your code and a library, your machine and the network, your process and the filesystem.
8. **Ask what changed.** If it worked yesterday, something moved. A deployment, a certificate, a DNS record, a dependency update, a disk filling up.
9. **Explain it out loud** to a person or an object. Saying it in order forces the gap into view. This is called rubber-duck debugging, and it works because it makes you state your assumptions.
10. **Take a break.** After ninety minutes with no progress, stop. Returning with fresh attention beats another hour of the same.
11. **Write down the fix** and why it worked. The same problem will come back to you or to somebody else.

■ 52.17.2 How to read an error message

1. Read the **first** line and the **last** line before anything in between. The first says what went wrong. The last is often where it went wrong.
2. Find the **first line that mentions your own code.** Everything above it is usually library internals.
3. Separate the **error type** from the **message** from the **location**. `FileNotFoundException` is the type. The path is the message. The line number is the location.
4. Look for a number: an exit code, an `errno`, a status code, a signal number. Those are searchable and precise.
5. Common exit codes: 0 success, 1 general error, 2 misuse of a shell builtin, 126 found but not executable, 127 command not found, 130 interrupted by Ctrl-C, 137 killed by signal 9, often the out-of-memory killer.
6. Common `errno` values: 2 `ENOENT` no such file, 13 `EACCES` permission denied, 32 `EPIPE` broken pipe, 98 `EADDRINUSE` address already in use, 110 `ETIMEDOUT` connection timed out, 111 `ECONNREFUSED` connection refused.
7. Note what the error does **not** say. An error that says “connection refused” tells you something answered. An error that says “timed out” tells you nothing answered. Those are different problems.

■ 52.17.3 The minimal reproduction

1. Copy the failing thing into a new, empty directory or file.
2. Delete half of it. Does it still fail. If yes, delete half of what is left. If no, put it back and delete the other half.
3. Repeat until removing anything makes the failure go away.
4. What remains is the reproduction. It is usually five to twenty lines.
5. Remove your credentials, your company's names and your real data. Replace them with `example.com` and the documentation addresses from section 52.4.3.
6. Half the time the bug becomes obvious during this process, and you never need to ask anyone. That is not wasted effort; that is the method working.

■ 52.17.4 Searching effectively

1. Search the **exact error text**, in quotes, with your own paths, names and numbers removed.
2. Add the technology and the version: `postgres 16`, not `postgres`.
3. Add the year if the technology moves fast, or restrict the results to the last year. Answers about a 2019 version of a fast-moving tool are often wrong now.
4. Search the project's own issue tracker before the general web. The people who wrote the code discuss it there.
5. Read the official documentation for the exact version you are running. Not the latest, the one you have.
6. When an answer says "just disable the check", read why the check exists before doing it.
7. Prefer answers that explain the mechanism. An answer that gives you a command with no reason will not help you next time, and may not be safe.

■ 52.17.5 Asking a question that gets answered

1. Give these seven things, in this order, and you will get an answer.
2. **What you are trying to do**, in one sentence, at the level of the goal and not the current attempt.
3. **What you did**, as exact commands or code that someone else could run.
4. **What you expected** to happen.
5. **What actually happened**, pasted verbatim, including the full error, as text and not as a picture.
6. **Your environment**: operating system and version, tool versions, and how it was installed.
7. **What you have already tried** and what each attempt ruled out. This is the single line that most changes the quality of the answers, because it stops people suggesting things you have done.
8. **A minimal reproduction**, from section 52.17.3.
9. Two things not to do. Do not ask "does anyone know about X"; just ask the question. And do not describe the problem you invented while trying to solve the real one, without saying what the real one is; that is the most common reason a thread goes nowhere.
10. If you solve it yourself before anyone replies, post the answer. Somebody with the same problem will find it.

52.98 Common wrong ideas

1. Wrong: a failed ping means the host is down. Right: it means ICMP did not come back. Very many hosts drop ICMP deliberately and serve HTTPS perfectly.
2. Wrong: * * * at the end of a traceroute proves the fault is at that hop. Right: it proves those routers did not send ICMP time-exceeded messages, which is common and often deliberate.
3. Wrong: a timeout and a connection refused are both just failures. Right: refused means something answered and said no; a timeout means nothing answered at all. They point at different causes.

4. Wrong: private addresses in a traceroute mean something is misconfigured. Right: ISPs number their own cores privately as normal practice, which is why the reader saw 172.31, 172.26 and 172.16 addresses.
5. Wrong: `git rev-parse origin/main` tells you where the remote is. Right: it tells you where the remote was at your last fetch. Only `git ls-remote` asks the server.
6. Wrong: a disconnect during a push means the push did not happen. Right: the disconnect happened while reading the reply, so it says nothing about whether the write landed. Re-query and find out.
7. Wrong: 1 GB and 1 GiB are the same. Right: they differ by 7.4 per cent, and the gap grows with every step up the scale.
8. Wrong: MB/s and Mb/s are the same. Right: they differ by a factor of eight. A 100 Mbit/s line delivers about 12 MB/s at best.
9. Wrong: a green tick on a branch means your commit passed CI. Right: it means some run passed for something on that branch. Check the run's head SHA against your own.
10. Wrong: hashing a password with SHA-256 is secure storage. Right: SHA-256 is far too fast. Use Argon2id, scrypt or bcrypt with the parameters in section 52.13.3.

52.99 Chapter summary in 20 lines

1. This chapter is for looking things up, not for reading through.
2. Learn the latency ladder as shape, not as digits: cache in nanoseconds, memory in hundreds of nanoseconds, storage in microseconds, disks and networks in milliseconds.
3. Scaled to human time at one nanosecond per second, a cross-continent round trip is several years and a fifteen-second timeout is several centuries.
4. About one millisecond of round trip per hundred kilometres of fibre is a rule you can do in your head, and it is physics.
5. Learn 2^{10} , 2^{20} and 2^{30} , and the three that cause bugs: 2^{31} , 2^{32} and 2^{53} .
6. Decimal and binary prefixes differ by a percentage that grows with scale: 2.4 per cent at kilo, 7.4 at giga, 20.9 at yotta.
7. One hex digit is four bits and two are one byte. That is the only reason hexadecimal exists in computing.
8. UTF-8 costs one byte for English, two for most European scripts, three for Devanagari and East Asian scripts, four for emoji.
9. The seven OSI layers are the vocabulary; the four TCP/IP layers are what actually runs. Both statements are true together.
10. The private ranges are 10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16, from RFC 1918, and the middle one reaches to 172.31.
11. For every command, ask two questions: what does a pass prove, and what does it not prove. That second question is the whole method.
12. For "I cannot reach a site", work upward: link, DNS, TCP, TLS, HTTP, content. Each pass eliminates everything below it.
13. Refused, unreachable and silent are three different failures with three different causes, and only the third costs you a full timeout.
14. Report a fault to an ISP in protocol terms with a control test, and never accuse anyone; the evidence supports a silent drop and does not name a party.
15. Almost every git command works offline, because the whole history is in the `.git` directory on your machine.

16. Git failures fall into five families: transport, authentication, scope, server-side rejection and local. Naming the family is most of the fix.
17. After any failed push, ask the server what it now believes with `git ls-remote` before doing anything else.
18. Quote your variables, start scripts with `set -euo pipefail`, and never pipe an unknown URL into a shell.
19. A magic number identifies a file; the extension only suggests one.
20. When stuck: reproduce, read the real error, separate what is proved from what is assumed, change one thing at a time, and write down the fix.

Master Timeline and Glossary

53.0 How to use this chapter

1. This is the last chapter, and it is the only one built for looking things up rather than reading straight through.
2. Every other chapter in this book teaches one topic twice: once in plain words, once in the engineer's words. This chapter holds the index to all of them.
3. Because it is a reference, it does not use the six-block shape that the rest of the book uses. It is a timeline, three smaller timelines, a long glossary, and two closing pages.
4. Section 53.1 is the master timeline of computing, from clay tokens around 3000 BCE to the state of things in 2026. Three columns: the year, what happened, and why it mattered.
5. Section 53.2 is a timeline of programming languages, with who made each one and what problem it was built for.
6. Section 53.3 is a timeline of standards and protocols: the written-down agreements that let machines from different makers work together.
7. Section 53.4 is the glossary. It is the main body of this chapter. It is alphabetical, one letter per heading, and it is meant to cover every term used anywhere in the 53 chapters.
8. Every glossary line has exactly three parts, separated by dashes: the term, then the plain meaning in one short sentence, then the technical meaning with the precise definition.
9. Read the plain meaning if you just want to follow a conversation. Read the technical meaning if you have to write it down, argue about it, or build it.
10. Section 53.5 is honest advice on what to do next. Section 53.6 compresses the whole book into exactly 100 numbered lines.
11. A word about dates. Where a thing has several defensible dates, we pick one and say which event it is: invented, first working, published, standardized or shipped. Those can be years apart.
12. Where experts genuinely disagree about a date, we say "about" and give the range. The Antikythera mechanism is the clearest case: published estimates cluster between 150 and 100 BCE.
13. Anything in this chapter dated 2025 or 2026 is recent enough that you should check it again before quoting it. Everything before about 2020 is settled history.
14. Two small conventions used throughout. BCE means before the common era, so the numbers count down. "c." is short for the Latin circa and means "about".

53.1 The master timeline of computing

The table below has one row per event. It is one continuous list in year order. Ancient dates are approximate by nature; modern ones are exact.

Year	Event	Why it mattered
c. 3000 BCE	Sumerian clay tokens	First stored count outside a head
c. 2700 BCE	Sumerian counting board	Ancestor of every abacus
c. 2000 BCE	Babylonian base-60 tables	Reusable answers, written down
c. 1900 BCE	Babylonian place value	Position of a digit carries value
c. 1550 BCE	Rhind papyrus copied	Oldest surviving arithmetic manual
c. 500 BCE	Chinese counting rods	Positional arithmetic with sticks
c. 300 BCE	Euclid writes the Elements	A procedure you can follow blindly
c. 300 BCE	Salamis counting tablet	Oldest surviving counting board
c. 100 BCE	Antikythera mechanism	Geared model of the sky, 30+ gears
c. 100 CE	Roman hand abacus	Calculation becomes portable
c. 499 CE	Aryabhata's Aryabhatiya	Indian decimal place value written
628 CE	Brahmagupta's rules for zero	Zero becomes a number, not a gap
c. 820 CE	Al-Khwarizmi on calculation	Gives us "algorithm" and "algebra"
c. 1200 CE	Chinese suanpan in wide use	Fast decimal arithmetic for trade
1202	Fibonacci's Liber Abaci	Arabic numerals reach Europe
1614	Napier publishes logarithms	Multiplying becomes adding
1617	Napier's bones described	Portable multiplication aid
1622	Oughtred's slide rule	Analogue calculation for 350 years
1623	Schickard's calculating clock	First known geared calculator
1642	Pascal builds the Pascaline	Geared adder with automatic carry
1673	Leibniz's stepped reckoner	First machine to multiply directly
1703	Leibniz publishes on binary	Base 2 arithmetic written down
1725	Bouchon's punched-tape loom	Holes in paper steer a machine
1728	Falcon's punched-card loom	Cards replace fragile tape
1801	Jacquard shows loom in Paris	Pattern separated from mechanism
1804	Jacquard head patented	The program becomes a thing you own
1822	Babbage's difference engine	Machine-made mathematical tables
1837	Cooke-Wheatstone telegraph	Messages faster than horses
1837	Analytical Engine designed	Mill, store, cards: a real computer
1838	Morse and Vail show their code	Letters become dots and dashes
1843	Lovelace publishes Note G	First published algorithm for a machine
1844	Washington to Baltimore line	Telegraph proven over distance

Year	Event	Why it mattered
1847	Boole's Mathematical Analysis	Logic written as algebra
1854	Boole's Laws of Thought	AND, OR and NOT get their rules
1858	First transatlantic cable works	Continents linked, then it failed
1866	Permanent Atlantic cable laid	The first reliable global network
1876	Bell patents the telephone	Sound itself travels on wires
1877	Edison's phonograph	Sound recorded and played back
1878	First telephone exchange	Switching, not point-to-point wires
1887	Hertz produces radio waves	Signals without any wire
1889	Hollerith patent 395,782	Counting by electricity and holes
1890	US census tabulated by machine	First mass data processing job
1896	Tabulating Machine Company	The firm that becomes IBM
1897	Thomson identifies the electron	The carrier of charge is named
1900	Hilbert lists 23 problems	Sets the century's maths agenda
1901	Marconi's transatlantic signal	Radio crosses an ocean
1904	Fleming's thermionic valve	First electronic one-way device
1906	De Forest's audion triode	Electronic amplification is possible
1911	CTR formed by merger	Punched cards become an industry
1924	CTR renamed IBM	Sixty years of dominance begin
1925	Bell Telephone Laboratories	Research lab behind much of this book
1928	The Entscheidungsproblem posed	Can proof itself be mechanized
1931	Godel's incompleteness theorems	Some true things cannot be proved
1936	Turing's On Computable Numbers	Defines what a computer can ever do
1936	Church's lambda calculus	A second, equivalent model of computing
1937	Shannon's relay thesis at MIT	Boolean algebra meets real wires
1937	Stibitz builds the Model K	Binary adder from relays, on a table
1938	Zuse's Z1 in a Berlin flat	Mechanical binary programmable machine
1939	Hewlett-Packard founded	The garage startup pattern begins
1940	Stibitz demonstrates remotely	First computing over a telegraph line
1941	Zuse's Z3 runs	First working program-driven computer
1942	Atanasoff-Berry Computer works	Electronic binary arithmetic, no program
1943	Colossus Mark 1 at Bletchley	First large-scale electronic computer

Year	Event	Why it mattered
1944	Harvard Mark I dedicated	US sequence-controlled calculator
1945	First Draft on the EDVAC	Stored-program design written down
1945	Bush's As We May Think	Imagines linked documents, the memex
1946	ENIAC shown to the press	Electronic speed becomes public news
1947	Transistor at Bell Labs	The end of the vacuum tube begins
1948	Manchester Baby runs a program	First stored program actually executed
1948	Shannon's theory of communication	Information gets a unit and a limit
1948	Wiener publishes Cybernetics	Feedback and control as one subject
1949	EDSAC enters regular service	Subroutines and a real user service
1950	Turing's imitation game paper	The question of machine thinking framed
1951	Ferranti Mark 1 delivered	First commercially sold computer
1951	UNIVAC I accepted by the census	Computing enters US government work
1952	Hopper's A-0 translator	The idea of a compiler appears
1952	UNIVAC predicts the election	Computers enter everyday speech
1953	IBM 701 shipped	IBM enters electronic computing
1953	Core memory in Whirlwind	Fast, reliable random-access memory
1954	Silicon transistor at TI	Silicon beats germanium on heat
1956	IBM 350 RAMAC disk drive	Random access to five megabytes
1956	Dartmouth workshop names AI	The field gets its name
1957	Fortran compiler delivered	Programming above the machine
1957	Sputnik launched	Panic that funds ARPA and the space race
1957	Eight leave Shockley	Fairchild, and then Silicon Valley
1958	Kilby demonstrates a circuit on one chip	Components without separate wires
1958	ARPA founded	The agency that funds the internet
1959	Noyce's planar integrated circuit	Chips can be made in quantity
1959	COBOL specified by CODASYL	Business data processing standardized
1959	DEC PDP-1 announced	Interactive computing begins
1960	ALGOL 60 report published	Block structure and formal grammar

Year	Event	Why it mattered
1960	Baran begins work at RAND	Networks that survive damage
1961	CTSS time-sharing demonstrated	Many users share one machine
1962	Licklider heads ARPA's IPTO	Funding aimed at networked computing
1962	Spacewar on the PDP-1	The first widely copied video game
1963	ASCII published	One agreed number per letter
1963	Sutherland's Sketchpad	Interactive graphics and constraints
1964	IBM System/360 announced	One family, one instruction set
1964	BASIC written at Dartmouth	Beginners can write programs
1964	Engelbart's first mouse	Pointing becomes an input method
1965	Moore's law article published	The industry gets a roadmap
1965	Davies coins the word packet	Data is chopped into labelled pieces
1965	DEC PDP-8 at \$18,500	The minicomputer era opens
1966	Taylor funds the ARPANET	The project is actually paid for
1968	Intel founded	The chip giant begins
1968	Engelbart's demo in San Francisco	Mouse, windows, hypertext, video call
1969	RFC 1 published	The open standards habit begins
1969	ARPANET's first message, 29 Oct	Two computers talk across a country
1969	Unix begun at Bell Labs	The operating system family that won
1969	AMD founded	A second source for chips
1970	Xerox PARC founded	Where the modern desktop was invented
1970	Codd's relational model paper	Data as tables you can query
1970	Intel 1103 DRAM shipped	Semiconductor memory beats magnetic core
1971	Intel 4004 announced	A whole processor on one chip
1971	Tomlinson sends network email	The @ sign is chosen as a separator
1971	IBM ships the floppy disk	Removable storage for everyone
1971	Project Gutenberg begins	Text made free and machine-readable
1972	C language at Bell Labs	Portable systems programming
1972	Atari's Pong released	Video games become a business
1972	ARPANET demonstrated publicly	Doubters are convinced
1973	Xerox Alto working	Screen, mouse, windows, network, all at once
1973	Metcalf's Ethernet memo	Local networks over one shared cable

Year	Event	Why it mattered
1973	Norway and UK join ARPANET	The network becomes international
1974	Cerf and Kahn's protocol paper	Networks of networks: the internet idea
1974	Intel 8080 released	The chip the hobbyists could buy
1975	Altair 8800 on sale	Personal computing starts in kit form
1975	Microsoft founded	Software becomes a separate industry
1975	MOS 6502 sold at \$25	Cheap CPU for home computers
1976	Apple founded	The other half of the home computer story
1976	Diffie and Hellman's paper	Secrets agreed over an open line
1976	Cray-1 delivered	Vector supercomputing arrives
1977	Apple II, PET and TRS-80	Computers you buy ready to use
1977	RSA public-key algorithm	Signatures and key exchange with maths
1977	Three-network TCP test	The internet idea proven end to end
1978	Intel 8086 released	The x86 line that still runs today
1978	TCP is split into TCP and IP	The thin waist of the internet appears
1978	Kernighan and Ritchie's C book	One small book teaches a generation
1979	VisiCalc released	The first application worth buying a computer for
1979	Usenet created	Global discussion with no owner
1979	Motorola 68000 released	The CPU behind the first Macs and Amigas
1980	Ethernet Blue Book published	Three firms agree one local network
1980	Sinclair ZX80 sold in the UK	Computers under 100 pounds
1981	IBM PC model 5150 announced	The business standard, and clones
1981	RFC 791 and RFC 793 published	IPv4 and TCP written down properly
1981	Xerox Star launched	The desktop metaphor goes on sale
1982	Sun Microsystems founded	Workstations and networked Unix
1982	Commodore 64 released	Best-selling single computer model ever
1982	SMTP defined in RFC 821	Email gets one agreed delivery protocol
1983	TCP/IP flag day, 1 January	The old protocol switched off overnight
1983	DNS proposed in RFC 882	Names stop living in one shared file

Year	Event	Why it mattered
1983	IEEE 802.3 Ethernet standard	Ethernet becomes vendor-neutral
1983	GNU project announced	Free software as an organized movement
1983	Apple Lisa launched	A graphical interface reaches the market
1984	Macintosh launched, 24 January	Graphical computing becomes affordable
1984	Cisco founded	Routing becomes a product
1984	ISO 7498 publishes the OSI model	Seven layers, the teaching vocabulary
1985	Windows 1.0 released	Microsoft enters graphical computing
1985	IEEE 754 floating point published	Every machine rounds the same way
1985	symbolics.com registered, 15 March	The first .com domain name
1985	ARM1 first works at Acorn	The CPU design now in every phone
1985	NSFNET begins	The research backbone that becomes the internet
1986	First IETF meeting	The internet's standards body forms
1987	Perl 1.0 released	Text processing and system glue
1987	RFC 1034 and 1035 published	DNS as it still works today
1987	Toshiba presents NAND flash	The storage in every phone and SSD
1987	IBM VGA introduced	640x480 becomes the baseline display
1988	The Morris worm, 2 November	First internet-wide security incident
1988	NSFNET upgraded to T-1	A 1.544 Mbit/s national backbone
1988	IRC written in Finland	Real-time text chat, still running
1989	Berners-Lee's web proposal	Linked documents over the internet
1989	RFC 1122 host requirements	What every internet host must do
1989	GNU General Public Licence v1	Copyleft in a legal document
1990	ARPANET decommissioned	The original network retires quietly
1990	First web server runs at CERN	info.cern.ch answers requests
1990	Windows 3.0 released	Windows finally sells in volume
1991	Linux kernel announced, 25 August	A free Unix-like kernel appears
1991	Web software released publicly	Anyone can run a server or a browser
1991	Python 0.9.0 released	Readability as a design goal
1991	Unicode 1.0 published	One number per character, all scripts

Year	Event	Why it mattered
1991	PGP released	Strong encryption in ordinary hands
1991	First GSM call made	Digital mobile telephony begins
1992	UTF-8 designed on a placemat	Unicode that does not break old software
1992	JPEG published as a standard	Photographs become small enough to send
1993	Mosaic browser released	Images inline, and the web grows fast
1993	CERN puts the web in public domain	No fee, no permission, no owner
1993	Intel Pentium released	The superscalar x86 era
1993	Doom released, 10 December	3D games and shareware distribution
1993	MP3 standardized in MPEG-1	Music files small enough to share
1994	Netscape founded	The web becomes an industry
1994	W3C founded	The web gets a standards body
1994	Linux 1.0 released	Free Unix reaches production quality
1994	Amazon founded	Retail moves online
1995	Netscape IPO, 9 August	The dot-com boom psychology begins
1995	NSFNET switched off, 30 April	The internet becomes commercial
1995	Windows 95 released, 24 August	The graphical PC becomes the default
1995	Java, JavaScript, PHP and Ruby	Four lasting languages in one year
1995	India's public internet, 15 August	VSNL opens dial-up to the public
1996	USB 1.0 published	One connector to replace many
1996	RFC 1918 private addresses	10.x, 172.16.x and 192.168.x reserved
1996	HTTP/1.0 written down, RFC 1945	The web's protocol described at last
1997	IEEE 802.11 published	Wireless local networking standardized
1997	HTTP/1.1 published, RFC 2068	Reused connections and virtual hosts
1997	Deep Blue beats Kasparov	Search plus hardware beats a champion
1998	Google founded, 4 September	Search becomes the front door
1998	ICANN incorporated	Names and numbers get a caretaker
1998	XML 1.0 published	Machine-readable structured text
1998	IPv6 specified in RFC 2460	128-bit addresses defined
1998	Bluetooth SIG founded	Short-range wireless gets an owner

Year	Event	Why it mattered
1999	Napster launched	Peer-to-peer file sharing at scale
1999	802.11b and the Wi-Fi name	Wireless becomes a consumer product
1999	TLS 1.0 published, RFC 2246	SSL becomes an open standard
1999	Nvidia GeForce 256	The word GPU enters common use
2000	NASDAQ peaks, 10 March	5,048.62, then a long fall
2000	USB 2.0 published	480 Mbit/s makes USB storage practical
2000	GPS accuracy degradation ended	Civil GPS becomes ten times better
2000	DDR SDRAM standardized	Two transfers per clock in main memory
2001	Wikipedia launched, 15 January	Reference work written by everyone
2001	Mac OS X 10.0 released	A Unix core under a consumer desktop
2001	Windows XP released	The PC desktop for a decade
2001	iPod released	Portable digital music goes mainstream
2001	Agile Manifesto published	A named alternative to big up-front plans
2001	IBM POWER4 has two cores	Multicore processors begin
2002	The Bezos API mandate	Teams must talk only through interfaces
2003	AMD Opteron ships x86-64	64-bit on ordinary hardware
2003	PCI Express 1.0 published	Serial lanes replace parallel buses
2003	Skype released	Voice calls over the public internet
2003	Steam launched	Games delivered without discs
2004	Facebook launched, 4 February	Social networking at national scale
2004	Firefox 1.0 released	Browser competition returns
2004	Gmail beta, 1 April	Applications move into the browser
2004	Ubuntu 4.10 released	Linux packaged for ordinary users
2005	BitKeeper licence withdrawn	The trigger for writing git
2005	Git's first commit, 7 April	Distributed version control done right
2005	YouTube founded	Video becomes a normal web object
2005	Dual-core desktop chips ship	The clock speed race ends
2006	Amazon S3 launched, 14 March	Storage rented by the gigabyte-month
2006	Amazon EC2 beta, 25 August	Servers rented by the hour
2006	Twitter launched	Short public messages at scale
2006	Hadoop released	Batch processing across cheap machines

Year	Event	Why it mattered
2007	iPhone announced, 9 January	The pocket computer with a real browser
2007	CUDA released	GPUs become general-purpose compute
2007	Android announced, 5 November	An open phone platform for everyone else
2007	Netflix begins streaming	Video moves from post to packets
2008	Bitcoin white paper, 31 October	Money without a central ledger keeper
2008	Chrome released, 2 September	Fast JavaScript and per-tab isolation
2008	GitHub launched	Code hosting becomes social
2008	First Android phone ships	The two-platform phone world begins
2008	App Store opens, 10 July	Software distribution through one gate
2008	USB 3.0 published	5 Gbit/s over a compatible connector
2009	Bitcoin genesis block, 3 January	The chain actually starts
2009	Node.js released	JavaScript on the server
2009	Go announced, 10 November	Simple concurrency for server software
2009	IEEE 802.11n published	MIMO takes Wi-Fi past 100 Mbit/s
2010	iPad released	Tablets become a product category
2010	Bluetooth 4.0 adds Low Energy	Coin-cell devices join the network
2010	Stuxnet discovered	Software as a weapon against machinery
2010	OpenStack launched	Open-source infrastructure clouds
2011	IANA runs out of IPv4, 3 February	The address shortage becomes real
2011	IBM Watson wins Jeopardy	Statistical language systems go public
2011	Siri ships on the iPhone 4S	Voice assistants reach the mainstream
2011	Smartphones outsell PCs	The centre of computing moves
2012	AlexNet wins ImageNet	Deep learning on GPUs beats everything
2012	World IPv6 Launch, 6 June	IPv6 turned on permanently by big sites
2012	Raspberry Pi released	A 35-dollar computer for learning
2012	DDR4 standard published	Higher density at lower voltage
2013	Docker released, March	Containers become easy to use
2013	Snowden disclosures begin, June	Encryption everywhere becomes the norm

Year	Event	Why it mattered
2013	IEEE 802.11ac published	Gigabit Wi-Fi in the 5 GHz band
2014	Kubernetes announced, June	Cluster orchestration gets a standard
2014	Heartbleed disclosed, 7 April	One buffer bug affects half the web
2014	Swift announced	Apple replaces Objective-C
2014	USB Type-C specified	A reversible connector for everything
2015	HTTP/2 published, RFC 7540	Binary framing and multiplexed streams
2015	Kubernetes 1.0 and the CNCF	Container orchestration becomes neutral
2015	Let's Encrypt begins issuing	Free automated certificates for all
2015	Rust 1.0 released, 15 May	Memory safety without a garbage collector
2015	TensorFlow released	Machine learning frameworks go public
2016	AlphaGo beats Lee Sedol, March	Search plus learning solves Go
2016	Mirai botnet DDoS, 21 October	Insecure home devices break big services
2016	Bluetooth 5.0 published	Four times the range for Low Energy
2016	PyTorch released	Define-by-run makes research faster
2017	Attention Is All You Need, June	The transformer architecture published
2017	WannaCry ransomware, 12 May	Unpatched machines at national scale
2017	PCI Express 4.0 published	16 GT/s per lane
2018	Meltdown and Spectre, 3 January	Speculation leaks data across boundaries
2018	GDPR takes effect, 25 May	Data protection with real penalties
2018	TLS 1.3 published, RFC 8446	Fewer round trips, forward secrecy only
2018	GPT-1 and then BERT	Pretraining plus fine-tuning wins in NLP
2018	Microsoft agrees to buy GitHub	Open source and big vendors converge
2019	GPT-2 released in stages	Staged release becomes a debate
2019	Wi-Fi 6 certification opens	OFDMA brings order to crowded air
2019	PCI Express 5.0 published	32 GT/s per lane
2019	USB4 specification published	Thunderbolt's tunnelling becomes open
2020	GPT-3 shows in-context learning	Scale alone changes what a model can do
2020	Apple M1 ships, November	ARM reaches the desktop with real speed

Year	Event	Why it mattered
2020	DDR5 standard published, July	4800 MT/s and on-module regulation
2020	Remote work at global scale	Video and cloud tools become essential
2021	GitHub Copilot preview, June	Code completion from a language model
2021	Log4Shell disclosed, 9 December	One logging library, global emergency
2021	IEEE 802.11ax approved	Wi-Fi 6 becomes a full standard
2022	PCI Express 6.0 published	PAM4 signalling and forward error correction
2022	HTTP/3 published, RFC 9114	HTTP over QUIC, on top of UDP
2022	The Chinchilla paper, March	Data, not just parameters, sets quality
2022	Stable Diffusion released, August	Image generation with open weights
2022	ChatGPT released, 30 November	Language models reach the general public
2023	GPT-4 released, 14 March	Multimodal input and much stronger results
2023	Llama and Llama 2 released	Open-weight models become serious
2023	Claude released, 14 March	A second major assistant family
2023	EU AI Act agreed, December	The first broad legal framework for AI
2024	Wi-Fi 7 certification, 8 January	320 MHz channels and multi-link operation
2024	Nvidia Blackwell announced	208 billion transistors in two dies
2024	Bluetooth 6.0 published	Channel sounding measures true distance
2024	El Capitan tops the TOP500	About 1.74 exaflops at Livermore
2024	Unicode 16.0 published	154,998 characters defined
2024	EU AI Act in force, 1 August	Obligations start phasing in
2025	PCI Express 7.0 specification	128 GT/s per lane, no products yet
2025	Reasoning models become normal	Models spend compute at answer time
2025	2 nm class production starts	Gate-all-around transistors in volume
2026	Agentic tools in daily use	Models call tools and run multi-step work
2026	Long contexts are routine	Whole codebases fit in one request
2026	On-device inference is common	Small models run on phones and laptops

53.2 Timeline of programming languages

The year given is the year the language first worked or was first released, not the year it was designed or the year it became popular. Those can differ by a decade.

Year	Language	Creator	What it was for
1949	Short Code	Mauchly and Schmitt	Readable arithmetic on UNIVAC
1952	Autocode	Alick Glennie	First compiler, Manchester Mark 1
1952	A-0	Grace Hopper	Assembling reusable subroutines
1956	IPL	Newell, Shaw, Simon	Lists and symbols for early AI
1957	FORTRAN	John Backus, IBM	Scientific and numerical work
1958	ALGOL 58	International committee	Publishing algorithms clearly
1958	LISP	John McCarthy	Symbolic computing and AI
1959	COBOL	CODASYL committee	Business records and reports
1960	ALGOL 60	Naur, Backus and others	Block structure, formal grammar
1962	APL	Kenneth Iverson	Whole-array mathematics
1962	Simula I	Dahl and Nygaard	Simulating real-world systems
1964	BASIC	Kemeny and Kurtz	Teaching students to program
1964	PL/I	IBM	One language for all workloads
1967	Simula 67	Dahl and Nygaard	Classes and objects, the first time
1967	Logo	Papert and colleagues	Teaching children with a turtle
1970	Pascal	Niklaus Wirth	Teaching structured programming
1970	Forth	Charles Moore	Small stack-based control systems
1972	C	Dennis Ritchie	Writing Unix portably
1972	Prolog	Colmerauer and Roussel	Rules and logical deduction
1972	Smalltalk	Alan Kay's team, PARC	Objects, messages, live interfaces
1973	ML	Robin Milner	Proofs, with inferred types
1975	Scheme	Sussman and Steele	A tiny, clean Lisp
1977	awk	Aho, Weinberger, Kernighan	One-line text processing
1979	Bourne shell	Stephen Bourne	Gluing Unix commands together
1979	SQL, first product	Chamberlin and Boyce	Asking questions of tables

Year	Language	Creator	What it was for
1980	Ada	Ichbiah for the US DoD	Long-lived safety-critical systems
1983	C++	Bjarne Stroustrup	Objects with no speed penalty
1984	Verilog	Moorby and Gateway	Describing hardware, not software
1984	MATLAB, commercial	Cleve Moler	Matrix and signal mathematics
1984	Objective-C	Cox and Love	Smalltalk-style objects over C
1985	PostScript	Warnock and Geschke	Telling a printer what a page is
1986	Erlang	Armstrong at Ericsson	Phone switches that never stop
1987	Perl	Larry Wall	Reports, text and system glue
1987	VHDL	US DoD project	Hardware description, IEEE 1076
1989	Bash	Brian Fox for GNU	A free, extended Bourne shell
1990	Haskell	Academic committee	Pure, lazy functional research
1991	Python	Guido van Rossum	Readable general-purpose code
1991	Visual Basic	Microsoft	Building Windows apps quickly
1993	Lua	Ierusalimschy and team	Small language embedded in apps
1993	R	Ihaka and Gentleman	Statistics and data graphics
1995	Java	James Gosling, Sun	One binary on many machines
1995	JavaScript	Brendan Eich, Netscape	Small scripts inside web pages
1995	PHP	Rasmus Lerdorf	Pages built on the server
1995	Ruby	Yukihiro Matsumoto	Programming that feels pleasant
1996	OCaml	INRIA team	ML with objects and speed
2000	C#	Anders Hejlsberg, Microsoft	Applications on .NET
2003	Scala	Martin Odersky	Objects and functions on the JVM
2005	F#	Don Syme, Microsoft	Functional programming on .NET
2007	Clojure	Rich Hickey	A Lisp with immutable data
2009	Go	Griesemer, Pike, Thompson	Simple, fast server programs
2011	Dart	Google	Web apps, later Flutter
2012	Elixir	Jose Valim	Erlang's reliability, nicer syntax

Year	Language	Creator	What it was for
2012	TypeScript	Anders Hejlsberg, Microsoft	Types added to JavaScript
2012	Julia	Bezanson and colleagues	Fast numerical computing
2014	Swift	Chris Lattner, Apple	Safer apps for Apple platforms
2014	Solidity	Gavin Wood and others	Contracts running on Ethereum
2015	Rust 1.0	Graydon Hoare, Mozilla	Safety without a garbage collector
2016	Kotlin 1.0	JetBrains	Concise, safer code for Android
2016	Zig	Andrew Kelley	C's job with fewer surprises
2023	Mojo	Chris Lattner, Modular	Python-like code for AI hardware

53.3 Timeline of standards and protocols

A standard is a document. It says what a thing must do so that two products made by strangers will work together. The number after the name is the document you can look up.

Year	Standard	What it defined
1963	ASCII, ASA X3.4	7-bit codes for 128 characters
1967	ASCII revision	Lower case and control codes settled
1969	RFC 1	The RFC document series itself
1974	Cerf-Kahn TCP paper	Internetworking between networks
1980	RFC 768, UDP	Send a datagram, no promises
1980	Ethernet Blue Book, DIX	10 Mbit/s over one shared cable
1981	RFC 791, IPv4	32-bit addresses and the IP header
1981	RFC 792, ICMP	Error and diagnostic messages
1981	RFC 793, TCP	Ordered, reliable byte streams
1982	RFC 821, SMTP	How mail moves between servers
1982	RFC 826, ARP	IP address to MAC address on a LAN
1983	IEEE 802.3	Ethernet as a vendor-neutral standard
1983	RFC 882 and 883	The first DNS design
1984	ISO 7498, OSI model	Seven named layers for teaching
1985	IEEE 754	Binary floating point, all machines alike
1985	RFC 959, FTP	File transfer with a control channel
1986	ISO 9660	The CD-ROM filesystem

Year	Standard	What it defined
1987	RFC 1034 and 1035	DNS as still used today
1988	IEEE 802.1D	Spanning tree, loop-free bridging
1989	RFC 1122 and 1123	What every internet host must do
1990	IEEE 1076, VHDL	Hardware description as a standard
1991	Unicode 1.0	One code point per character
1992	ITU T.81, JPEG	Lossy image compression
1993	RFC 1459, IRC	Real-time chat rooms
1993	ISO/IEC 11172-3, MP3	Perceptual audio compression
1993	RFC 1519, CIDR	Prefix lengths replace address classes
1993	UTF-8 in the X/Open standard	Unicode as bytes, ASCII-compatible
1994	RFC 1738, URLs	The shape of a web address
1995	SSL 3.0	Encrypted web sessions, now obsolete
1996	RFC 1918	Private address ranges for local use
1996	RFC 1945, HTTP/1.0	Request line, headers, body
1996	USB 1.0	12 Mbit/s, one connector, hot plug
1997	RFC 2068, HTTP/1.1	Keep-alive, host header, chunking
1997	IEEE 802.11	Wireless LAN at 1 and 2 Mbit/s
1997	RFC 2131, DHCP	Automatic address assignment
1998	RFC 2460, IPv6	128-bit addresses, simpler header
1998	RFC 2373, IPv6 addressing	Address types and text notation
1999	RFC 2246, TLS 1.0	SSL becomes an IETF standard
1999	RFC 2616, HTTP/1.1	The version everyone implemented
1999	IEEE 802.11b	11 Mbit/s at 2.4 GHz
1999	IEEE 802.11a	54 Mbit/s at 5 GHz, OFDM
1999	Bluetooth 1.0	Short-range personal wireless links
1999	IEEE 802.1Q	VLAN tags inside Ethernet frames
2000	USB 2.0	480 Mbit/s high speed
2000	JEDEC JESD79, DDR	Data on both clock edges
2001	RFC 3168, ECN	Congestion signalled, not dropped
2003	IEEE 802.11g	54 Mbit/s at 2.4 GHz
2003	PCI Express 1.0	2.5 GT/s serial lanes
2003	JEDEC DDR2	4n prefetch, lower voltage
2003	WPA, then WPA2 in 2004	Wi-Fi encryption that actually works
2004	Bluetooth 2.0 + EDR	About 3 Mbit/s over the same radio

Year	Standard	What it defined
2005	RFC 4271, BGP-4	How networks tell each other routes
2006	RFC 4291, IPv6 addressing	The current IPv6 address rules
2006	RFC 4443, ICMPv6	Errors and neighbour discovery for v6
2006	RFC 4346, TLS 1.1	Fixes for CBC attacks
2007	PCI Express 2.0	5 GT/s per lane
2007	JEDEC DDR3	8n prefetch, 1.5 V
2008	RFC 5246, TLS 1.2	SHA-256 and AEAD cipher suites
2008	RFC 5321, SMTP	The current mail transfer standard
2008	IEEE 754-2008	Decimal formats and fused multiply-add
2008	USB 3.0	5 Gbit/s SuperSpeed, extra wires
2009	IEEE 802.11n	MIMO, 40 MHz, frame aggregation
2009	Bluetooth 3.0 + HS	Wi-Fi used for the bulk transfer
2010	Bluetooth 4.0	Low Energy, GATT, coin-cell devices
2010	PCI Express 3.0	8 GT/s, 128b/130b encoding
2011	RFC 6265, cookies	How browsers store and send cookies
2011	RFC 6455, WebSocket	A two-way channel over one HTTP port
2012	RFC 6598, shared space	100.64.0.0/10 for carrier NAT
2012	RFC 6749, OAuth 2.0	Delegated access without passwords
2012	JEDEC DDR4	1.2 V, bank groups, higher density
2013	IEEE 802.11ac	80 and 160 MHz, 256-QAM, MU-MIMO
2014	RFC 7230 to 7235	HTTP/1.1 rewritten as six documents
2014	USB Type-C 1.0	A reversible 24-pin connector
2014	Bluetooth 4.2	Privacy, longer packets, IPv6 support
2015	RFC 7540, HTTP/2	Binary frames, streams, header compression
2015	RFC 7519, JWT	A signed token you can read
2016	Bluetooth 5.0	Four times the range, twice the speed
2017	IEEE 754-2019 work begins	Refinement of the 2008 revision
2017	PCI Express 4.0	16 GT/s per lane
2017	USB 3.2	20 Gbit/s using two lanes
2018	RFC 8446, TLS 1.3	One round trip, forward secrecy only
2018	RFC 8200, IPv6	IPv6 promoted to full internet standard
2019	PCI Express 5.0	32 GT/s per lane

Year	Standard	What it defined
2019	USB4	Tunnelling, 40 Gbit/s, Type-C only
2019	WPA3 required for new Wi-Fi	Individual encryption, better handshake
2020	JEDEC DDR5	4800 MT/s, on-module power regulation
2021	RFC 9000, QUIC	Reliable streams on top of UDP
2021	IEEE 802.11ax, Wi-Fi 6	OFDMA, 1024-QAM, target wake time
2022	RFC 9110 to 9114	HTTP semantics, and HTTP/3 over QUIC
2022	PCI Express 6.0	64 GT/s, PAM4, forward error correction
2022	USB4 Version 2.0	80 Gbit/s, and 120 Gbit/s one way
2024	Wi-Fi 7 from IEEE 802.11be	320 MHz, 4096-QAM, multi-link operation
2024	Bluetooth 6.0	Channel sounding for true distance
2024	Unicode 16.0	154,998 characters across 168 scripts
2025	PCI Express 7.0	128 GT/s per lane, on paper first

53.4 The glossary

Every line has the same three parts: the term, the plain meaning, then the technical meaning. Where a word means two different things in two fields, both senses are given as separate lines.

■ 53.4.1 A

1. **A record** — the DNS entry holding an IPv4 address — a resource record of type A mapping a name to a 32-bit address; AAAA does the same for IPv6.
2. **ABI** — the binary rulebook two compiled parts must share — application binary interface: calling convention, register use, struct layout, symbol naming.
3. **Abstraction** — hiding detail behind a simpler idea — a layer offering a stable interface while the implementation beneath is free to change.
4. **Activation** — a number flowing through a network as it runs — a layer's output for one input, alive only during the pass and not part of the model.
5. **Activation function** — the bend that stops a network being one straight line — an elementwise nonlinearity such as ReLU, GELU or SwiGLU.
6. **ADC** — turns a real signal into numbers — analogue-to-digital converter, sampling a voltage at a fixed rate and quantizing to a fixed bit depth.
7. **Address space** — every address a program is allowed to name — the range of addresses a process can generate, commonly 2^{48} bytes of user space on 64-bit systems.
8. **AES** — the normal way to scramble bulk data — Advanced Encryption Standard, 128-bit block cipher with 128, 192 or 256-bit keys, FIPS 197, 2001.
9. **Agent** — a model that acts instead of only answering — a loop where a model picks a tool call, the tool runs, the result returns, until a stop condition.
10. **Aliasing** — false patterns from sampling too slowly — signal components above half the sample rate folding back as lower, wrong frequencies.

11. **Alignment** (memory) — putting data at tidy addresses — placing an object at an address that is a multiple of its size, required or faster on most architectures.
12. **Alignment** (AI) — making a model behave as intended — the work of getting model behaviour to match stated human intent, values and constraints.
13. **Allocator** — the code that hands out memory — the runtime or kernel component that finds, records and returns memory blocks to callers.
14. **Alpha channel** — how see-through a pixel is — a per-pixel opacity value, usually 8 bits, used when compositing layers.
15. **ALU** — the part that does the sums — arithmetic logic unit, a combinational block performing add, subtract, bitwise and shift operations and setting flags.
16. **Amdahl's law** — the slow part limits your speedup — speedup is bounded by $1/(s + p/N)$ for serial fraction s over N processors.
17. **Ampere** — how much electricity flows per second — the SI unit of current, one coulomb of charge per second.
18. **Amplitude** — how big a wave is — the peak deviation of a signal from its resting value; in audio it maps to loudness.
19. **Analogue** — smoothly varying, not stepped — a signal that may take any value in a continuous range, unlike a digital signal with discrete levels.
20. **Anti-aliasing** — smoothing jagged edges — MSAA, FXAA, TAA and similar methods that add or reuse samples to reduce sampling artefacts.
21. **Anti-cheat** — software that catches players who cheat — combined client integrity checks, server-authoritative rules and statistical detection.
22. **Anycast** — one address, many machines — announcing the same prefix from several sites so each client reaches the topologically nearest one.
23. **Apex domain** — the bare name with no www — the zone's own name, such as example.com, which cannot hold a CNAME under standard DNS rules.
24. **API** — an agreed way for programs to ask each other for things — the endpoints, functions, arguments and guarantees one piece of software offers another.
25. **Argon2** — a password hash built to be slow and greedy for memory — winner of the 2015 Password Hashing Competition, with time, memory and parallelism costs.
26. **ARM** — the processor family in nearly every phone — a RISC architecture licensed as designs by Arm Ltd, first silicon working in 1985.
27. **ARP** — finding a neighbour's hardware address — Address Resolution Protocol, RFC 826, broadcasting for the MAC address owning an IPv4 address on the link.
28. **ARPANET** — the network the internet grew out of — the ARPA-funded packet network, first message 29 October 1969, retired 1990.
29. **Artifact** — a file a build makes and keeps — a stored pipeline output such as a binary, image or test report, retained for later jobs or download.
30. **ASCII** — numbers for English letters — a 7-bit character set from 1963 with 128 code points, 95 printable and 33 control codes.
31. **ASLR** — moving things so attackers cannot guess addresses — address space layout randomization of stack, heap and library bases at load time.
32. **Assembler** — turns readable instructions into machine code — a translator producing one machine instruction per mnemonic and resolving labels and directives.
33. **Assembly language** — machine code you can read — a notation with one mnemonic per instruction, plus labels, macros and assembler directives.
34. **AST** — a program drawn as a tree — abstract syntax tree, the parser's output, with nodes for

expressions and statements and no punctuation.

35. **Asynchronous** — start now, collect the answer later — an operation returning before completion, signalling later by callback, event, future or poll.
36. **Atomic** — all at once or not at all — an operation no observer can see half-finished, enforced by hardware instructions, locks or transactions.
37. **Attention** — letting each word look at the others — a weighted sum of value vectors, weighted by the scaled dot product of one query with every key.
38. **Attenuation** — a signal weakening with distance — loss of power along a path, measured in decibels, from resistance, absorption or spreading.
39. **Authoritative server** — the machine holding the real answer for a name — a DNS server answering from its own zone data rather than from cache.
40. **Autonegotiation** — two ports agreeing a speed by themselves — Ethernet link partners exchanging capability pulses and choosing the best shared speed and duplex.
41. **Autonomous system** — one network under one routing policy — an AS with a 16 or 32-bit number that announces its prefixes to neighbours using BGP.
42. **Availability** — how much of the time it works — uptime as a fraction; 99.9 percent permits about 43 minutes of downtime per month.
43. **Availability zone** — one isolated site inside a cloud region — a data centre with independent power and cooling, linked to its region at low latency.
44. **AVX** — wide maths instructions on x86 chips — Advanced Vector Extensions, SIMD sets using 256-bit registers, or 512-bit in AVX-512.

■ 53.4.2 B

1. **Backpressure** — telling the sender to slow down — a flow-control signal propagated upstream when a consumer cannot keep up.
2. **Backpropagation** — working out which weight caused which mistake — reverse-mode automatic differentiation applying the chain rule from the loss back to every parameter.
3. **Bandwidth** — how much can flow per second — the data rate of a link in bits per second, distinct from latency, which is delay.
4. **Bandwidth throttling** — deliberately slowing traffic — rate limiting applied by policy at a network device, service or account level.
5. **Bandwidth-delay product** — how much data fits inside the wire — bandwidth multiplied by round-trip time, the amount that must be in flight to keep a link full.
6. **Bank** (memory) — one independently addressed section of a chip — a DRAM subdivision with its own row buffer, allowing one row open per bank.
7. **Bare repository** — a git repository with no working files — a repository containing only the object database and refs, normally used as a push target.
8. **Base model** — the raw model before it is taught manners — a network trained only on next-token prediction, not yet instruction-tuned or preference-tuned.
9. **Base station** — the mast your phone talks to — the mobile network's radio equipment serving one or more cells, called an eNodeB in LTE and gNodeB in 5G.
10. **Batch** — a handful of examples processed together — the group of samples whose gradients are averaged before one parameter update.
11. **bcrypt** — an old but sound password hash — a 1999 adaptive hash based on Blowfish, with a cost factor that raises work as hardware improves.
12. **Beamforming** — aiming a radio signal instead of shouting everywhere — controlling phase across several antennas so signals add in one direction and cancel elsewhere.

13. **Benchmark** — a test everyone runs so results can be compared — a fixed task and scoring rule; it measures the benchmark first and the world second.
14. **BGP** — how networks tell each other which routes they have — Border Gateway Protocol, RFC 4271, a path-vector protocol carrying prefixes and AS paths between autonomous systems.
15. **Bias** (AI) — the number added after the multiply — a per-unit learned constant added to the weighted sum before the activation function.
16. **Big-endian** — the biggest byte first — a byte order storing the most significant byte at the lowest address; network byte order is big-endian.
17. **Binary** — counting with only 0 and 1 — base-2 positional notation, where each place is worth twice the one to its right.
18. **Binary search** — halving the haystack each time — locating a value in sorted data in $O(\log n)$ comparisons by discarding half the range each step.
19. **Binning** — sorting finished chips by how good they turned out — testing dies and selling them at the highest speed and core count they pass.
20. **BIOS** — the old start-up firmware on PCs — the 16-bit firmware that initializes hardware and loads a 512-byte boot sector, superseded by UEFI.
21. **Bisect** — a binary search through history for the bad commit — git bisect, halving the commit range each test to find the first bad commit in $\log_2(n)$ steps.
22. **Bit** — one on-or-off — a binary digit, the smallest unit of information, value 0 or 1.
23. **Bit depth** — how many steps a sample can take — the number of bits per sample; 16-bit audio gives 65,536 levels and about 96 dB of dynamic range.
24. **Bit rot** — data quietly going bad — undetected corruption of stored bits from charge leakage, media decay or cosmic rays, caught only by checksums.
25. **BLE** — the low-power half of Bluetooth — Bluetooth Low Energy, introduced in version 4.0 in 2010, using 40 channels and short connection events.
26. **BLE GATT** — the rules for reading a small value over Bluetooth — Generic Attribute Profile, a table of services and characteristics addressed by handle and UUID.
27. **Blob** — git's word for a file's contents — a git object holding raw bytes, named by the SHA-1 or SHA-256 of a header plus the content.
28. **Block** (storage) — the smallest lump a drive reads or writes — a fixed-size unit, 512 bytes or 4096 bytes on disks, and typically 128 KiB to 4 MiB in NAND erase blocks.
29. **Blocking** — waiting instead of doing something else — a call that does not return until its work completes, holding the calling thread.
30. **Boolean algebra** — the maths of true and false — an algebra over $\{0,1\}$ with AND, OR and NOT, shown by Shannon in 1937 to describe switching circuits.
31. **Boot sector** — the very first block a machine reads from a disk — the first 512-byte sector, holding MBR partition entries and a small loader on legacy systems.
32. **Bootloader** — the small program that starts the real one — code that loads and hands control to a kernel, such as GRUB, systemd-boot or U-Boot.
33. **Bootstrapping** — a compiler built with itself — writing a compiler for a language in that same language, after first compiling it with another one.
34. **Boule** — the big silicon crystal a wafer is sliced from — a single-crystal ingot, typically 300 mm across, pulled from a melt by the Czochralski process.
35. **BPE** — cutting text into common chunks — byte-pair encoding, repeatedly merging the most frequent adjacent pair to build a fixed-size subword vocabulary.
36. **Branch** (CPU) — an instruction that may jump — a control-flow instruction whose target or direction may depend on a condition.

37. **Branch** (git) — a moving name for a line of work — a ref under refs/heads that points at one commit and moves forward as you commit.
38. **Branch prediction** — guessing which way a jump goes before knowing — hardware predicting direction and target so the pipeline keeps fetching; a wrong guess costs the pipeline depth.
39. **Bridge** — a device joining two network segments — a layer 2 device forwarding frames by learned MAC address; a switch is a multi-port bridge.
40. **BSSID** — the identity of one specific access point — the 48-bit MAC address of an access point's radio, unlike an SSID, which is a network name.
41. **Buffer** — a waiting area for data — memory holding data between a producer and a consumer running at different speeds.
42. **Buffer overflow** — writing past the end of a box — writing beyond an allocation's bounds, corrupting adjacent memory and often control data such as return addresses.
43. **Bufferbloat** — too much queueing makes everything feel slow — excessive buffering in network devices that inflates latency under load without preventing loss.
44. **Bus** — a shared set of wires — a set of conductors and rules carrying address, data and control signals between components.
45. **Byte** — eight bits, the usual unit of size — an octet, 8 bits, able to hold 256 distinct values.
46. **Bytecode** — halfway between source and machine code — a compact instruction set for a virtual machine, such as JVM bytecode or Python's .pyc format.

■ 53.4.3 C

1. **Cache** — a small fast copy of things you keep needing — memory holding recently or likely used data closer to the consumer than the original.
2. **Cache coherence** — keeping every core's copy honest — a protocol such as MESI ensuring all cores see one consistent value for each cache line.
3. **Cache hit** — the copy was already there — a request satisfied from the cache without going to the slower backing store.
4. **Cache line** — the block a cache moves as one unit — the unit of transfer between cache levels, 64 bytes on most current x86 and ARM designs.
5. **Cache miss** — the copy was not there — a request that must be served from the next level, costing that level's latency.
6. **Callback** — code you hand over to be run later — a function passed to another routine to be invoked when an event or completion occurs.
7. **Canary release** — trying a change on a few users first — routing a small share of traffic to a new version and watching error rates before wider rollout.
8. **Capability** — permission attached to a token, not a person — a transferable right to do a specific operation, held as an unforgeable reference.
9. **Capacitor** — a component that stores a little charge — two conductors separated by a dielectric, storing charge Q equal to capacitance C times voltage V .
10. **CDN** — copies of a site kept near users — content delivery network: caches in many locations, steered by DNS or anycast to reduce distance and origin load.
11. **Certificate** — a signed statement that a key belongs to a name — an X.509 document binding a public key to identifiers, signed by an issuer and valid between two dates.
12. **Certificate authority** — an organization browsers already trust — a CA whose root certificate ships in trust stores and which signs certificates after validating control of a name.
13. **Certificate pinning** — accepting only one specific certificate — a client refusing certificates other than pinned keys, ignoring the wider CA set.

14. **CGNAT** — your provider sharing one public address between many homes — carrier-grade NAT, usually using 100.64.0.0/10 from RFC 6598.
15. **cgroup** — a fence around how much a process may use — a Linux control group limiting and accounting CPU, memory, I/O and process counts for a set of tasks.
16. **Chain of trust** — a signed line from a root to your site — the verification path from a trusted root through intermediates to the leaf certificate.
17. **Check run** — one automated test result on a commit — a status object on a forge, attached to a specific commit SHA, with a conclusion such as success or failure.
18. **Checkout** — putting a commit's files into your folder — updating the working tree and HEAD to match a specified commit, branch or path.
19. **Checksum** — a small number that proves nothing got mangled — a value computed over data and compared after transfer or storage to detect accidental change.
20. **Cherry-pick** — copying one commit onto another branch — applying a single commit's diff elsewhere, creating a new commit with a new hash.
21. **Chinchilla** — the finding that models were trained on too little data — the 2022 DeepMind result giving compute-optimal token and parameter ratios.
22. **Chiplet** — a big chip built from several smaller ones — separate dies packaged together and joined by a high-speed interconnect, improving yield and mixing process nodes.
23. **Chunk** — a piece of a bigger thing sent on its own — a bounded portion of data transferred or processed separately, as in chunked transfer encoding.
24. **CIDR** — writing an address range as a prefix — Classless Inter-Domain Routing, RFC 1519, notation such as 192.168.0.0/24 giving address and prefix length.
25. **Cipher suite** — the exact set of algorithms a connection uses — the negotiated combination of key exchange, authentication, bulk cipher and MAC or AEAD mode.
26. **Circuit switching** — a dedicated path reserved for one call — reserving capacity end to end for the call's duration, unlike packet switching, which shares links.
27. **Client prediction** — the game moving you before the server replies — applying local input immediately and reconciling later against authoritative server state.
28. **Clock** — the machine's heartbeat — a periodic square wave defining the instants at which state elements sample their inputs.
29. **Clock domain** — a region sharing one clock — part of a chip driven by one clock, with synchronizers needed for signals crossing to another domain.
30. **Closure** — a function that remembers where it came from — a function value together with captured bindings from its defining scope.
31. **Cloud region** — a geographic cluster of data centres — a named location such as ap-south-1, containing several availability zones with independent failure domains.
32. **CMOS** — the way nearly all logic is built — complementary metal-oxide-semiconductor, pairing n-channel and p-channel FETs so steady states draw almost no current.
33. **CNAME** — a name that points to another name — a DNS record aliasing one name to another, after which resolution restarts at the target.
34. **Coalescing** — joining small pieces into one bigger one — combining adjacent free blocks, memory accesses or writes to reduce overhead.
35. **Codec** — the thing that squeezes and unsqueezes media — coder-decoder: the algorithm pair compressing and decompressing audio or video, such as AAC, H.264 or AV1.
36. **Codepoint** — the number that identifies a character — a Unicode scalar value from U+0000 to U+10FFFF, excluding the surrogate range.
37. **Cold start** — the first call being slow — the delay while a serverless platform allocates and initial-

izes an execution environment before running your code.

38. **Cold storage** — cheap storage you rarely read — archival tiers with low cost per gigabyte and retrieval delays measured in minutes to hours.
39. **Colour space** — the agreed meaning of a colour number — a defined set of primaries, white point and transfer function, such as sRGB, Display P3 or Rec. 2020.
40. **Commit** — a saved snapshot with a message — a git object naming a tree, zero or more parents, author, committer, timestamps and a message, identified by its hash.
41. **Compaction** — tidying storage by rewriting it — merging and rewriting data to reclaim space, as in log-structured stores and SSD garbage collection.
42. **Compiler** — turns source code into machine code — a program translating a whole source program into another language, usually machine code, before it runs.
43. **Compression** — making data smaller — encoding that reduces size, lossless if the original returns exactly, lossy if some information is discarded.
44. **Concurrency** — several things in progress at once — structuring work so tasks overlap in time; parallelism means they literally run at the same instant.
45. **Conflict** (git) — two edits to the same lines — a merge where the same region changed on both sides, leaving markers for a human to resolve.
46. **Constant folding** — doing the maths at compile time — a compiler optimization evaluating constant expressions during compilation.
47. **Container** — an isolated box for one program and its files — a process group isolated by kernel namespaces and limited by cgroups, sharing the host kernel.
48. **Container** (media) — the wrapper holding audio and video together — a file format such as MP4, MKV or WebM carrying encoded streams, timing and metadata.
49. **Contamination** — a model already saw the exam — benchmark test data appearing in training data, inflating scores without matching ability.
50. **Context switch** — the CPU swapping one task for another — saving one thread's registers and state, then loading another's, costing roughly one to a few microseconds.
51. **Context window** — how much a model can read at once — the maximum number of tokens a model may attend over in one request, including its own output.
52. **Continuous delivery** — every good build is ready to ship — an automated pipeline that keeps the main branch in a deployable state, with release as a decision.
53. **Continuous integration** — merging often, tested every time — the practice of integrating work frequently, with an automated build and test on every change.
54. **Control plane** — the part that decides — the systems computing routes, policy and configuration, as opposed to the data plane that moves packets.
55. **Cookie** — a small note a site asks your browser to keep — a name-value pair stored by the browser and returned on matching requests, defined by RFC 6265.
56. **Copy-on-write** — share until someone writes — mapping the same pages into several users and duplicating a page only when one of them modifies it.
57. **Core** — one complete processor inside a chip — an independent set of execution units, registers and L1 caches able to run its own instruction stream.
58. **Coroutine** — a function that can pause and resume — a routine with suspendable execution state, allowing cooperative multitasking without OS threads.
59. **CORS** — a site giving permission for another site to read it — Cross-Origin Resource Sharing: response headers letting a browser relax the same-origin policy.
60. **CPU** — the part that follows the instructions — central processing unit, the general-purpose pro-

cessor fetching, decoding and executing instructions.

61. **CPU-bound** — limited by thinking, not waiting — a workload whose completion time is set by processor throughput rather than I/O.
62. **CRC** — a strong checksum for detecting damage — cyclic redundancy check, polynomial division over GF(2); CRC-32 protects Ethernet frames.
63. **Cross-compilation** — building on one machine for another — producing binaries for a target architecture or OS different from the build host.
64. **Cryptographic hash** — a fingerprint of any data — a one-way function producing a fixed-size digest, where finding collisions is computationally infeasible.
65. **CSMA/CA** — listen before you transmit, and back off — Carrier Sense Multiple Access with Collision Avoidance, the Wi-Fi access method with random backoff.
66. **CSRF** — a site tricking your browser into using your login — cross-site request forgery, defeated by anti-forgery tokens and SameSite cookies.
67. **CSS** — the rules that decide how a page looks — Cascading Style Sheets: selectors and declarations applied by cascade, specificity and inheritance.
68. **Culling** — not drawing what nobody can see — discarding geometry outside the view frustum, facing away, or hidden behind other objects.
69. **Current** — the flow of electricity — the rate of charge flow, measured in amperes; one ampere is one coulomb per second.
70. **Czochralski** — the way a silicon crystal is grown — dipping a seed crystal into molten silicon and slowly pulling and rotating it to form a single-crystal boule.

■ 53.4.4 D

1. **DAC** — turns numbers back into a real signal — digital-to-analogue converter, reconstructing a voltage from samples at a fixed rate.
2. **Daemon** — a program that runs quietly in the background — a long-lived process with no controlling terminal, usually started at boot.
3. **Datagram** — a message sent with no promises — a self-contained packet delivered without connection setup, ordering or retransmission.
4. **dBm** — signal strength on a logarithmic scale — power relative to one milliwatt; -50 dBm is strong Wi-Fi and -85 dBm is marginal.
5. **DDR** — memory that moves data on both clock edges — double data rate SDRAM; DDR5, standardized 2020, starts at 4800 MT/s.
6. **Dead code** — code that can never run — instructions unreachable on any path, normally removed by the compiler.
7. **Deadlock** — two tasks each waiting for the other — a cycle of hold-and-wait dependencies where no participant can proceed.
8. **Debouncing** — ignoring a switch's noisy chatter — filtering multiple contact transitions into one clean event, typically over 5 to 50 ms.
9. **Decibel** — a way to write huge ratios in small numbers — ten times the base-10 logarithm of a power ratio; 3 dB is roughly double.
10. **Decode (AI)** — writing the answer one token at a time — the autoregressive generation phase, one token per forward pass, memory-bandwidth bound.
11. **Decoder-only** — the transformer shape used by chat models — a stack of masked self-attention blocks with no separate encoder.
12. **Deep learning** — networks with many stacked layers — machine learning using multi-layer neural networks trained by gradient descent.

13. **Default route** — where to send anything you have no rule for — the 0.0.0.0/0 entry, or ::/0 for IPv6, in a routing table.
14. **Defragmentation** — putting a file's pieces back together — rearranging blocks so files occupy contiguous space, useful on disks and pointless on SSDs.
15. **Delta compression** — storing only what changed — encoding an object as a difference against a similar one, as git does inside packfiles.
16. **Demand paging** — loading pages only when touched — bringing a page into memory on its first access, signalled by a page fault.
17. **Dependency** — something your code needs to work — an external library, service or tool required to build or run software.
18. **Dependency injection** — passing in what a thing needs — supplying collaborators from outside rather than constructing them internally.
19. **Detached HEAD** — you are on a commit, not on a branch — HEAD holding a commit hash directly, so new commits belong to no branch.
20. **Deterministic lockstep** — every machine simulates the same thing — sending only inputs and relying on identical simulation on all clients.
21. **DHCP** — the network hands you an address automatically — Dynamic Host Configuration Protocol, RFC 2131, using the DORA exchange to lease settings.
22. **Die** — one chip cut from a wafer — a single rectangle of processed silicon containing a complete circuit.
23. **Dielectric** — the insulating filling in a capacitor — a non-conducting material that polarizes in a field and raises capacitance.
24. **Diffie-Hellman** — agreeing a secret over an open line — key exchange where each side combines its private value with the other's public value.
25. **Diode** — a one-way valve for electricity — a PN junction conducting when forward biased above about 0.7 V in silicon.
26. **Directory** — a folder holding names — a filesystem object mapping names to inode numbers or file records.
27. **Dirty bit** — a mark saying this copy changed — a flag showing a page or cache line differs from the backing store and must be written back.
28. **Disk image** — a file holding a whole drive's contents — a byte-for-byte copy of a block device, including partition table and filesystems.
29. **Distributed system** — several machines pretending to be one — a system whose components run on networked machines and coordinate by message passing.
30. **Distro** — one packaged version of Linux — a distribution bundling a kernel, libraries, package manager and defaults, such as Debian or Fedora.
31. **DMA** — devices moving data without bothering the CPU — direct memory access, where a controller transfers to or from memory and interrupts on completion.
32. **DNS** — the phone book turning names into addresses — a distributed hierarchical database of resource records, RFC 1034 and 1035.
33. **DNS resolver** — the helper that does the lookups for you — a recursive server that queries root, TLD and authoritative servers and caches results.
34. **Docker** — the tool that made containers easy — a container engine with an image format, build file and registry workflow, released 2013.
35. **Docker image** — a frozen filesystem plus start instructions — a layered, content-addressed bundle of filesystem layers and configuration metadata.
36. **DOM** — the page as a tree the code can change — Document Object Model, the browser's live

object tree built from parsed HTML.

37. **Doping** — adding impurities to make silicon useful — introducing donors such as phosphorus or acceptors such as boron to make n-type or p-type material.
38. **DPI** — dots per inch on paper — printer resolution; PPI is the screen equivalent, pixels per inch.
39. **DPO** — training from preferences without a reward model — direct preference optimization, fitting the policy directly to pairs of preferred and rejected answers.
40. **DRAM** — the main memory that needs constant refreshing — dynamic RAM storing a bit as charge on a capacitor, refreshed roughly every 32 or 64 ms.
41. **Draw call** — one order sent to the graphics card — a command to render a batch of geometry with one state and shader set.
42. **Driver** — the code that speaks a device's language — kernel or user-space software translating generic requests into device-specific commands.
43. **DROP** — silently binning a packet — a firewall action discarding a packet with no reply, so the sender only sees a timeout.
44. **Duplex** — whether both ends can talk at once — full duplex allows simultaneous send and receive; half duplex allows only one direction at a time.
45. **Dynamic linking** — libraries joined at start-up, not build time — resolving symbols at load or first use, sharing one library copy across processes.

■ 53.4.5 E

1. **eBPF** — running safe little programs inside the kernel — an in-kernel virtual machine with a verifier, used for tracing, networking and security.
2. **ECC (memory)** — memory that corrects its own small errors — error-correcting code memory, typically SECDED, fixing single-bit and detecting double-bit errors.
3. **ECC (crypto)** — strong keys that are short — elliptic curve cryptography; a 256-bit curve key is roughly as strong as a 3072-bit RSA key.
4. **ECMP** — spreading traffic over several equally good paths — equal-cost multipath, hashing a flow's five-tuple to pick one of several next hops.
5. **Edge case** — the odd input that breaks things — a condition at the boundary of valid input, often untested.
6. **Edge computing** — doing work near the user instead of far away — running compute at points of presence close to clients to cut latency.
7. **Elasticity** — capacity that grows and shrinks with demand — automatic scaling of resources in response to load, with matching cost changes.
8. **ELF** — the executable format on Linux — Executable and Linkable Format, with a header, program headers for loading and section headers for linking.
9. **Embedding** — a meaning turned into a list of numbers — a learned dense vector whose geometry places related items close together.
10. **Emulation** — one machine pretending to be another — software reproducing another system's instruction set and devices closely enough to run its software.
11. **Encapsulation (networks)** — wrapping one message inside another — placing a higher-layer unit into a lower-layer payload with its own header.
12. **Encoder** — the half that reads rather than writes — a transformer stack producing representations of an input without generating tokens.
13. **Endianness** — which end of a number comes first — byte order in memory; little-endian stores the least significant byte at the lowest address.
14. **Endpoint** — one address a service answers on — a specific URL and method combination, or a

network address plus port.

15. **Entropy** — how surprising the data is — the average information per symbol in bits, the floor for lossless compression.
16. **Environment variable** — a setting passed to a program at start — a name-value pair inherited through the process environment, such as PATH or HOME.
17. **Ephemeral port** — the temporary port your side uses — a short-lived source port, usually from 32768 to 60999 on Linux, chosen per connection.
18. **Epoch** (training) — one full pass through the data — one complete presentation of the training set to the learner.
19. **Epoch** (time) — the moment counting starts — the Unix epoch, 00:00:00 UTC on 1 January 1970, the zero point for time_t.
20. **Error budget** — the amount of failure you are allowed — the difference between 100 percent and an SLO, spendable on risky changes.
21. **Etching** — cutting the pattern into the wafer — removing material chemically or with plasma where photoresist does not protect it.
22. **Ethernet** — the wired local network everyone uses — IEEE 802.3 framing and access rules, from 10 Mbit/s in 1983 to 800 Gbit/s today.
23. **EUV** — the light used to print the smallest features — extreme ultraviolet lithography at 13.5 nm wavelength, in production since 2019.
24. **Eventual consistency** — everyone agrees, given a moment — a model where replicas converge if updates stop, without guaranteeing immediate agreement.
25. **Exception** — an error that jumps out of normal flow — a signalled condition transferring control to a handler and unwinding the stack.
26. **Exit code** — the number a program leaves behind — a process's status, 0 for success by convention and 1 to 255 for failure.
27. **Exponent** — the part of a float that sets the scale — the biased power-of-two field in IEEE 754, 8 bits in binary32 and 11 in binary64.
28. **Extension** — the bit after the dot in a filename — a naming convention, not a guarantee; the real type comes from content or magic numbers.

■ 53.4.6 F

1. **Fan-out** — how many things one thing drives — the number of inputs a single output feeds, limiting speed in logic and load in messaging.
2. **Fast-forward** — the branch just slides forward — a merge where the target is an ancestor of the source, so the ref simply moves, creating no commit.
3. **Fault tolerance** — carrying on despite broken parts — designing so component failure degrades service instead of stopping it.
4. **Feature flag** — a switch that turns a feature on or off — configuration allowing behaviour to change at runtime without redeploying.
5. **Fetch** (git) — download new objects but change nothing — updating remote-tracking refs and objects without touching your branches or working tree.
6. **FIFO** — first in, first out — a queue discipline, or a named pipe file created with mkfifo.
7. **File descriptor** — the small number a program uses to name an open file — a per-process index into the kernel's open file table; 0, 1 and 2 are stdin, stdout, stderr.
8. **File handle** — the object behind an open file — the kernel structure holding the file's position, mode and reference count.
9. **Filesystem** — the way files are laid out on a device — the on-disk structures and rules holding

names, metadata and data blocks, such as ext4, APFS or NTFS.

10. **Fine-tuning** — extra training on your own examples — continued training of a pretrained model on a smaller task-specific dataset.
11. **FinFET** — a transistor with a fin sticking up — a 3D transistor whose gate wraps a thin silicon fin, in production from 2011 to about the 3 nm class.
12. **Firewall** — a guard deciding which packets may pass — a device or kernel subsystem filtering traffic by rules over addresses, ports, state and content.
13. **Firmware** — software living in the device itself — code stored in non-volatile memory on a device, run before or beneath the operating system.
14. **Firmware update** — replacing the software inside a device — writing new code to a device's flash, usually with A/B slots and a signature check.
15. **Fixed timestep** — the game's physics ticks at one steady rate — advancing simulation in constant increments regardless of frame time, with interpolation for display.
16. **Flash memory** — storage that keeps data with no power — floating-gate or charge-trap cells holding charge, erased in blocks and written in pages.
17. **Flip-flop** — a circuit that remembers one bit — an edge-triggered storage element capturing its input at a clock edge.
18. **Floating point** — a number with a moving decimal point — a sign, exponent and fraction encoding under IEEE 754, trading precision for range.
19. **Flow control** — stopping a fast sender flooding a slow receiver — TCP's advertised receive window, or the equivalent at other layers.
20. **Flush** — pushing buffered data out now — forcing pending writes from a buffer to the next level, such as fsync to storage.
21. **Fork (process)** — making a copy of the running program — the Unix call creating a child process with a copied address space, usually copy-on-write.
22. **Fork (repository)** — your own copy of someone's project — a server-side clone under your account, with its own refs and pull-request relationship.
23. **Forward proxy** — a middleman your client chooses — a server making requests on a client's behalf, often for policy or caching.
24. **Forward secrecy** — yesterday's traffic stays safe if today's key leaks — using ephemeral key exchange so long-term key compromise cannot decrypt past sessions.
25. **Fragment (IP)** — a packet cut up to fit — an IP packet split to fit a link's MTU, reassembled at the destination.
26. **Frame (network)** — the layer 2 envelope — the Ethernet or Wi-Fi unit with MAC addresses, payload and a frame check sequence.
27. **Frame (graphics)** — one complete picture — one full rendered image; at 60 Hz the frame budget is 16.67 ms.
28. **Frame budget** — the time you have for one picture — the per-frame time allowance, 16.67 ms at 60 Hz and 6.94 ms at 144 Hz.
29. **Frame rate** — pictures per second — displayed frames per second; smoothness also depends on frame time consistency, not the average alone.
30. **Framebuffer** — the memory holding the picture to be shown — a region of memory holding pixel values scanned out to a display.
31. **Free software** — software you may run, study, change and share — software distributed with the four freedoms, usually under a copyleft licence such as the GPL.
32. **Frequency** — how many cycles per second — cycles per second in hertz; a 4 GHz clock has a period of 250 picoseconds.

33. **FTL** — the SSD's private map of where data really is — flash translation layer, mapping logical block addresses to physical pages and handling wear and garbage collection.
34. **Full-duplex** — talking and listening at the same time — simultaneous bidirectional transmission on separate paths or with echo cancellation.
35. **Function call** — jumping into a named block and coming back — transferring control with arguments and a return address, following the ABI's calling convention.
36. **Fuzzing** — throwing random rubbish at a program to break it — automated testing with mutated or generated inputs, watching for crashes and assertion failures.

■ 53.4.7 G

1. **Game loop** — read input, update, draw, repeat — the cycle sampling input, advancing simulation and rendering, once per frame.
2. **Gamma** — the curve between number and brightness — a nonlinear transfer function matching encoding to perception, roughly 2.2 for sRGB.
3. **Garbage collection** — memory tidied up automatically — a runtime reclaiming unreachable objects, by tracing or reference counting.
4. **Gate** (logic) — a tiny circuit applying one logical rule — a circuit implementing AND, OR, NOT, NAND, NOR or XOR over binary inputs.
5. **Gate** (transistor) — the leg that switches the flow — the control terminal whose voltage forms or removes the channel between source and drain.
6. **Gateway** — the way out of your local network — the router a host sends to when the destination is not on its own subnet.
7. **GATT** — how Bluetooth devices expose small values — Generic Attribute Profile, a hierarchy of services, characteristics and descriptors addressed by handle.
8. **GDDR** — memory built for graphics cards — graphics DDR SDRAM tuned for very high bandwidth with wide buses and high clock rates.
9. **Gigabit** — a thousand million bits — 10^9 bits; note that 1 Gbit/s is 125 MB/s before protocol overhead.
10. **GIL** — only one thread runs Python bytecode at a time — CPython's global interpreter lock, serializing bytecode execution across threads.
11. **Git** — the version control system almost everyone uses — a content-addressed distributed VCS created by Linus Torvalds in April 2005.
12. **Golden image** — the trusted base copy you build from — a validated machine or container image used as the starting point for deployments.
13. **Goodput** — the useful data rate, not the raw one — application-level throughput after headers, retransmissions and protocol overhead.
14. **Goroutine** — a very cheap thread in Go — a lightweight task multiplexed onto OS threads by the Go runtime scheduler.
15. **GPIO** — a pin you can set high or low yourself — general-purpose input/output, a configurable digital pin on a microcontroller or board.
16. **GPT** (partition) — the modern partition table — GUID Partition Table, part of UEFI, supporting large disks and 128 partitions by default.
17. **GPT** (model) — a generative pretrained transformer — a decoder-only transformer trained to predict the next token.
18. **GPU** — the chip built for many small sums at once — a processor with thousands of simple lanes, built for throughput on parallel work.
19. **Graceful degradation** — losing features instead of falling over — designing so partial failure re-

duces function rather than causing an outage.

20. **Gradient clipping** — stopping an update from being too big — rescaling gradients whose norm exceeds a threshold, to keep training stable.
21. **Gradient descent** — walking downhill on the error — updating parameters against the gradient of the loss, scaled by the learning rate.
22. **GraphQL** — the client asks for exactly the fields it wants — a query language and runtime with one endpoint, a typed schema and client-shaped responses.
23. **Ground** — the reference point voltages are measured from — the common return path defined as zero volts in a circuit.
24. **gRPC** — fast structured calls between services — a remote procedure call framework using Protocol Buffers over HTTP/2, with streaming support.
25. **Guard clause** — checking the bad case first and leaving — an early return that removes nesting from the main path.
26. **GUI** — pointing at pictures instead of typing commands — a graphical user interface with windows, icons, menus and a pointer.

■ 53.4.8 H

1. **Hallucination** — a model stating something untrue with confidence — fluent output not grounded in training data or provided context.
2. **Handover** — your phone moving to the next mast without dropping — transferring an active connection between cells while keeping the session alive.
3. **Handshake** — the greeting that sets up a connection — the exchange establishing parameters, such as TCP's SYN, SYN-ACK, ACK.
4. **Hard link** — two names for the same file — a second directory entry pointing at the same inode, with a shared link count.
5. **Hash table** — a lookup that is fast because it computes where to look — a map using a hash of the key to index buckets, average $O(1)$ lookup.
6. **HBM** — memory stacked right next to the processor — high bandwidth memory, stacked DRAM dies on an interposer, giving terabytes per second.
7. **HDD** — a drive with spinning magnetic platters — a hard disk with rotating platters and moving heads, giving milliseconds of seek latency.
8. **HDR** — brighter brights and darker darks — high dynamic range imaging with wider luminance range, usually with a PQ or HLG transfer function.
9. **HEAD** — where you are in the repository right now — a ref pointing to the current branch, or directly to a commit when detached.
10. **Header** — the label on the front of a message — the fixed or structured prefix carrying addressing and control information before the payload.
11. **Heap** — memory you ask for and give back yourself — the region for dynamic allocation, managed by the allocator rather than by scope.
12. **Heat sink** — metal that carries heat away — a finned metal block spreading and radiating heat from a package to moving air or liquid.
13. **Hertz** — cycles per second — the SI unit of frequency, one cycle per second.
14. **Hexadecimal** — counting in sixteens — base-16 notation using 0 to 9 and A to F, where one digit encodes exactly four bits.
15. **Hit rate** — how often the cache had it — the fraction of requests served from cache, the main determinant of average latency.
16. **Hop** — one router along the way — a single forwarding step; traceroute reveals hops by increasing

the TTL.

17. **Hotfix** — an urgent small change — a minimal patch shipped outside the normal release cycle to fix a live problem.
18. **HTML** — the markup that says what parts a page has — HyperText Markup Language, a tree of elements the browser parses into the DOM.
19. **HTTP** — the rules browsers and servers use to talk — a request-response protocol with methods, status codes and headers, now RFC 9110.
20. **HTTPS** — HTTP inside an encrypted tunnel — HTTP carried over TLS, normally on port 443, giving confidentiality, integrity and server authentication.
21. **Huffman coding** — short codes for common symbols — an optimal prefix-free code built from symbol frequencies, used inside DEFLATE and JPEG.
22. **Hyperparameter** — a setting you choose before training — a value not learned by gradient descent, such as learning rate, batch size or layer count.
23. **Hyper-threading** — one core pretending to be two — simultaneous multithreading, running two instruction streams over one core's execution resources.
24. **Hypervisor** — software that runs whole machines inside one machine — a layer creating and scheduling virtual machines, type 1 on bare metal or type 2 on a host OS.

■ 53.4.91

1. **IaaS** — renting raw machines and networks — infrastructure as a service, where you manage the OS and everything above it.
2. **IAM** — who is allowed to do what — identity and access management: principals, policies, roles and permission evaluation.
3. **ICMP** — the network's error and echo messages — the control protocol carrying unreachable, time exceeded and echo, RFC 792.
4. **Idempotent** — doing it twice changes nothing more — an operation whose repeated application has the same effect as one application.
5. **Idle** — doing nothing on purpose — a scheduler state with no runnable task, usually entering a low-power CPU state.
6. **IEEE 754** — the rule that makes floating point the same everywhere — the 1985 standard defining binary32, binary64, rounding modes, infinities and NaN.
7. **Image** (container) — a frozen filesystem plus start-up settings — a layered, content-addressed bundle of layers and configuration.
8. **IMEI** — the serial number identifying a phone itself — a 15-digit equipment identity, distinct from the SIM's subscriber identity.
9. **Immutable** — it cannot be changed after it is made — a value or object whose state is fixed at creation, simplifying sharing and caching.
10. **Impedance** — how much a component resists an alternating signal — the complex ratio of voltage to current; speakers are commonly 4 or 8 ohms.
11. **Index** (git) — the staging area between edits and a commit — a binary file listing paths, modes and blob hashes for the next commit.
12. **Inductor** — a coil that resists sudden changes in current — a component storing energy in a magnetic field, with voltage proportional to di/dt .
13. **Inference** — using a trained model to get an answer — the forward pass only, with no gradients and no parameter updates.
14. **Infrastructure as code** — servers described in files, not clicked into existence — declarative resource definitions kept in version control and applied by a tool.

15. **init** — the first process the kernel starts — PID 1, which starts services and adopts orphaned processes.
16. **Inode** — the record holding a file's facts — a filesystem structure with mode, owner, times, size and block pointers, but not the name.
17. **Instruction pipeline** — several instructions in progress at once — overlapping fetch, decode, execute and write-back stages to raise throughput.
18. **Instruction set** — the list of operations a CPU understands — the ISA: opcodes, registers, addressing modes and the binary encoding.
19. **Instruction tuning** — teaching a model to follow requests — supervised fine-tuning on instruction-and-response pairs.
20. **Integer overflow** — a number too big for its box — a result exceeding the type's range, wrapping in unsigned types and undefined for signed C.
21. **Integration test** — checking that the parts work together — a test exercising several components and their real interfaces.
22. **Interface (network)** — one connection point on a machine — a NIC or virtual device with its own MAC address, addresses and statistics.
23. **Intermediate representation** — the compiler's private middle language — an IR such as LLVM IR, easier to optimize than source or machine code.
24. **Interpreter** — runs source code directly, line by line — a program executing a source or bytecode representation without producing a native binary.
25. **Interrupt** — a device tapping the CPU on the shoulder — an asynchronous signal causing the CPU to jump to a handler and return.
26. **Interrupt latency** — how long before the handler starts — the delay from signal assertion to the first handler instruction.
27. **IOMMU** — an address translator for devices — hardware mapping and restricting device DMA addresses, essential for safe passthrough.
28. **IP address** — the number that identifies a machine on a network — a 32-bit IPv4 or 128-bit IPv6 address, assigned to an interface, not a device.
29. **IPC (CPU)** — instructions finished per clock tick — instructions per cycle, the other half of performance beside frequency.
30. **IPC (software)** — how two programs talk — inter-process communication by pipes, sockets, shared memory or message queues.
31. **IPv4** — the old 32-bit address scheme — RFC 791 addressing, about 4.3 billion addresses, exhausted at IANA in 2011.
32. **IPv6** — the 128-bit replacement — RFC 8200 addressing with 2^{128} addresses, no header checksum and no router fragmentation.
33. **IXP** — a building where networks meet and swap traffic — an internet exchange point with a shared switch fabric for peering.

■ 53.4.10 J

1. **JavaScript** — the language that runs inside web pages — a dynamically typed, event-driven language standardized as ECMAScript.
2. **JIT** — compiling while the program runs — just-in-time compilation of hot paths to native code, guided by runtime profiles.
3. **Jitter** — how much the delay wobbles — variation in packet delay, more damaging than average latency for voice and games.
4. **Job (CI)** — one unit of work in a pipeline — a set of steps run on one runner, with its own environment and result.
5. **Job control** — putting a command in the background — shell management of process groups with foreground, background and stopped states.
6. **Journaling** — writing down the plan before doing it — a filesystem log of intended metadata changes, replayed after a crash.
7. **JSON** — a simple text format for structured data — JavaScript Object Notation: objects, arrays, strings, numbers, booleans and null.
8. **JWT** — a signed token you can read yourself — JSON Web Token, RFC 7519, a base64url header, payload and signature joined by dots.

■ 53.4.11 K

1. **Kernel (OS)** — the part of the system with full power over the machine — the code running in privileged mode, owning scheduling, memory and devices.
2. **Kernel (maths)** — the small window slid over data — a filter matrix applied by convolution across an image or signal.
3. **Kernel mode** — the privileged half of the processor — the state allowing all instructions and address ranges, entered by trap or interrupt.
4. **Kernel panic** — the system giving up — an unrecoverable kernel error that halts the machine rather than risk corruption.
5. **Key (attention)** — what each token offers to be matched against — a projected vector compared with queries to produce attention weights.
6. **Key derivation** — turning a password into a proper key — a KDF such as PBKDF2, scrypt or Argon2 stretching a secret into key material.
7. **Key exchange** — agreeing a secret without posting it — a protocol such as Diffie-Hellman letting both sides derive a shared key.
8. **Keyframe** — a video frame that stands alone — an intra-coded frame decodable without reference to others, the seek point in a stream.
9. **Kill** — telling a process to stop — sending a signal such as SIGTERM to request exit or SIGKILL to force it.
10. **Kilobyte** — a thousand bytes — 1000 bytes in SI usage; 1024 bytes is a kibibyte, KiB.
11. **Knowledge cutoff** — the date a model's reading stops — the point beyond which pretraining data was not collected.
12. **Kubernetes** — the system that keeps containers running — a container orchestrator with a declarative API and reconciling controllers.
13. **KV cache** — remembering earlier keys and values so they are not redone — stored per-layer key and value tensors for tokens already processed.

■ 53.4.12 L

1. **Lag compensation** — the server rewinding time to judge your shot — re-evaluating a hit against the world state the shooter actually saw.
2. **Latch** — a memory circuit that follows while enabled — a level-sensitive storage element, unlike an edge-triggered flip-flop.
3. **Latency** — the waiting time before something starts — the delay between cause and effect, measured end to end and reported as percentiles.
4. **LBA** — numbering blocks straight through — logical block addressing, a linear block number replacing cylinder, head and sector.
5. **LCD** — a screen where a backlight shines through gates — liquid crystal display, twisting crystals to control light per subpixel.
6. **Learning rate** — how big a step training takes — the scale factor on the gradient, the most sensitive hyperparameter.
7. **Least privilege** — give only the rights actually needed — restricting each principal to the minimum permissions for its task.
8. **Lexer** — the part that cuts source code into words — the tokenizer producing a stream of tokens from characters.
9. **LFS** — keeping big files out of git's history — Large File Storage, replacing large files with pointers and storing content separately.
10. **Library** — reusable code you call — a packaged collection of functions linked statically or loaded dynamically.
11. **Linked list** — items each pointing at the next — a sequence of nodes with pointers, cheap to insert and slow to index.
12. **Linker** — joins compiled pieces into one program — the tool resolving symbols, laying out sections and producing an executable or shared object.
13. **Little-endian** — the smallest byte first — byte order storing the least significant byte at the lowest address, as x86 and ARM do.
14. **Load balancer** — spreading requests across servers — a device or service distributing connections by round robin, least connections or hashing.
15. **Loader** — the code that puts a program into memory — the kernel or dynamic loader mapping segments, resolving symbols and jumping to the entry point.
16. **localhost** — the name a machine uses for itself — the hostname resolving to 127.0.0.1 and ::1, never leaving the machine.
17. **Locality** — recently or nearby data is likely used again — temporal and spatial locality, the reason caches work.
18. **Lock** — one at a time, please — a synchronization primitive giving exclusive access to shared state.
19. **LOD** — simpler models for distant objects — level of detail, swapping meshes or textures by screen size to save work.
20. **Log** — the running record of what happened — an append-only stream of events used for debugging, audit and recovery.
21. **Logic gate** — a tiny circuit applying one rule — a combinational circuit implementing a Boolean function of its inputs.
22. **Logit** — the raw score before it becomes a probability — the unnormalized output per vocabulary entry, fed to softmax.
23. **Loopback** — the machine talking to itself — the virtual interface for 127.0.0.0/8 and ::1, never leaving the host.
24. **LoRA** — training a small add-on instead of the whole model — low-rank adaptation, learning two

small matrices whose product updates a frozen weight.

25. **Loss function** — the score that says how wrong the model is — the objective minimized in training, typically cross-entropy for language models.
26. **Lossless** — nothing is thrown away — compression from which the exact original bytes can be restored, as in FLAC, PNG and gzip.
27. **Lossy** — some detail is thrown away for size — compression discarding information judged imperceptible, as in JPEG, MP3 and H.264.
28. **LRU** — throw out what was used longest ago — least recently used, a common cache replacement policy.
29. **LTE** — the fourth generation mobile standard — 3GPP Long Term Evolution, an all-IP radio network using OFDMA downlink.

■ 53.4.13 M

1. **MAC** (crypto) — a tag proving a message was not altered — message authentication code, such as HMAC, computed with a shared secret key.
2. **MAC address** — the hardware name of a network port — a 48-bit link-layer address, the first 24 bits identifying the manufacturer.
3. **Mach-O** — the executable format on macOS — Apple's binary format, with load commands and support for fat binaries holding several architectures.
4. **Magic number** — the few bytes at the start that say what a file is — a fixed signature such as 0x7F ELF or PK for zip, used by the file command.
5. **Malloc** — asking for memory while running — the C library call returning a pointer to an uninitialized block of the requested size.
6. **MBR** — the old partition table in the first sector — master boot record, four primary partitions and a 2 TiB limit.
7. **Memory leak** — memory taken and never given back — allocated memory that stays reachable but unused, growing the process footprint.
8. **Merge** — joining two lines of work — a commit with two or more parents, combining changes since the merge base.
9. **Merge base** — the last commit both branches share — the best common ancestor used as the reference point in a three-way merge.
10. **Merge queue** — testing changes in the order they will land — a forge feature building each pull request against the real future main branch.
11. **mergeStateStatus** — a forge's summary of whether a merge is possible — GitHub's field with values such as CLEAN, BLOCKED, BEHIND, DIRTY or UNKNOWN.
12. **MESI** — the four states a cache line can be in — modified, exclusive, shared, invalid: the coherence protocol keeping cores consistent.
13. **Metadata** — data about the data — attributes such as size, times, type and permissions, held separately from content.
14. **Microkernel** — a tiny kernel with drivers outside it — a design keeping only scheduling, memory and IPC privileged, with services in user space.
15. **Microservice** — a small service that does one job — an independently deployable service with its own data and interface.
16. **Middleware** — code sitting between request and handler — a chain of components processing requests before or after the application.
17. **Migration** (data) — changing the shape of stored data — a versioned script altering schema or contents, applied in order and recorded.

18. **MIMO** — several antennas sending several streams — multiple input multiple output, using spatial paths to multiply capacity.
19. **Mipmap** — smaller pre-made copies of a texture — a chain of half-size texture levels selected by screen footprint to avoid shimmering.
20. **Mixture of experts** — only part of the model runs per token — a layer routing each token to a few expert subnetworks, so active parameters are fewer than total.
21. **MMU** — the hardware that translates addresses — memory management unit, walking page tables to map virtual to physical addresses.
22. **Mode (file)** — the permission bits on a file — the 12-bit field holding owner, group and other permissions plus setuid, setgid and sticky.
23. **Model card** — the label describing what a model is and is not — a document stating intended use, training data summary, evaluations and limitations.
24. **Modulation** — changing a wave so it carries data — varying amplitude, frequency or phase to encode symbols on a carrier.
25. **Monolithic kernel** — one big kernel with drivers inside — a design running services and drivers in kernel mode, as Linux does.
26. **Monorepo** — everything in one repository — a single repository holding many projects, with shared tooling and atomic cross-project changes.
27. **Moore's law** — transistors per chip keep doubling — Gordon Moore's 1965 observation, revised in 1975 to a doubling roughly every two years.
28. **MOSFET** — the switch nearly all chips are made from — metal-oxide-semiconductor field-effect transistor, whose gate voltage forms a channel.
29. **Mount** — attaching a filesystem into the tree — making a device or remote share visible at a directory path.
30. **Mount point** — the directory where a filesystem appears — the path at which a filesystem's root is grafted into the namespace.
31. **MTU** — the biggest packet a link will carry — maximum transmission unit, 1500 bytes on standard Ethernet.
32. **Multicast** — one send, many listeners — delivery to a group address, where the network duplicates packets only where paths diverge.
33. **Multi-head attention** — several attention views at once — running attention in parallel subspaces and concatenating the results.
34. **Multiplexing** — many conversations over one link — interleaving several logical streams on one physical or transport channel.
35. **Mutation testing** — checking that tests would notice a bug — deliberately altering code to see whether the test suite fails.
36. **Mutex** — a lock only one holder may take — mutual exclusion primitive with owner tracking, blocking other threads until released.

■ 53.4.14 N

1. **Namespace** (Linux) — a private view of a system resource — a kernel feature isolating mounts, PIDs, network stacks, users or hostnames per process group.
2. **NaN** — the float that means “not a number” — an IEEE 754 value produced by invalid operations, unequal even to itself.
3. **NAND flash** — the storage in phones and SSDs — non-volatile memory of series-connected cells, written per page and erased per block.
4. **NAT** — many private machines behind one public address — network address translation, rewriting addresses and ports and tracking the mapping.
5. **Netcode** — the part of a game that handles the network — the synchronization design: tick rate, prediction, interpolation and reconciliation.
6. **Netmask** — which part of an address is the network — a bit mask marking the prefix, written as 255.255.255.0 or as /24.
7. **Network interface** — the port a machine uses to reach a network — a physical or virtual device with a MAC address and its own addresses.
8. **Neural network** — layers of weighted sums with bends between them — a differentiable function of stacked linear maps and nonlinearities, fitted by gradient descent.
9. **Next hop** — the router to hand this packet to — the neighbouring address a routing table entry sends matching traffic to.
10. **Nginx** — a common web server and reverse proxy — an event-driven server handling many connections with few processes.
11. **Nibble** — half a byte — four bits, exactly one hexadecimal digit.
12. **NIC** — the network card — network interface controller, handling framing, checksums and often offloads.
13. **Node** (compute) — one machine in a cluster — a host contributing CPU, memory and storage, managed by the cluster’s scheduler.
14. **Node** (process) — the transistor generation name — a marketing label such as 5 nm or 3 nm, no longer matching any physical dimension.
15. **Nonce** — a number used exactly once — a value ensuring freshness or uniqueness, never repeated under one key.
16. **Non-volatile** — it remembers with the power off — storage retaining state without power, unlike DRAM and SRAM.
17. **Normalization** (AI) — keeping numbers in a sensible range — layer or RMS normalization rescaling activations to stabilize training.
18. **Notification** (interrupt) — see interrupt — an asynchronous event delivered to software rather than polled for.
19. **NTP** — how machines agree what time it is — Network Time Protocol, disciplining clocks to within milliseconds over the internet.
20. **Null pointer** — a pointer deliberately pointing at nothing — the reserved address zero, whose dereference traps on most systems.
21. **NUMA** — memory that is closer to some cores than others — non-uniform memory access, where latency depends on which node owns the memory.
22. **NVMe** — the protocol built for flash, not for disks — a command set over PCIe with deep parallel queues and very low overhead.
23. **NVRAM** — small memory that survives a reboot — non-volatile RAM used for firmware settings and boot variables.
24. **Nyquist** — sample at least twice the highest frequency — the sampling theorem requiring a rate

above twice the signal bandwidth to avoid aliasing.

■ 53.4.15 O

1. **OAuth** — letting an app act for you without your password — a delegation framework issuing scoped access tokens, RFC 6749.
2. **Object file** — compiled code not yet joined up — a relocatable binary with sections, symbols and relocation entries.
3. **Observability** — being able to tell what a system is doing — the ability to answer new questions from existing logs, metrics and traces.
4. **OFDM** — splitting a channel into many narrow lanes — orthogonal frequency division multiplexing, used by Wi-Fi, LTE, 5G and DSL.
5. **Ohm's law** — push equals flow times resistance — $V = IR$, relating voltage, current and resistance in a linear element.
6. **OLED** — a screen where each dot makes its own light — organic light-emitting diode panels with no backlight, giving true black.
7. **Opcode** — the number that says which operation — the field of an instruction encoding selecting the operation to perform.
8. **Opcode fusion** — merging two instructions into one operation — a CPU front-end optimization combining common instruction pairs.
9. **Open weights** — the trained numbers are published — a model whose parameters are downloadable, which is not the same as open training data.
10. **Orchestration** — deciding what runs where — automated placement, scaling and recovery of workloads across a cluster.
11. **Origin** — the default name for where you cloned from — the conventional remote name git creates for the source repository.
12. **Origin server** — the machine holding the real content — the source a CDN fetches from when it has no valid cached copy.
13. **Orphan** (git) — an object nothing points to — an unreachable object, removable by garbage collection after its grace period.
14. **Oscillator** — the circuit that makes the ticking — a crystal or ring circuit producing a periodic reference signal.
15. **OSI model** — seven named layers for teaching — ISO 7498, from physical up to application, useful as vocabulary, not as an implementation.
16. **Out-of-order** — doing ready work first, then tidying up — executing instructions as operands become available, retiring them in program order.
17. **Overclocking** — running a chip faster than rated — raising frequency and often voltage beyond specification, at the cost of heat and stability.
18. **Overfitting** — memorizing the answers instead of learning — fitting noise in training data, shown by training loss falling while validation loss rises.
19. **Overlay network** — a network built on top of another — virtual links tunnelled over an existing network, as with VXLAN or WireGuard.
20. **Overprovisioning** — spare flash the drive keeps hidden — reserved capacity that improves garbage collection and endurance.

■ 53.4.16 P

1. **PaaS** — you bring the code, they run everything else — platform as a service, managing runtime, scaling and patching for you.
2. **Packaging** (chips) — putting the die in something you can solder — attaching a die to a substrate with bumps or wires and encapsulating it.
3. **Packet** — one labelled chunk of data — a network-layer unit of header plus payload, routed independently.
4. **Packet loss** — a packet that never arrives — discard caused by congestion, errors or policy, inferred from missing acknowledgements.
5. **Packfile** — many git objects compressed into one file — a container storing objects with delta compression, plus an index for lookup.
6. **Page** — the unit memory is managed in — a fixed-size block of virtual memory, usually 4 KiB, mapped to a physical frame.
7. **Page fault** — the page was not in memory — a trap when a virtual page has no valid mapping, handled by loading or allocating.
8. **Page table** — the map from virtual to physical pages — a multi-level structure walked by the MMU, cached in the TLB.
9. **Parameter** (AI) — one learned number in the model — a single weight or bias updated by training and stored in the model file.
10. **Parity** — one extra bit to spot an odd error — a bit making the count of ones even or odd, detecting single-bit errors.
11. **Parser** — the part that works out the structure — the component turning tokens into a syntax tree by grammar rules.
12. **Partition** — one labelled region of a drive — a contiguous block range described by an MBR or GPT entry.
13. **Pass@k** — did any of k attempts pass — a benchmark metric giving credit if at least one of k samples is correct.
14. **Passphrase** — a long password made of words — a secret chosen for entropy through length rather than symbol variety.
15. **Patch** — a file describing a change — a unified diff with context lines, applicable to matching source.
16. **PATH** — the list of folders the shell searches — an environment variable of directories scanned in order for a command name.
17. **Payload** — the actual content, minus the wrapping — the data a protocol carries, excluding headers and trailers.
18. **PCIe** — the fast lanes connecting chips inside a computer — PCI Express, point-to-point serial lanes; gen 5 runs 32 GT/s per lane.
19. **PCM** — sound stored as plain numbers — pulse-code modulation, uncompressed samples at a fixed rate and bit depth.
20. **PE** — the executable format on Windows — Portable Executable, with a DOS stub, COFF header and sections such as .text and .rdata.
21. **Peering** — two networks exchanging traffic directly — a bilateral agreement to swap customer routes, often settlement-free.
22. **Permission** — what you are allowed to do to a file — access bits or ACL entries checked by the kernel on open.
23. **Persistent storage** — data that outlives the process — storage surviving restarts, unlike memory or a container's writable layer.

24. **Phase** — where a wave is in its cycle — angular offset in degrees or radians, the basis of phase-shift keying.
25. **Photolithography** — printing the circuit pattern with light — projecting a mask image onto photoresist to define features on a wafer.
26. **Photoresist** — the light-sensitive coating on a wafer — a polymer whose solubility changes on exposure, defining where etching happens.
27. **Piconet** — a small group of Bluetooth devices working together — one primary device and up to seven active secondaries sharing a hopping pattern.
28. **Ping** — asking a host whether it is there — sending ICMP echo requests and timing the replies.
29. **Pipe** — the output of one command becoming the input of the next — a kernel buffer joining one process's stdout to another's stdin.
30. **Pipeline (CPU)** — instructions in a production line — overlapping instruction stages so one finishes per cycle in the ideal case.
31. **Pixel** — one dot of the picture — the smallest addressable element of a raster image or display.
32. **Plist** — the settings file format on Apple systems — a property list, in XML or binary form, holding key-value configuration.
33. **PN junction** — where p-type meets n-type silicon — the boundary forming a depletion region and conducting in only one direction.
34. **Pod** — the smallest deployable unit in Kubernetes — one or more containers sharing a network namespace, address and storage volumes.
35. **Pointer** — a value that holds an address — a variable containing the memory address of another object.
36. **Polling** — asking repeatedly instead of being told — checking a status in a loop rather than waiting for an interrupt or event.
37. **Port** — the number identifying which program on a machine — a 16-bit endpoint identifier in TCP and UDP headers.
38. **Positional encoding** — telling the model where each word sits — added or rotated information giving order to otherwise position-blind attention.
39. **POSIX** — the portable Unix rulebook — IEEE 1003 standards defining system calls, shell and utilities.
40. **PPI** — how tightly packed the dots are — pixels per inch of a display, about 460 on current high-end phones.
41. **Precision** — how finely a number can be written — the count of significant bits or digits a format can represent.
42. **Prefetch** — fetching before it is asked for — hardware or software loading data into cache ahead of predicted use.
43. **Prefill** — reading the whole prompt in one go — the parallel forward pass over input tokens that fills the KV cache, compute bound.
44. **Pretraining** — the long first training on huge text — self-supervised next-token training that produces a base model.
45. **Primary key** — the column that identifies a row — a unique, non-null identifier for a table's rows.
46. **Privilege escalation** — getting more power than you were given — an exploit raising a principal's permissions beyond its intended level.
47. **Process** — one running program with its own memory — an address space, one or more threads, and a table of open file descriptors.
48. **Profiling** — measuring where the time actually goes — sampling or instrumenting a program to attribute cost to code paths.

- 49. **Prompt injection** — hidden text telling a model to misbehave — untrusted content in a model's context that overrides intended instructions.
- 50. **Propagation delay** — the time a gate takes to change its output — the interval from input transition to a valid output, tens of picoseconds today.
- 51. **Protocol** — the agreed rules for a conversation — the message formats, ordering and state machine two parties follow.
- 52. **Proxy** — a middleman making requests for you — a server that terminates one connection and opens another on the client's behalf.
- 53. **PTY** — a fake terminal for a program that expects one — pseudo-terminal, a master and slave pair providing terminal semantics without hardware.
- 54. **Pull request** — asking for your branch to be merged — a forge object linking a source branch, target branch, review and checks.
- 55. **Push** — sending your commits to a remote — uploading objects and asking the remote to update refs, refused if it is not a fast-forward.
- 56. **PWM** — a switch flicked fast to fake a level — pulse-width modulation, varying duty cycle to control average power.

■ 53.4.17 Q

- 1. **QAM** — packing more bits into each radio symbol — quadrature amplitude modulation; 1024-QAM carries 10 bits per symbol.
- 2. **Quantization** — using coarser numbers to save space — reducing weight precision, for example FP16 to INT8 or 4-bit, with a small accuracy cost.
- 3. **Query** (attention) — what one token is looking for — a projected vector compared against every key to produce attention weights.
- 4. **Queue** — a line of waiting work — a FIFO structure decoupling producers from consumers.
- 5. **QUIC** — a fast transport built on UDP — RFC 9000, giving streams, encryption and connection migration without head-of-line blocking.
- 6. **Quorum** — enough members agreeing to decide — a majority of replicas required for a consistent read or write.
- 7. **Quota** — a cap on how much you may use — an enforced limit on storage, requests or resources per account or user.

■ 53.4.18 R

- 1. **Race condition** — the answer depends on who gets there first — a bug where behaviour depends on the unsynchronized timing of concurrent access.
- 2. **RAG** — letting a model read documents before answering — retrieval-augmented generation, injecting retrieved passages into the prompt.
- 3. **RAID** — several drives acting as one, for speed or safety — striping, mirroring or parity across disks; RAID is not a backup.
- 4. **RAM** — the fast memory that forgets when power stops — volatile random-access memory holding running code and data.
- 5. **Rebase** — replaying your commits on a new base — creating new commits with the same changes but new parents and new hashes.
- 6. **Refactoring** — tidying code without changing behaviour — a behaviour-preserving transformation improving structure or clarity.
- 7. **Reflog** — git's private diary of where refs pointed — a local log of ref updates, letting you recover from a bad reset.

8. **Refresh rate** — how often the screen redraws — display updates per second in hertz, commonly 60, 120 or 144.
9. **Register** — the CPU's own handful of tiny stores — a named storage location inside the core, accessed in a fraction of a cycle.
10. **Registry** (containers) — the shop images are pulled from — a service storing and serving container images by name and digest.
11. **Registry** (Windows) — the central settings database — a hierarchical store of keys and values used by Windows and its applications.
12. **Regularization** — stopping the model memorizing — techniques such as weight decay, dropout or early stopping that limit overfitting.
13. **REJECT** — refusing a packet and saying so — a firewall action sending an ICMP unreachable or TCP RST, so the sender fails fast.
14. **Remote** — a named repository somewhere else — a stored URL plus fetch and push refsspecs, such as origin.
15. **Remote-tracking ref** — your last known idea of a remote branch — a local ref under refs/remotes updated only by fetch or push.
16. **Replication** — keeping copies on more than one machine — maintaining synchronized data on several nodes for durability or read scaling.
17. **Repository** — the whole project with all its history — a directory containing the object database, refs and configuration.
18. **Resistance** — how much a material fights the flow — the ratio of voltage to current, in ohms, dissipating power as heat.
19. **Resolution** — how many dots the picture has — the pixel count of an image or display, such as 1920 by 1080.
20. **Resolver** — the helper that does DNS lookups for you — a recursive server querying the hierarchy and caching results by TTL.
21. **REST** — using plain HTTP methods on named resources — an architectural style with stateless requests and uniform verbs over URLs.
22. **Reverse proxy** — a middleman in front of the servers — a server accepting client requests and forwarding them to backends.
23. **RFC 1918** — the address ranges reserved for private use — 10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16, never routed on the internet.
24. **Ring buffer** — a fixed queue that wraps around — a circular array with head and tail indices, overwriting or blocking when full.
25. **RLHF** — training a model on what people preferred — reinforcement learning from human feedback, optimizing against a learned reward model.
26. **Root server** — the servers at the top of DNS — the 13 named root identities serving the root zone, reached by anycast.
27. **Rootkit** — malware that hides itself in the system — code subverting the kernel or firmware to conceal its own presence.
28. **RoPE** — encoding position by rotating the vectors — rotary position embedding, applying a position-dependent rotation to queries and keys.
29. **Round-trip time** — how long there and back takes — the delay from sending to receiving the acknowledgement, driving TCP behaviour.
30. **Router** — the box that forwards between networks — a device choosing an outgoing interface by longest-prefix match on the destination.
31. **Routing table** — the list of where to send what — the forwarding entries of prefix, next hop,

interface and metric.

32. **RSA** — an old public-key algorithm based on big numbers — a scheme whose security rests on the difficulty of factoring; keys are 2048 bits or larger.
33. **RST** — a rude end to a TCP connection — a reset flag telling the peer to abandon the connection immediately.
34. **Runner** — the machine that executes a CI job — an agent that claims jobs from a queue and reports results back.

■ 53.4.19 S

1. **SaaS** — you just use the website — software as a service, run entirely by the provider with no infrastructure for you.
2. **Salt** — a random extra ingredient in a password hash — a unique per-password value stored alongside it, defeating precomputed tables.
3. **Same-origin policy** — one site cannot read another's data — the browser rule isolating documents by scheme, host and port.
4. **Sampling** (AI) — choosing the next word from the odds — drawing from the model's output distribution, shaped by temperature and top-p.
5. **Sampling rate** — how many measurements per second — samples per second; CD audio uses 44,100 Hz.
6. **Sandbox** — a restricted area a program cannot escape — an isolation mechanism limiting a process's syscalls and resources.
7. **SATA** — the older cable standard for drives — Serial ATA, 6 Gbit/s in revision 3, with one command queue.
8. **Scaling laws** — bigger models and more data give predictable gains — empirical power-law relations between compute, data, parameters and loss.
9. **Scheduler** — decides which task runs next — the kernel component choosing runnable threads by policy and priority.
10. **Scope** (OAuth) — the exact list of things a token may do — a string set attached to a token limiting the operations it authorizes.
11. **SDK** — the tools you need to build for a platform — a software development kit of libraries, headers, tools and samples.
12. **Sector** — the smallest addressable piece of a disk — 512 bytes traditionally, 4096 bytes on Advanced Format drives.
13. **Segment** — TCP's unit of data — the transport-layer unit with sequence numbers, flags and payload.
14. **Semantic versioning** — the version number tells you what changed — MAJOR.MINOR.PATCH, where a major bump means a breaking change.
15. **Semaphore** — a counter that limits how many may proceed — a synchronization primitive with wait and signal operations.
16. **Semiconductor** — a material between a conductor and an insulator — a solid such as silicon whose conductivity is controlled by doping, field or light.
17. **Serialization** — turning objects into bytes to send or store — encoding structures into a byte stream, reversed by deserialization.
18. **Serverless** — you write the function, not the server — a model where the platform runs code on demand and bills by invocation and duration.
19. **Session** — the memory of who you are between requests — server or token state associating requests with an authenticated user.

20. **SHA-256** — a common cryptographic fingerprint — a hash producing a 256-bit digest, used in TLS, git and signing.
21. **Shader** — a small program the GPU runs per vertex or pixel — GPU code for stages such as vertex, fragment or compute.
22. **Sharding** — splitting data across several databases — partitioning by key so each shard holds a subset of the rows.
23. **Shebang** — the first line saying which interpreter to use — the two bytes `#!` plus a path, read by the kernel when running a script.
24. **Shell** — the program that reads your commands — a command interpreter handling expansion, redirection, pipelines and job control.
25. **Signal** (Unix) — a short message telling a process something happened — an asynchronous notification such as SIGINT, SIGTERM or SIGSEGV.
26. **Signal-to-noise ratio** — how loud the message is compared with the hiss — the power ratio of signal to noise, usually in decibels.
27. **SIM** — the card that identifies you to the mobile network — a smart card holding the IMSI and authentication key, or an eSIM profile.
28. **SIMD** — one instruction working on many numbers — single instruction multiple data, as in SSE, AVX and NEON.
29. **Single point of failure** — the one part whose loss stops everything — a component with no redundancy on the critical path.
30. **SLC to QLC** — how many bits are squeezed into one flash cell — single, multi, triple and quad-level cells, trading endurance and speed for density.
31. **Slice** — a view into part of an array — a reference with pointer, length and sometimes capacity, without copying.
32. **SLO** — the promise you hold yourself to — a service level objective, a target for a measured indicator over a window.
33. **SMTP** — how mail servers hand messages to each other — Simple Mail Transfer Protocol, RFC 5321, on ports 25, 465 or 587.
34. **Snapshot** — a frozen picture of data at one moment — a point-in-time copy, often made cheaply by copy-on-write.
35. **Socket** — the endpoint a program reads and writes a connection through — an OS object identified by address, port and protocol.
36. **Softmax** — turning scores into probabilities — exponentiating logits and dividing by their sum so the values total one.
37. **Spinlock** — waiting by looping instead of sleeping — a lock that busy-waits, correct only for very short critical sections.
38. **Spread spectrum** — deliberately smearing a signal over more bandwidth — modulating over a wide band to resist interference and interception.
39. **Sprite** — a small image drawn over the scene — a 2D bitmap composited into a frame, historically with dedicated hardware.
40. **SQL injection** — smuggling commands inside data — an attack where unescaped input becomes part of a database query.
41. **SRAM** — fast memory built from logic, used in caches — static RAM using six transistors per bit, needing no refresh.
42. **SSD** — storage with no moving parts — a solid-state drive of NAND flash plus a controller, with microsecond latency.
43. **SSH** — a secure way to log in to another machine — the Secure Shell protocol on port 22, with

host keys and public-key user authentication.

44. **SSID** — the name of a wireless network — the human-readable network identifier broadcast in beacons.
45. **SSRF** — tricking a server into fetching an internal URL — server-side request forgery, reaching private services from a trusted position.
46. **Stack** — the memory used for calls and locals — a LIFO region holding frames with return addresses, arguments and locals.
47. **Staging area** — what will go into the next commit — git's index, holding the exact content to be committed.
48. **Stateful firewall** — it remembers connections, not just packets — a filter tracking flow state so replies are allowed automatically.
49. **Static linking** — the library is baked into the binary — copying library code into the executable at build time.
50. **Streaming** — playing while it is still arriving — delivering media progressively rather than downloading it whole.
51. **Subnet** — a slice of an address range — a network defined by a prefix, sharing one broadcast domain in IPv4.
52. **Sub-pixel** — the red, green or blue part of a dot — one colour element of a pixel, exploitable for text rendering.
53. **Superscalar** — issuing several instructions each cycle — a core with multiple parallel execution ports and wide issue.
54. **Swap** — using disk when memory runs out — backing anonymous pages with storage so more can be committed than exists.
55. **Symbolic link** — a file that points at another path — a special file holding a path string, resolved at open time.
56. **System call** — a program asking the kernel to do something — the trap instruction and ABI by which user code enters the kernel.
57. **systemd** — the thing that starts everything on modern Linux — an init system and service manager with units, sockets and journal.

■ 53.4.20 T

1. **Tag** — a fixed name for one commit — a ref under refs/tags, lightweight or an annotated object with message and signature.
2. **TCP** — the protocol that makes sure everything arrives in order — a connection-oriented transport with sequence numbers, acknowledgements and retransmission.
3. **Tearing** — the picture split across two frames — a display artefact when the buffer swaps mid-scanout, fixed by vsync.
4. **Temperature** — how adventurous the model's word choice is — a divisor on logits before softmax; lower is more deterministic.
5. **Thread** — one line of execution inside a program — a schedulable flow with its own stack and registers, sharing the process address space.
6. **Threat model** — deciding who you are defending against — an explicit statement of assets, adversaries, capabilities and out-of-scope risks.
7. **Three-way handshake** — the greeting that opens a TCP connection — the SYN, SYN-ACK, ACK exchange establishing sequence numbers.
8. **Throughput** — how much work gets done per second — completed operations or bits per unit time, distinct from latency.

9. **Tick rate** — how often the game server updates the world — server simulation steps per second, often 20 to 128 in shooters.
10. **TLB** — a small cache of address translations — translation lookaside buffer, holding recent virtual-to-physical page mappings.
11. **TLD** — the last label of a domain name — the top-level domain such as com, org or in, delegated from the root zone.
12. **TLS** — the encryption under HTTPS — Transport Layer Security; version 1.3, RFC 8446, needs one round trip and requires forward secrecy.
13. **Token** — the small piece of text a model actually sees — a subword unit from the tokenizer's vocabulary, roughly 0.75 English words each.
14. **Tokenizer** — the tool that cuts text into tokens — the encoder mapping text to integer IDs, usually by byte-pair encoding.
15. **Top-p** — keep only the most likely words that add up to p — nucleus sampling, restricting choice to the smallest set whose probability exceeds p.
16. **Trace** (observability) — the story of one request across services — a tree of timed spans sharing one trace identifier.
17. **Traceroute** — showing the routers along the way — sending packets with increasing TTL and reading the ICMP time-exceeded replies.
18. **Trailing whitespace** — invisible spaces at the end of a line — characters that break diffs and comparisons without being visible.
19. **Transaction** — all of it or none of it — a group of operations with atomicity, consistency, isolation and durability.
20. **Transformer** — the design behind modern language models — a stack of self-attention and feed-forward blocks, from the 2017 paper Attention Is All You Need.
21. **Transistor** — an electrical tap — a semiconductor switch or amplifier; in CMOS logic, a field-effect transistor.
22. **Transpiler** — a compiler from one language to another readable one — a source-to-source translator, as TypeScript to JavaScript.
23. **Trap** — a deliberate jump into the kernel — a synchronous exception raised by an instruction, such as a system call.
24. **Trilateration** — finding a position from several distances — computing a location from ranges to known points, as GPS does.
25. **TRIM** — telling the drive which blocks are now rubbish — a command letting the filesystem mark blocks free so the SSD can erase them early.
26. **Truncation** — cutting data that does not fit — discarding bits, characters or tokens beyond a limit, often silently.
27. **Truth table** — every input combination and its output — an exhaustive listing of a Boolean function's behaviour.
28. **TTL (IP)** — a hop counter that stops loops — a header field decremented per router, triggering ICMP time exceeded at zero.
29. **TTL (DNS)** — how long an answer may be cached — the seconds a resolver may reuse a record before asking again.
30. **TTY** — the terminal a program is attached to — a kernel device providing line discipline, echo and job control.
31. **Tunnel** — one network carried inside another — encapsulating packets inside another protocol, as with VPNs and GRE.

32. **Two-factor authentication** — a password plus something else — requiring a second independent factor such as a code or hardware key.
33. **Two's complement** — how computers store negative numbers — inverting all bits and adding one, giving one zero and a range from $-2^{(n-1)}$ to $2^{(n-1)}-1$.
34. **Type system** — rules about what values may be combined — a static or dynamic discipline classifying values and rejecting invalid operations.

■ 53.4.21 U

1. **UDP** — send it and hope — a connectionless transport with no ordering, no retransmission and very little overhead.
2. **UEFI** — the modern replacement for the BIOS — firmware with its own boot manager, GPT support, drivers and optional Secure Boot.
3. **Unicode** — one number for every character in every script — a character set of code points from U+0000 to U+10FFFF, with 154,998 assigned in version 16.0.
4. **Unit test** — checking one small piece on its own — a fast, isolated test of a single function or class.
5. **Unreachable** — nothing can get there from here — an object with no path from a root, or a network with no route.
6. **Upstream** — the branch or project yours follows — the configured remote branch a local branch compares and pushes to.
7. **Uptime** — how long since the last restart — elapsed time since boot, not the same as availability.
8. **USB** — the connector that replaced all the others — Universal Serial Bus, from 12 Mbit/s in 1996 to 80 Gbit/s in USB4 Version 2.0.
9. **User mode** — the restricted half of the processor — the unprivileged state where a program cannot touch devices or kernel memory directly.
10. **User space** — everything outside the kernel — processes running unprivileged, reaching the kernel only by system calls.
11. **UTF-8** — Unicode written as bytes, ASCII still working — a variable-length encoding using one to four bytes per code point.
12. **UUID** — a name unique without asking anyone — a 128-bit identifier, usually version 4 random or version 7 time-ordered.

■ 53.4.22 V

1. **Value** (attention) — the content a token contributes when attended to — a projected vector summed with attention weights to form the output.
2. **Vector database** — storage that finds things by meaning — an index over embeddings supporting approximate nearest-neighbour search.
3. **Version control** — keeping every version and who changed what — a system recording snapshots, authorship and lineage of files.
4. **Vertex** — one corner of a shape — a point with position and attributes, processed by the vertex stage.
5. **Virtual machine** — a whole fake computer inside a real one — an isolated guest with its own kernel, scheduled by a hypervisor.
6. **Virtual memory** — every program thinks it owns the whole address space — per-process address translation through page tables, with protection and paging.
7. **Virtualization** — one machine pretending to be several — hardware-assisted isolation of guests, using extensions such as VT-x or AMD-V.
8. **VLAN** — several separate networks over one set of wires — IEEE 802.1Q tagging, giving separate broadcast domains on shared links.

9. **Voice coil** — the coil that moves a speaker cone — a wire coil in a magnetic gap, driven by current to move the diaphragm.
10. **Volatile** (memory) — it forgets when power stops — storage requiring power to retain state, as DRAM and SRAM do.
11. **Voltage** — electrical push — potential difference in volts, the work per unit charge between two points.
12. **Volume** — storage attached to a container or machine — a named block or filesystem mount whose lifetime is separate from the workload.
13. **VPN** — a private tunnel across a public network — an encrypted tunnel giving remote hosts an address inside a protected network.
14. **Vsync** — waiting for the screen before swapping the picture — synchronizing buffer swaps to the display refresh to avoid tearing.
15. **Vulnerability** — a weakness someone can exploit — a flaw allowing an attacker to violate a security property, often tracked by CVE number.

■ 53.4.23 W

1. **Wafer** — the thin silicon disc chips are made on — a 300 mm polished slice cut from a boule, carrying hundreds of dies.
2. **Watchdog** — a timer that reboots a stuck system — hardware or software resetting the system if not periodically reset by software.
3. **Wear levelling** — spreading writes so no cell dies early — an SSD controller policy distributing program and erase cycles across blocks.
4. **Webhook** — the server calls you when something happens — an HTTP POST sent by a service to a URL you registered.
5. **WebSocket** — a two-way channel over one web connection — RFC 6455, upgrading an HTTP connection to a persistent frame-based channel.
6. **Weight** — one number that scales an input — a learned multiplier in a layer's matrix, the bulk of a model's parameters.
7. **Wi-Fi** — wireless local networking — IEEE 802.11 standards, branded by generation, currently up to Wi-Fi 7 from 802.11be.
8. **Word** (CPU) — the natural size the machine works in — the register and native operand width, 64 bits on current mainstream CPUs.
9. **Workflow** — the file describing an automated pipeline — a declarative definition of triggers, jobs and steps for a CI system.
10. **Working set** — the pages a program actually needs right now — the memory touched over a recent interval, the amount that must stay resident.
11. **Working tree** — the actual files you can see and edit — the checked-out files in your directory, separate from the index and object store.
12. **WPA2** — the Wi-Fi encryption most networks still use — 802.11i security with AES-CCMP and a four-way handshake.
13. **WPA3** — the newer Wi-Fi encryption — security using SAE key exchange, resisting offline dictionary attacks on the passphrase.
14. **Write amplification** — the drive writing more than you asked — the ratio of physical NAND writes to host writes, driven by garbage collection.

■ 53.4.24 X

1. **x86** — the instruction set in most PCs — Intel's CISC architecture from the 1978 8086, extended to 64 bits as x86-64 in 2003.

2. **XDR** — the widest brightness range a display claims — extreme dynamic range, a marketing name for HDR panels with high peak luminance.
3. **XML** — structured text with tags — a markup language for machine-readable data, standardized in 1998.
4. **XOR** — true when the inputs differ — the exclusive-or Boolean function, the core of parity, adders and simple ciphers.
5. **XSS** — injecting script into someone else's page — cross-site scripting, where untrusted input is rendered as executable script.

■ 53.4.25 Y

1. **Y2K** — the year 2000 date problem — the failure risk from two-digit year fields, largely averted by remediation before 2000.
2. **YAML** — an indentation-based configuration format — a superset of JSON where structure comes from indentation, common in CI and Kubernetes.
3. **Yield** — the share of chips that come out working — the fraction of dies on a wafer passing test, the main driver of chip cost.

■ 53.4.26 Z

1. **Zero trust** — trust nothing just because it is inside the network — an architecture authenticating and authorizing every request regardless of network position.
2. **Zero-day** — an attack for which no fix exists yet — a vulnerability exploited before the vendor has released a patch.
3. **Zombie process** — a finished process nobody has collected — a terminated child whose exit status has not yet been reaped by its parent.
4. **Zone** (DNS) — one slice of the name tree under one authority — a portion of the namespace served by one set of authoritative servers.

53.5 Where to go next

1. Understanding comes from doing. You did not learn to ride a bicycle by reading about balance, and this is the same.
2. Pick the one layer in this book that annoyed you most because you did not quite get it. Go there first. Discomfort is a good compass.
3. Build the smallest possible version of something you used today. A shell that runs one command. A web server that returns one line. A tiny assembler.
4. Then break it on purpose. Unplug the network mid-transfer. Fill the disk. Send a malformed header. You learn more from the failure than the success.
5. Read real source code, not just tutorials. Start with something small and old: a `coreutils` command, a toy compiler, an early version of a database.
6. Read the actual specifications. RFCs are shorter and clearer than their reputation. So is the IEEE 754 summary. So is the git object format.
7. Learn one tool that shows you the truth: a packet capture tool, a debugger, a profiler, or `strace`. One of these changes how you think permanently.
8. When something breaks, write down what you observed before you write down what you think it means. Keep those two columns separate. That is diagnosis.
9. Do not chase every new framework. The layers under them barely change. The person who knows the layers can pick up any framework in a week.

10. Teach it to someone. If you cannot explain it plainly, you have not finished learning it. That is the whole reason this book has a plain half.
11. Be sceptical of benchmark numbers, including ones you like. Ask what was measured, on what, by whom, and what was left out.
12. It is normal to forget most of the details. Keep this chapter. Remember the shape of the thing, and look the numbers up when you need them.
13. Nobody understands all of this at once, including the people who built it. Every expert you admire has a large map and a few deep holes.
14. You started with charge in a wire and finished with a language model. That is one continuous story, and you now know it end to end. Go and build.

53.6 The whole book in 100 lines

1. Everything starts with charge, a property of matter that pushes and pulls.
2. Voltage is push, current is flow, resistance is how hard the path fights back.
3. Ohm's law ties them together: voltage equals current times resistance.
4. Silicon is neither a good conductor nor a good insulator, and that is why it is useful.
5. Add tiny amounts of other elements and you get n-type and p-type silicon.
6. Join them and you get a junction that passes current in only one direction.
7. Add a third terminal and a small voltage controls a large flow. That is a transistor.
8. A transistor used as a switch is the only real trick in the whole machine.
9. Sand becomes a pure crystal, the crystal becomes a wafer, the wafer becomes chips.
10. Light prints the pattern, chemicals cut it, and the layers are built up in turn.
11. Not every die works, so yield quietly decides the price of everything.
12. Wire a few transistors one way and you get a gate: AND, OR, NOT, NAND.
13. NAND alone can build every other gate, and therefore every digital circuit.
14. Gates wired in a pattern add binary numbers without knowing what addition is.
15. Feed a gate's output back into its input and it holds a value. That is memory.
16. A clock decides when things may change, so signals have time to settle first.
17. Six transistors hold one bit quickly. That is SRAM, and it becomes cache.
18. One transistor and one capacitor hold one bit cheaply. That is DRAM, and it forgets.
19. Charge trapped on an isolated gate holds a bit with no power. That is flash.
20. Nothing in any of this knows what a number means.
21. Meaning is a convention. We agree that a pattern of bits stands for something.
22. Binary counts in twos. Hexadecimal writes four bits per digit for human eyes.
23. Negative numbers use two's complement, so one adder handles both signs.
24. Fractions use floating point, standardized as IEEE 754 in 1985, and it rounds.
25. Text is numbers too. ASCII gave 128 characters. Unicode gives over a million.
26. UTF-8 stores Unicode as bytes and leaves old ASCII files working unchanged.
27. Pictures are grids of pixels, and each pixel is three numbers for brightness.
28. A framebuffer holds those numbers, and the screen scans them out many times a second.
29. Sound is air pressure measured thousands of times a second and stored as numbers.
30. A converter turns those numbers back into a voltage that moves a coil and a cone.
31. A CPU fetches an instruction, decodes it, executes it, and writes the result.
32. It does this in a pipeline, so several instructions are in progress at once.
33. It guesses which way each branch will go, and it is usually right.
34. It runs work out of order and finishes it in order, to hide the waiting.
35. Memory is far slower than the core, so caches keep recent data close by.

36. A cache line is 64 bytes, and code that respects that runs several times faster.
37. Programs use virtual addresses, and hardware translates them through page tables.
38. That gives every process its own world and stops one program reading another's.
39. Storage is not memory. Storage keeps things when the power stops.
40. A hard disk seeks in milliseconds. A solid-state drive answers in microseconds.
41. Flash erases in big blocks, so drives shuffle data and hide where it really is.
42. Filesystems turn a flat array of blocks into names, folders and permissions.
43. Buses carry data between chips, and inside a computer the fast one is PCIe.
44. Firmware runs before any operating system and knows how to wake the machine.
45. Software is a pattern of bits held in the same memory as data. That is the big idea.
46. A compiler reads source text, works out what it means, and writes machine code.
47. It tokenizes, parses into a tree, checks meaning, optimizes, then emits code.
48. An interpreter skips the output file and carries out the meaning directly.
49. A linker joins the compiled pieces and a loader maps them into memory to run.
50. Languages are tools with opinions, each built for the problem of its own time.
51. An operating system shares one machine between many programs, safely.
52. The kernel is privileged. Your code is restricted and asks by system call.
53. A scheduler decides who runs next, thousands of times every second.
54. Drivers translate general requests into the exact language of one device.
55. The shell is only a program that reads your words and starts other programs.
56. Pipes join those programs so one program's output becomes another's input.
57. A file extension is a hint, not a fact. The bytes at the start are the fact.
58. An executable is a file with a header telling the loader how to start it.
59. Pressing a key sends a code that becomes an event that becomes a character.
60. A font renderer turns that character into pixels, composited into one frame.
61. A GPU exists because pictures need the same small sum done millions of times.
62. Thousands of simple lanes beat a few clever cores when the work is parallel.
63. A network is machines agreeing how to wrap, address and forward data.
64. Data is cut into packets so one big transfer cannot block everyone else.
65. Each layer adds a header and treats the layer above it as opaque payload.
66. An IP address names an interface. A MAC address names a port on one link.
67. Private addresses are reused everywhere, and NAT hides many machines behind one.
68. Routers forward by longest prefix match and know nothing of your conversation.
69. BGP is how independent networks tell each other which prefixes they can reach.
70. DNS turns names into addresses through a hierarchy of caches and authorities.
71. TCP adds sequence numbers, acknowledgements and retransmission to make a stream.
72. UDP adds almost nothing, which is exactly what real-time traffic wants.
73. Traceroute shows the path by sending packets that expire one hop further each time.
74. Silence is evidence. A timeout with no reply tells you what a refusal does not.
75. Diagnosis is separating what an observation proves from what it merely suggests.
76. Radio carries the same bits on a wave, shaped by modulation and shared by turns.
77. TLS wraps a connection in encryption, and a certificate chain proves who answered.
78. A firewall drops or rejects, a VPN tunnels, and a CDN moves content closer to you.
79. The web is documents fetched by HTTP, marked up in HTML, styled by CSS, moved by JavaScript.
80. An API is a promise about requests and answers, so machines can call machines.
81. Git stores snapshots, not differences, and names every object by its content hash.
82. A commit points at a tree and at its parents, and cannot change without changing its name.

83. Branches are only moving labels, which is why making one costs nothing.
84. Because the whole history is local, you can work fully with no network at all.
85. Merge joins two histories. Rebase rewrites yours onto a new base, with new hashes.
86. A forge adds accounts, review, permissions and automation on top of plain git.
87. Continuous integration runs your tests on a machine you do not own, against one exact commit.
88. Cloud is renting someone else's computers, with the hard parts already solved.
89. Containers isolate processes. Virtual machines isolate whole kernels.
90. Security is a threat model, least privilege, and honesty about what you cannot stop.
91. A game is a loop: read input, advance the world by a fixed step, draw, repeat.
92. Everything in that loop must fit the frame budget, or the player feels it.
93. Across a network the client guesses ahead, and the server has the final say.
94. Machine learning replaces rules you write with numbers fitted to examples.
95. A network of weighted sums and bends is trained by gradient descent on a loss.
96. Backpropagation works out which weight caused how much of the error.
97. A transformer lets every token look at every other token, all at once.
98. A language model only predicts the next token. Everything else follows from that.
99. It has no memory beyond its context, no access to truth, and no intent of its own.
100. From charge in a wire to a conversation with a machine, it is one continuous story, and you have now read it.

ABOUT THE AUTHOR

Shikhar Singh

Founder & Chairman

KedByte Technologies Private Limited

Shikhar Singh is the Founder & Chairman of KedByte Technologies Private Limited.

He wrote this book to solve a problem he kept meeting: capable people concluding they were not technical, when in truth nobody had ever shown them the bottom of the stack. His conviction is that computing has no genuinely hard idea in it, only a very long chain of simple ones, and that anybody who is walked along that chain in order can hold the whole of it in their head.

That conviction shapes the method used here. Explain it plainly, with a comparison and a worked example, so that it lands. Then explain it again in the exact professional vocabulary, with the real units and the real figures, so that it is usable. Say clearly where the plain version stops being true. Never let a reader carry away a confident misunderstanding.

Under his direction, KedByte Technologies Private Limited treats technical education as infrastructure rather than as a product feature: knowledge that should be transferable, verifiable, and free of the vocabulary barriers that keep capable people out of the field.

ABOUT THE PUBLISHER

KedByte Technologies Private Limited

KedByte Technologies Private Limited publishes technical work under the KedByte Press imprint.

The editorial standard applied to this volume is simple to state and difficult to meet. Every explanation must work twice: once for a reader who has never opened a terminal, and once for an engineer who needs the exact figure. Every claim that can be checked must be checked. Every figure that will go stale must carry a date. Where the field genuinely disagrees, both positions are given rather than one being quietly chosen.

Where this book uses real evidence, it uses it unaltered. The network trace in Part F, the failed pushes in Part G and the error messages quoted throughout are reproduced exactly as they were observed. Diagnosis is taught against ambiguous real data, because that is the only kind there is.



KEDBYTE

TECHNOLOGIES PRIVATE LIMITED

COLOPHON

This first edition of *How Computers Work* runs to nine parts and fifty-three chapters, and was set entirely from plain text.

The body text is set in TeX Gyre Pagella, a digital revival of Hermann Zapf's Palatino, chosen for long-form reading. Headings, tables and the KedByte identity are set in Poppins. Code, command output and diagrams are set in DejaVu Sans Mono, and every diagram in this book is drawn in plain ASCII so that it survives being copied anywhere.

The book was composed with pandoc and typeset with XeLaTeX on A4 stock. The single accent colour used on rules, chapter numbers and section headings is KedByte navy.

Shikhar Singh

Founder & Chairman, KedByte Technologies Private Limited



First Edition • KedByte Press