KB

VOLUME FIVE

Intelligence, Games and Reference

from *How Computers Work*

KedByte Technologies Private Limited

KEDBYTE

TECHNOLOGIES PRIVATE LIMITED

HOW COMPUTERS WORK

From a Grain of Sand to Artificial Intelligence

VOLUME FIVE

Intelligence, Games and Reference

How a game runs sixty times a second, how a language model is trained, judged, released and served, and the reference desk of numbers, commands, decision trees, timeline and glossary.

WRITTEN BY

Shikhar Singh

Founder & Chairman

KedByte Technologies Private Limited

Parts H and I • Chapters 45 to 53 • First Edition

HOW COMPUTERS WORK

Volume Five — Intelligence, Games and Reference

Parts H and I, chapters 45 to 53

Author	Shikhar Singh
Designation	Founder & Chairman, KedByte Technologies Private Limited
Published by	KedByte Technologies Private Limited
Imprint	KedByte Press
Edition	First Edition
Volume	5 of 5
Complete work	Nine parts, fifty-three chapters

About this volume. This is Volume 5 of five. It contains Parts H and I of *How Computers Work*, chapters 45 to 53. Chapter numbers are those of the complete work, so a cross-reference to any chapter means the same chapter whether you are reading a single volume or the complete edition. References to chapters outside this volume point to the other volumes in the set.

Copyright © KedByte Technologies Private Limited. All rights reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other non-commercial uses permitted by copyright law.

Trademarks. All product names, company names, service marks and trademarks referred to in this book are the property of their respective owners, and are used in an editorial and educational capacity only.

Accuracy and currency. This book describes technology that changes. Specifications, prices, version numbers and industry figures were checked at the time of writing and are dated in the text wherever they are liable to change. Where experts disagree, the book says so rather than choosing a side.

Disclaimer. The information in this book is provided on an "as is" basis, for education. Neither the author nor KedByte Technologies Private Limited shall have any liability to any person or entity with respect to any loss or damage caused, or alleged to have been caused, directly or indirectly by the information contained in this book.

BEFORE YOU START

How to read this book

Every idea in this book is explained twice. First in plain language, with an everyday comparison and a worked example, so that a beginner can follow it. Then again in the correct technical vocabulary, with real units, real specifications and the exact figures an engineer would quote. The two halves are labelled, so you can read the plain pass end to end and then come back for the engineering.

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

1. Almost everything is written as short numbered lines, one idea per line, so nothing hides inside a paragraph.
2. *The honest version* marks the places where a simple explanation is not strictly true. The accurate model follows immediately.
3. The book says whether a thing is a **standard**, a **convention**, or one product's **implementation detail**.
4. Anything that goes stale is dated. Check it again before relying on it.
5. Every chapter ends with *Common wrong ideas* and a twenty-line summary. Short of time, read those two first.

The five volumes

1. **The Machine** — chapters 1 to 14
2. **Software** — chapters 15 to 22
3. **Networks** — chapters 23 to 34

4. **Building and Shipping Software** — chapters 35 to 44
5. **Intelligence, Games and Reference** — chapters 45 to 53

Contents

Part H — Games and Machine Intelligence 1

45 Games — How They Are Made and How They Run 2

45.0 What this chapter gives you 2
45.1 What a game actually is as a program 2
45.2 A history of video games as technology 5
45.3 The game loop in detail 8
45.4 How a game is built, as a project 13
45.5 Game engines 16
45.6 Assets and the content pipeline 19
45.7 Physics and collision 22
45.8 Game AI 24
45.9 Rendering a game frame, specifically 27
45.10 Audio in games 29
45.11 Multiplayer and netcode 32
45.12 Cheating and anti-cheat 35
45.13 What actually happens when you double-click a game 37
45.14 Consoles versus PC 40
45.15 Emulation 43
45.16 Making your first game, practically 45
45.98 Common wrong ideas 47
45.99 Chapter summary in 20 lines 48

46 Machine Learning From Scratch — Fitting, Loss, Gradients and Neural Networks 50

46.0 What this chapter gives you 50
46.1 What machine learning actually is 50
46.2 A short honest history 53
46.3 Fitting a line, the foundation of everything 57
46.4 Loss: how you measure being wrong 60
46.5 Gradient descent: walking downhill in fog 63
46.6 The vocabulary of training 68
46.7 Classical machine learning before neural networks 71
46.8 The neuron 75
46.9 Activation functions 77
46.10 Layers and depth 80
46.11 The forward pass as matrix multiplication 83
46.12 Backpropagation 85
46.13 Training in practice 89
46.14 The families of neural network 92
46.15 What a trained model physically is 96
46.98 Common wrong ideas 98
46.99 Chapter summary in 20 lines 99

47 Parameters, Tokens and Embeddings 100

47.0 What this chapter gives you 100
47.1 What a parameter actually IS in a language model 100
47.2 What 7B, 70B and 405B actually count 103
47.3 Parameters versus the four things they get confused with 107

47.4 Parameters, memory and precision 110
47.5 Quantization 113
47.6 Why capability does not scale linearly with parameter count 117
47.7 Tokens: why text is not fed in as words or characters 120
47.8 What tokenization does in practice 123
47.9 Token economics 127
47.10 Embeddings: turning a token id into a vector 129
47.11 What embeddings capture 132
47.12 Embeddings as a tool in their own right 135
47.13 The context window 138
47.14 Putting it together: one prompt from text to numbers 141
47.98 Common wrong ideas 143
47.99 Chapter summary in 20 lines 144

48 The Transformer 146

48.0 What this chapter gives you 146
48.1 What came before, and why it was not enough 146
48.2 Attention in plain language, before any mathematics 150
48.3 Query, key and value, taught slowly 153
48.4 Scaled dot-product attention, with the actual mathematics 156
48.5 Masking: stopping the model reading ahead 161
48.6 Multi-head attention 164
48.7 Position: attention has no idea what order things are in 168
48.8 The rest of the block: feed-forward, residuals and normalization 172
48.9 The whole model, assembled 176
48.10 The 2017 paper 179
48.11 The three architectures 182
48.12 Generation: producing text one token at a time 184
48.13 The cost of attention: the quadratic problem 188
48.14 Mixture of experts: paying for a big model, running a small one 191
48.15 Why one architecture does so many tasks 198
48.98 Common wrong ideas 203
48.99 Chapter summary in 20 lines 205

49 How a Model Is Actually Made — Pretraining to Release 207

49.0 What this chapter gives you 207
49.1 The pipeline in one view 207
49.2 The data: where it comes from and how it is cleaned 210
49.3 Pretraining itself 214
49.4 Scaling laws: how big, how much data 218
49.5 What a base model actually is 221
49.6 Supervised fine-tuning 223
49.7 Learning from preferences 226
49.8 What each stage actually changes 230
49.9 Efficient adaptation: LoRA, QLoRA and distillation 232
49.10 What a training run looks like physically 235
49.11 What weights physically are 238
49.12 Open weights versus open source 242
49.13 The Open Source Initiative's position 245
49.14 Model releases in practice 247
49.98 Common wrong ideas 250
49.99 Chapter summary in 20 lines 251

50 Benchmarks, Evaluation, Versions and Dates 253

50.0 What this chapter gives you 253
50.1 What a benchmark is, mechanically 253
50.2 The major benchmarks, one at a time 256
50.3 Actually running an evaluation 262
50.4 Why two labs report different numbers 266
50.5 Contamination 268

50.6 The other reasons to be sceptical 272
50.7 The alternatives, and their own weaknesses 275
50.8 What "eval" means inside a company 278
50.9 Reading a model announcement critically 280
50.10 Training cutoff versus release date 282
50.11 Versions, snapshots and the moving-name problem 284
50.12 Deprecation and retirement 286
50.13 What you should actually do 288
50.98 Common wrong ideas 290
50.99 Chapter summary in 20 lines 291

51 Serving, Running One Yourself, RAG, Agents and the Limits 292

51.0 What this chapter gives you 292
51.1 Inference versus training: what changes 292
51.2 What an API call actually contains 294
51.3 How generation actually performs 297
51.4 The serving optimizations, each explained 299
51.5 Cost and capacity 301
51.6 Running one yourself, practically 303
51.7 What hardware you actually need 306
51.8 Getting weights and reading a model card 308
51.9 Building something small from scratch, honestly 310
51.10 Prompting as an engineering practice, not a trick 312
51.11 RAG: giving a model your documents 314
51.12 Tool use and function calling 317
51.13 Agents: what the word actually means 319
51.14 Memory, and why a model with none appears to have some 321
51.15 The limits, stated plainly 322
51.16 The gaps you did not ask about but need 325
51.17 A practical decision guide 326
51.98 Common wrong ideas 328
51.99 Chapter summary in 20 lines 329

Part I — The Reference Desk 330

52 The Engineer's Reference 331

52.0 How to use this chapter 331
52.1 Numbers every engineer should know 331
52.2 Units and conversions 336
52.3 Number bases and encoding 340
52.4 The networking reference 343
52.5 Networking command cheat sheet 351
52.6 Decision tree: I cannot reach a site 358
52.7 Git command cheat sheet 361
52.8 Decision tree: my git operation failed 365
52.9 Shell and system cheat sheet 369
52.10 File format identification by magic number 372
52.11 Complexity and performance reference 373
52.12 Cloud and infrastructure reference 375
52.13 Security reference 377
52.14 AI and machine learning reference 380
52.15 Regular expressions 382
52.16 Keyboard shortcuts and editor basics 383
52.17 What to do when you are stuck 385
52.98 Common wrong ideas 386
52.99 Chapter summary in 20 lines 387

53 Master Timeline and Glossary 388

53.0 How to use this chapter . . . 388
53.1 The master timeline of computing . . . 388
53.2 Timeline of programming languages . . . 400
53.3 Timeline of standards and protocols . . . 402
53.4 The glossary . . . 405
53.5 Where to go next . . . 435
53.6 The whole book in 100 lines . . . 435

KB

PART H

Games and Machine Intelligence

How a game runs sixty times a second, and how a language model is built, judged and used.

45

CHAPTER

KB

Games — How They Are Made and How They Run

45.0 What this chapter gives you

1. You will be able to say exactly what makes a game different from every other program on your machine, in one sentence, and defend it.
2. You will be able to write a correct game loop with a fixed physics step, an accumulator and interpolation, and explain every line of it.
3. You will be able to do frame budget arithmetic in your head, and explain why "average frames per second" hides the thing that actually annoys you.
4. You will be able to describe how a real game project is staffed and funded, from a one-person hobby project to a three-hundred-person AAA title, with real money figures.
5. You will be able to compare Unity, Unreal Engine and Godot honestly on cost, language and licence, including the 2023 Unity fee episode.
6. You will be able to explain what an asset pipeline does, why a build takes hours, and what "cooking" content means.
7. You will be able to work through a collision resolution by hand and check that momentum is conserved.
8. You will be able to run the A-star pathfinding algorithm on a grid by hand, and explain why game AI is designed to be beaten.
9. You will be able to explain client-side prediction, server reconciliation, entity interpolation and lag compensation, and say precisely why you get shot after you reached cover.
10. You will be able to trace what happens between the double-click and the first frame of gameplay, in twenty-five steps with real timings.

45.1 What a game actually is as a program

45.1.1 PLAIN – in simple words

1. Most programs are lazy. They sit still and wait for you to do something.
2. A text editor draws your document once, then does nothing until you press a key. If you walk away for an hour, it does nothing for an hour.
3. A game is the opposite. A game never sits still.
4. A game runs the same short list of jobs over and over, very fast, forever, whether or not you touched anything.
5. That list is: read the controls, work out what the world looks like now, draw it, show it. Then do it again.
6. One trip round that list is called a **frame**.
7. At 60 frames a second, the game does this 60 times every second. That is once every 16.67 thousandths of a second.

8. It keeps doing it even when you stand perfectly still in an empty room.
9. Why? Because the world keeps moving. Water flows, clouds drift, an enemy walks somewhere far away, a timer counts down.
10. The game cannot know in advance whether anything changed. Working that out would cost more than just redoing the work.
11. So it redoes all of it, every single time, on a strict clock.
12. That one decision, "redo everything on a clock", explains almost every other odd thing about how games are built.

45.1.2 PLAIN - a picture in your head

1. Think of two people who work in the same building.
2. The first is a receptionist. She sits at a desk and does nothing until someone walks in. Then she deals with them and goes back to waiting.
3. Her day is quiet. Her energy use is low. Nothing happens between visitors.
4. The second is a film projectionist in an old cinema.
5. He cannot wait for anything. Twenty-four times a second a new picture must be in front of the lamp, whether the film is exciting or boring.
6. If he is late by even a fraction of a second, the audience sees a flicker and the illusion of motion breaks.
7. The receptionist is an ordinary application. The projectionist is a game.
8. Now notice what this does to the projectionist's job. He cannot go and make tea. He cannot take a long phone call. He has a deadline every 41 thousandths of a second, all day.
9. Everything about how he arranges his booth is decided by that deadline.

Where this comparison breaks: the projectionist is only showing pictures that already exist on the film. A game invents each picture from scratch, having first worked out what the world looks like at that instant. It is closer to a projectionist who must also draw each frame in the time between showing them. Also, real film runs at a fixed 24 frames per second and cannot be late. A game can be late, and when it is, you see it as a stutter rather than a flicker.

45.1.3 PLAIN - a worked example

1. Take one hour of a person using a word processor, and one hour of the same person playing a game.
2. In the word processor, the screen is redrawn when something changes: a letter typed, a scroll, a menu opened.
3. A fast typist hits about 400 keys a minute, so about 24,000 keys an hour.
4. Even if each key caused one redraw, that is 24,000 redraws in an hour.
5. In the game at 60 frames a second: 60 x 60 x 60 = 216,000 frames an hour.
6. That is nine times more drawing, and the drawing itself is far heavier: millions of triangles instead of a page of text.
7. Now watch what the machine does.

Program state	Word processor	Game at 60 fps
Screen updates per hour	~24,000	216,000
CPU while idle	near 0 percent	one core busy
GPU while idle	near 0 watts	150 to 450 watts
Deadline per update	none	16.67 ms

8. The word processor has no deadline at all. If a redraw takes 200 milliseconds, nobody dies.
9. The game has a hard deadline, 60 times a second, and missing it is visible.
10. This is why your laptop fans spin up for a game and stay silent for a document, even though both are "just a program".

45.1.4 PLAIN - what is really happening inside

1. Both kinds of program start the same way. The operating system loads them and calls a starting function.
2. The lazy program then enters a **message loop**. It asks the operating system: "is there anything for me?" and, if not, it goes to sleep.
3. Going to sleep is the key. The program is removed from the list of things that need the processor. It uses no time at all.
4. The operating system wakes it only when a real event arrives: a key, a mouse click, a window resize, a timer.
5. When it wakes, it handles the one event, redraws only the part of the window that changed, and goes back to sleep.
6. A game also has a message loop, because the operating system still sends it events. But it never sleeps in it.
7. Instead it asks "is there anything for me?", takes whatever is there without waiting, and immediately moves on to its own work.
8. Its own work is: read the current state of every input device, advance the simulated world by a small slice of time, and build a new picture.
9. Then it asks the graphics system to show that picture, and starts again.
10. The only place it ever waits is at the very end, when it hands the finished picture to the display and waits for the screen's next refresh.
11. That single wait is what keeps a game at exactly 60 or 120 frames a second instead of running as fast as the hardware allows.

45.1.5 TECHNICAL - the engineer's version

1. An event-driven application on Windows blocks in `GetMessage`, which does not return until a message is in the thread's queue. The thread is in a wait state and consumes no scheduler quantum.
2. It redraws in response to `WM_PAINT`, and the system tracks an **update region** so only invalidated rectangles are repainted. On macOS the equivalent is `setNeedsDisplay` and `drawRect`; on X11 it is `Expose`.
3. A real-time application instead uses `PeekMessage` with `PM_REMOVE`, which returns immediately whether or not a message is waiting, then falls through to its own frame code. This is the standard game shape and has been since the Win32 era.
4. The frame ends with a present: `IDXGISwapChain::Present` in Direct3D, `vkQueuePresentKHR` in Vulkan, `[CAMetalDrawable present]` in Metal, `SDL_GL_SwapWindow` or `eglSwapBuffers` in the portable stacks.
5. The swap chain holds two or three back buffers. With a sync interval of 1, Present blocks until the display's vertical blanking interval, which is what pins the frame rate to the refresh rate.
6. With variable refresh rate displays (VESA Adaptive-Sync, marketed as AMD FreeSync from 2015 and NVIDIA G-Sync from 2013) the panel waits for the game rather than the other way round, within a supported range, typically 48 to 144 Hz or wider.
7. The consequences of the loop shape are measurable. A game pins at least one CPU core near 100 percent and keeps the GPU in its highest power state. A discrete GPU that idles at 10 to 25 W will draw 150 to 450 W under load.
8. The other big consequence is memory behaviour. An event-driven application can allocate freely, because a 5 ms garbage collection pause is invisible. A game cannot: a 5 ms pause inside a 16.67 ms budget is 30 percent of the frame, and a 30 ms pause is a visible hitch.
9. This is why game code pre-allocates pools, avoids per-frame heap allocation, and why C# game code is written to keep the generation 0 heap quiet rather than in idiomatic allocating style.

Property	Event-driven app	Real-time game
Blocking call	GetMessage	none, PeekMessage

Property	Event-driven app	Real-time game
Redraw trigger	invalidated region	every frame
Frame deadline	none	16.67 ms at 60 Hz
Allocation policy	free	pooled, per-frame zero

10. Tools that show the difference: Windows Task Manager and `powermetrics` on macOS for power draw, `perf top` on Linux for the busy loop, PresentMon on Windows for per-frame present timings, and RenderDoc for a captured frame.

45.1.6 WORDS - remember these

1. Frame — one complete picture — one iteration of the game loop producing one presented back buffer.
2. Game loop — the list of jobs done over and over — the input, update, render, present cycle running at the target frame rate.
3. Event-driven — waits until poked — blocks on a message queue and repaints only invalidated regions.
4. Present — showing the finished picture — handing a back buffer to the display engine, optionally synchronized to vertical blank.
5. Swap chain — the queue of pictures waiting to be shown — a set of back buffers rotated between the renderer and the display controller.
6. Vertical blank — the pause between screen refreshes — the interval in which a buffer swap can occur without tearing.

45.2 A history of video games as technology

45.2.1 PLAIN - in simple words

1. Video games did not start as a business. They started as people playing with expensive laboratory equipment.
2. In 1958 a physicist wired an analogue computer to a small round screen so that visitors to his laboratory's open day would have something to do.
3. In 1962 students at a university did something similar on a room-sized computer, and the program spread to every machine of that type.
4. The first machines you could buy for a home, in 1972, had no computer inside at all. They were built from simple switching parts wired together.
5. Then chips got cheap. Arcade cabinets became small computers with one game burned into them.
6. Then home consoles became small computers with a slot for a cartridge.
7. Then home computers became fast enough to draw a three-dimensional world, and then extra chips were added just for drawing.
8. Then the internet arrived, and games stopped being finished things and started being services that keep changing.
9. Then phones arrived, and more people played games than ever before, mostly for free, paying in small amounts inside the game.
10. Every one of those steps was a technical step first, and a business step second.

45.2.2 PLAIN - a picture in your head

1. Think about how photography grew up.
2. First there were one-off chemical experiments by scientists, making a single image with enormous effort.
3. Then there were heavy plate cameras owned by professionals with studios.
4. Then a company put film in a small box and said "you press the button, we do the rest", and ordinary people took pictures.

5. Then colour arrived, then instant prints, then video, then digital, then a camera in every pocket connected to the whole world.
6. Games followed the same shape, about eighty years later and much faster.
7. The laboratory oscilloscope is the chemical experiment. The arcade cabinet is the professional studio. The home console is the box with film in it. The phone is the camera in every pocket.

Where this comparison breaks: photography captures something that already exists. A game must create everything, so each step forward needed not just better hardware but also new mathematics and new ways of organizing a team. And unlike photography, games have kept the old forms alive: people still make and play games that would run on 1985 hardware.

45.2.3 PLAIN – a worked example

1. Compare one specific thing across the decades: how many moving pictures per second the machine had to produce, and what it had to remember.

Machine and year	Working memory	Screen
Odyssey, 1972	none	3 white dots
Atari 2600, 1977	128 bytes	160x192, 4 colours
NES, 1983	2 KB	256x240, 25 colours
PlayStation, 1994	2 MB + 1 MB video	320x240 to 640x480
PlayStation 5, 2020	16 GB shared	3840x2160

2. Read the memory column again. 128 bytes on the Atari 2600 is not a typing error. That is 128 individual bytes for all game state.
3. The whole screen would not fit in that memory, so the machine did not have a screen buffer at all.
4. Instead the processor had to change the graphics chip's registers while the television's electron beam was sweeping across the screen, line by line.
5. Programmers called this **racing the beam**. You had about 76 processor cycles per scan line to set up the next line.
6. From 128 bytes to 16 gigabytes is a factor of 134 million in about 43 years.

45.2.4 PLAIN – what is really happening inside

1. Each era solved one specific bottleneck, and the games of that era look the way they do because of that one bottleneck.
2. The 1972 machines had no processor, so the game was fixed in wiring. You could not have a game with rules that changed.
3. The 1977 to 1983 machines had a processor but almost no memory, so the world had to be tiny and mostly regenerated each frame.
4. The 1983 to 1990 machines added dedicated graphics hardware that could move small pictures around cheaply, so games became about sprites and scrolling.
5. The early 1990s machines were fast enough to compute perspective in software, so first-person games appeared, with clever mathematics used to avoid work rather than to do it.
6. From 1996 a separate chip did the triangle filling, so the limit moved from "how many pixels can we fill" to "how many triangles can we send".
7. From about 2001 those chips became programmable, so the limit moved again to "what can we compute per pixel".
8. From about 2013 the limit for many games became not the hardware but the cost of making enough content to fill it.
9. That last shift is the reason a modern game costs two hundred million dollars and a 1985 game cost a few hundred thousand.

45.2.5 TECHNICAL - the engineer's version

Year	Milestone	Technical advance
1958	Tennis for Two	Real-time analogue graphics
1962	Spacewar, PDP-1	Stored-program interactive
1971	Computer Space	First coin-operated cabinet
1972	Magnavox Odyssey	Home console, discrete logic
1972	Pong arcade	TTL logic, no CPU, no code
1977	Atari VCS / 2600	Cartridge ROM, 6507 CPU
1978	Space Invaders	Microprocessor arcade game
1983	Market crash	Quality control matters
1983	Famicom in Japan	Sprite and scroll hardware
1985	NES in North America	Lockout chip, licensing model
1988	Mega Drive	16-bit 68000, bigger sprites
1990	Super Famicom	Mode 7 affine transforms
1992	Wolfenstein 3D	Ray-cast pseudo-3D at speed
1993	Doom	BSP trees, modding, LAN play
1994	PlayStation	Hardware triangle rasterizer
1996	3dfx Voodoo Graphics	Consumer 3D acceleration
1999	Dreamcast online	Built-in modem, online play
2002	Xbox Live	Unified console online service
2003	Steam	Digital distribution, patching
2008	Apple App Store	Mobile mass market
2017	Fortnite Battle Royale	Live service at scale
2020	PS5 and Xbox Series X	NVMe streaming, RT hardware
2024	PS5 Pro	Machine-learning upscaling
2025	Nintendo Switch 2	DLSS on a handheld

1. Tennis for Two was shown on 18 October 1958 at Brookhaven National Laboratory in Upton, New York, built by William Higinbotham with technician Robert V. Dvorak. It ran on a Donner Model 30 analogue computer with a five-inch oscilloscope as the display. Each player had an aluminium box with a knob to set the angle and a button to hit the ball.
2. Spacewar was written in 1962 by Steve Russell and others at MIT on a DEC PDP-1 with 9 kilowords of core memory and a Type 30 point-plotting display. It was copied to essentially every PDP-1 installation, which makes it the first game with a distribution story.
3. The Magnavox Odyssey, designed by Ralph Baer at Sanders Associates from the 1967 "Brown Box" prototype, shipped in September 1972. It had no processor, no memory and no software: transistor and diode logic produced three white spots, and coloured plastic overlays taped to the television supplied the backgrounds.
4. Pong, designed by Allan Alcorn at Atari and installed in November 1972, was also built from discrete transistor-transistor logic. There is no program to disassemble because there is no program.
5. The Atari Video Computer System, later the 2600, launched 11 September 1977 with a MOS 6507 at about 1.19 MHz, 128 bytes of RAM, and the Television Interface Adaptor. It had no frame buffer, so the CPU had to modify the TIA registers within each of the 262 scan lines of an NTSC field.
6. The 1983 crash is well documented in numbers: United States home video game revenue fell from roughly 3.2 billion dollars in 1983 to roughly 100 million dollars in 1985, a drop of about 97 percent. Contributing factors were oversupply, poor quality third-party titles, and cheap home computers. Atari buried roughly 728,000 cartridges in a New Mexico landfill in September 1983, confirmed by an excavation in 2014.

7. Nintendo's answer was a hardware lock. The Famicom launched 15 July 1983 in Japan; the redesigned Nintendo Entertainment System launched in New York on 18 October 1985. Its 10NES system used a Checking Integrated Circuit in the console and an identical one in the cartridge, sharing a 4 MHz clock. The console's chip sends a challenge and the cartridge's chip must return a matching pseudo-random stream. On mismatch the console pulls the CPU and picture processing unit reset lines low with a 1 Hz square wave, producing the flashing screen. Different part numbers enforced regions: 3193 and 6113 for NTSC, 3197 for PAL-A, 3195 for PAL-B, 3196 for Asia.
8. The honest version: the lockout chip was not security in the modern sense. It was a business control. It let Nintendo decide who published on its platform, cap the number of titles per publisher, and manufacture all cartridges itself. Every console since has had some version of that control, and it is the reason the certification process in section 45.14 exists.
9. Doom shipped on 10 December 1993. Its renderer precomputed a binary space partitioning tree at level build time so that walls could be drawn in correct depth order without a depth buffer. Walls were drawn as vertical texture columns; floors and ceilings as horizontal spans called visplanes, with a hard limit of 128 of them. All arithmetic was 16.16 fixed point. It could not do rooms above rooms, sloped floors, or a proper look up and down. It also shipped as shareware, kept its data in an editable WAD file, and supported four-player networked deathmatch over IPX, which created the modding and competitive scenes at the same time.
10. 3dfx Voodoo Graphics arrived in late 1996 as an add-in card that did only 3D and passed 2D through from the existing card. GLQuake in January 1997 was the demonstration that made the category mainstream.
11. The PlayStation launched 3 December 1994 in Japan and 9 September 1995 in North America. Its geometry engine had no floating-point unit and its rasterizer had no depth buffer, so it used fixed-point maths and sorted polygons into ordering tables, which is why its textures wobble and polygons sometimes pop through each other.
12. Console online gaming became normal in stages: the Dreamcast shipped with a 56k modem in 1998 in Japan and 1999 elsewhere, and Xbox Live launched on 15 November 2002 with a broadband-only, subscription, unified-identity model that everyone else eventually copied.
13. As of August 2026 the current landscape is: PlayStation 5 and PlayStation 5 Pro, Xbox Series X and Series S, Nintendo Switch 2 released 5 June 2025, a PC market dominated by Steam, a handheld PC category led by the Steam Deck, and a mobile market larger by player count than all of them together.

45.2.6 WORDS - remember these

1. Racing the beam — changing the picture while the screen draws it — writing video registers in sync with horizontal scan timing on a frame-buffer-less system.
2. Sprite — a small movable picture — a hardware-composited image layer with its own position registers, independent of the background.
3. Lockout chip — the part that says which cartridges are allowed — a challenge-response microcontroller pair gating the console reset line.
4. BSP tree — a way of pre-sorting a level — a binary space partitioning structure allowing back-to-front or front-to-back traversal from any point.
5. Shareware — try the first part free — a distribution model splitting a game into a freely copyable episode and paid episodes.
6. Fixed point — fractions stored as whole numbers — a numeric format such as 16.16 with an implicit binary scaling factor, used where no FPU exists.

45.3 The game loop in detail

45.3.1 PLAIN - in simple words

1. The loop has four jobs, always in this order.
2. **Input**: find out what the player is doing right now. Which keys are down, where the mouse moved, how far the trigger is pulled.
3. **Update**: move the world forward by a small slice of time. Apply gravity, move characters, run the enemy brains, check what hit what.
4. **Render**: build the picture that shows the world in its new state.
5. **Present**: hand that picture to the screen.
6. Then start again from input.
7. There is one hard question in the middle of this: how big is that "small slice of time"?
8. The obvious answer is "however long the last frame took". That is called a **variable timestep**, and it is wrong for physics.
9. The reason is that physics maths gives different answers for different slice sizes. Big slices lose detail and small slices keep it.
10. So if your machine is fast and mine is slow, a jump that clears a gap on your machine can fail on mine. That is not acceptable.
11. The fix is to always advance physics by exactly the same slice, for example one sixtieth of a second, no matter how long the frame took.
12. If the frame took longer, you run the physics slice more than once. If it took less, sometimes you do not run it at all.
13. That is called a **fixed timestep**, and it is what almost every serious game does.

45.3.2 PLAIN - a picture in your head

1. Imagine filling a bucket from a tap, and every time the bucket is full you must pour it into a barrel.
2. The tap is real time flowing past. The bucket is a store of time you have not simulated yet. The bucket's size is your fixed step, say 16.67 thousandths of a second.
3. Each time round the loop, you measure how much water came out of the tap since last time and add it to the bucket.
4. Then, while the bucket has at least one bucketful in it, you empty one bucketful into the barrel. That emptying is one physics step.
5. Sometimes you empty it twice, because the frame was slow and two bucketfuls collected. Sometimes you empty it zero times, because the frame was fast.
6. The leftover water in the bucket is real time that has passed but has not been simulated. It is always less than one full bucket.
7. That leftover is not wasted. You use it to decide how far between the last two physics positions to draw the picture. Half a bucket left means draw halfway between.
8. That last trick is called **interpolation**, and it is why a game with a 60-step physics simulation can look perfectly smooth at 144 frames per second.

Where this comparison breaks: water is continuous and time in a computer is not. Real clocks are read as whole counts of a hardware timer. Also, a real bucket cannot overflow into an infinite queue; a game must put a limit on how much time it will try to catch up, or a slow frame causes a longer frame, which causes a longer one, and the game locks up. That failure is called the **spiral of death** and the fix is to clamp the measured frame time.

45.3.3 PLAIN - a worked example

1. Fixed step is 16.67 ms, which is 1/60 of a second. The screen refreshes at 144 Hz, so a frame arrives every 6.94 ms.
2. Start with the bucket empty. Follow ten frames.

Frame	Real time added	Bucket after	Steps run
1	6.94 ms	6.94 ms	0
2	6.94 ms	13.88 ms	0
3	6.94 ms	4.15 ms	1
4	6.94 ms	11.09 ms	0
5	6.94 ms	1.36 ms	1
6	6.94 ms	8.30 ms	0
7	6.94 ms	15.24 ms	0
8	6.94 ms	5.51 ms	1

3. Over eight frames, three physics steps ran. That is 3 x 16.67 = 50 ms of simulated time for 8 x 6.94 = 55.5 ms of real time, with 5.5 ms still in the bucket. The books balance.
4. Notice that the picture changed on all eight frames, even the five where no physics ran. Those five frames were drawn by interpolating.
5. On frame 4 the bucket held 11.09 ms out of a 16.67 ms step, so the interpolation factor was 11.09 / 16.67 = 0.665. Draw the world 66.5 percent of the way from the previous physics position to the current one.
6. Now the frame budget arithmetic, which you should know by heart.

Target rate	Budget per frame	Ticks per second
30 fps	33.33 ms	30
60 fps	16.67 ms	60
90 fps	11.11 ms	90
120 fps	8.33 ms	120
144 fps	6.94 ms	144
240 fps	4.17 ms	240

7. The budget is 1000 divided by the target rate. That budget covers everything: input, physics, AI, animation, audio submission, all the render commands, and the driver's own work.
8. At 144 fps you have 6.94 ms total. If your shadow pass alone takes 3 ms you have already spent 43 percent of the frame on shadows.

45.3.4 PLAIN - what is really happening inside

1. Here is the real loop, with the fixed step, the accumulator and the interpolation. Every line is explained underneath.

```
const double dt = 1.0 / 60.0;   /* fixed step: 16.667 ms   */
double accumulator = 0.0;       /* unsimulated time store  */
double t           = 0.0;       /* total simulated time    */
double currentTime = now();     /* clock at loop start     */

while (running) {
    double newTime   = now();
    double frameTime = newTime - currentTime;
    if (frameTime > 0.25) frameTime = 0.25;   /* clamp   */
    currentTime = newTime;

    accumulator += frameTime;

    poll_input();

    while (accumulator >= dt) {
        previous_state = current_state;
```

```
        simulate(&current_state, t, dt);
        t           += dt;
        accumulator -= dt;
    }

    double alpha = accumulator / dt;
    State view = lerp(previous_state, current_state, alpha);

    render(view);
    present();
}
```

2. `dt` is the fixed simulation step. It never changes at run time. Every physics calculation in the game is written knowing this exact number.
3. `accumulator` holds real time that has passed but has not yet been fed to the simulation. It is always between 0 and `dt` at the bottom of the loop.
4. `t` is total simulated time. Effects, animations and networked timestamps are keyed off this, not off the wall clock.
5. `currentTime` remembers when the previous loop iteration started, so we can subtract and get the elapsed time.
6. `now()` must be a monotonic high-resolution clock. On Windows that is `QueryPerformanceCounter`; on Linux `clock_gettime(CLOCK_MONOTONIC)`; on macOS `mach_absolute_time`. Never use wall-clock time, because it can jump backwards when the machine syncs with a time server.
7. `frameTime` is how long the previous whole frame took, in seconds.
8. The clamp to 0.25 seconds is the spiral-of-death guard. If the process was suspended, or a big file loaded, or you dragged the window, `frameTime` might be 8 seconds. Without the clamp the loop would try to run 480 physics steps in one frame, take even longer, and never recover. With the clamp the game simply loses that time. Slow motion is better than a freeze.
9. `poll_input` reads the devices once per frame, before any simulation, so that every physics step inside this frame sees the same input.
10. The inner `while` is the fixed-step loop. It runs zero, one or more times, consuming one `dt` of accumulator each time.
11. `previous_state = current_state` saves the state before stepping, so we have two snapshots to interpolate between.
12. `simulate` advances the world by exactly `dt`. It never sees the real frame time and never needs to.
13. `alpha` is the leftover fraction, between 0 and 1. It says how far past the last physics step the display moment is.
14. `lerp` is linear interpolation: `a + (b - a) * alpha`. Applied to positions it is straightforward; applied to rotations you use spherical interpolation of quaternions instead.
15. `render` draws the interpolated view, not the simulation state. This is the line most beginners get wrong, and getting it wrong causes visible judder when the frame rate is not an exact multiple of the tick rate.
16. `present` hands the buffer to the display and, with vertical sync on, blocks until the next refresh.

45.3.5 TECHNICAL - the engineer's version

1. The accumulator pattern above is the one popularized by Glenn Fiedler's article "Fix Your Timestep", first published in 2004 and widely revised since. It is a convention, not a standard, but it is close to universal.
2. Fixed step is required whenever the integration scheme is not step-size-invariant, which is every scheme used in games. Semi-implicit Euler with a 33 ms step and with a 4 ms step give different trajectories, and constraint solvers converge differently at different step sizes.
3. Typical simulation rates: 60 Hz is the default for most engines; 30 Hz for large-world console titles that trade physics fidelity for CPU; 120 Hz or 240 Hz for racing, fighting and VR titles; 128 Hz for competitive shooters on the server.
4. Substepping is separate: a solver may internally split one 16.67 ms step into 4 substeps of 4.17 ms to improve stability for fast objects, while the game still ticks at 60 Hz.
5. **Frame time** is the wall-clock duration of one frame in milliseconds. **Frames per second** is 1000 divided by the mean frame time. Frame time is the primary measurement; frames per second is a derived, compressed and lossy summary of it.
6. Why the average lies. Take 100 frames: 99 of them at 8.00 ms and one at 100 ms. Total elapsed is 99 x 8 + 100 = 892 ms. Mean frame time is 8.92 ms, so the reported average is 112.1 fps. But the player saw one frame held on screen for a tenth of a second, which reads as a clear hitch.
7. This is why reviewers and profilers report percentiles. The **1 percent low** is, by the most common convention, the average frame rate of the slowest 1 percent of frames; some tools instead report the frame time at the 99th percentile. The two definitions give different numbers and the industry has not settled on one. When comparing figures, check which was used.
8. In the example above, the 1 percent low is 10 fps by either definition, against a 112 fps average. That gap is exactly the complaint people describe as "the frame rate is fine but it feels bad".
9. **Frame pacing** is a separate problem from frame rate. A sequence of frame times of 8, 8, 8, 8 ms feels smoother than 4, 12, 4, 12 ms even though both average 8 ms, because the second sequence presents images at uneven intervals against an even display refresh.
10. The full click-to-photon latency chain, with realistic numbers for a PC at 60 Hz with vertical sync and a 2-frame render queue:

Stage	Typical latency
Mouse sample and USB	1 to 8 ms
Input poll to sim	0 to 16.7 ms
Simulate and render	16.7 ms
Render queue depth	16.7 to 33.4 ms
Display scanout and panel	5 to 20 ms

11. That totals roughly 40 to 95 ms, which is why competitive players disable vertical sync, cap frames just below the refresh rate, use variable refresh, and enable low-latency modes such as NVIDIA Reflex (2020) or AMD Anti-Lag, all of which mainly work by shortening the render queue.
12. Measuring tools: PresentMon and its front ends on Windows give per-frame present timings and a latency estimate; `stat unit` and Unreal Insights in Unreal Engine; the Unity Profiler; RenderDoc and PIX for GPU timings; `perf` and Tracy for CPU timeline capture.
13. The distinction between CPU frame time and GPU frame time matters. If the GPU is the bottleneck the CPU finishes early and waits; if the CPU is the bottleneck the GPU starves. Profilers report both, and the larger one is your frame time.

```
frame N          frame N+1        frame N+2
CPU  [sim|render cmds]  [sim|render cmds]  [sim|...
GPU        [ draw N  ]        [ draw N+1 ]      [ ...
DISP                  [show N ]         [show N+1]
```

```
CPU builds frame N while GPU draws N-1: this is pipelining.
It raises throughput and adds one frame of latency.
```

45.3.6 WORDS - remember these

1. Timestep — the slice of time simulated at once — the `dt` passed to the integrator each simulation tick.
2. Fixed timestep — always the same slice — a constant `dt` decoupled from frame duration, required for reproducible integration.
3. Accumulator — the store of unsimulated time — a scalar holding elapsed real time not yet consumed by fixed steps.
4. Interpolation — drawing between two known positions — blending previous and current simulation states by the leftover accumulator fraction.
5. Spiral of death — the game falls further and further behind — unbounded catch-up stepping where step cost exceeds real time elapsed.
6. Frame time — how long one frame took — wall-clock duration in milliseconds, the primary performance measurement.
7. 1 percent low — how bad the worst frames are — the mean frame rate of the slowest one percent of frames, or the 99th-percentile frame time.
8. Frame pacing — whether frames arrive evenly — the variance of frame intervals relative to the display refresh period.

45.4 How a game is built, as a project

45.4.1 PLAIN - in simple words

1. A game is not written. It is built, the way a film is shot or a building is put up, by people with very different skills.
2. The **designer** decides what the player does and why it is fun. Levels, rules, numbers, pacing.
3. The **programmer** makes the machine do it. Gameplay code, engine code, graphics code, tools for everyone else, network code.
4. The **artist** makes what you see. Concept art first, then the actual models, textures, environments and characters.
5. The **animator** makes things move: walking, reloading, dying, a door opening, a face speaking.
6. The **audio** people make what you hear: footsteps, gunfire, wind, music, voices.
7. **Quality assurance**, usually called QA, plays the game deliberately and methodically to find what is broken, and writes it down precisely.
8. The **producer** keeps the whole thing on a schedule and decides what gets cut when there is not enough time. There is never enough time.
9. On a big game there are more roles: technical artist, user interface designer, writer, localization, build engineer, live operations, community.
10. The single most important fact about this list is that programmers are a minority. On a large modern game, roughly one person in five writes code.

45.4.2 PLAIN - a picture in your head

1. Think of a restaurant opening.
2. Somebody decides what kind of restaurant it is and writes the menu. That is the designer.
3. Somebody builds the kitchen and makes the equipment work. That is the programmer.
4. Chefs cook. Decorators decorate. Someone chooses the music. Those are the artists, animators and audio people.
5. Before opening, friends come in and eat for free and say what was wrong. That is QA and playtesting.
6. Someone watches the money and the calendar and says “the dessert menu is cut, we open in three weeks”. That is the producer.

7. And a restaurant is never finished on opening night. The first month is fixing what real customers break. That is the launch patch and live operations.

Where this comparison breaks: a restaurant can open with a small menu and add dishes. A game usually has to ship one enormous simultaneous thing, and a missing piece in the middle stops everything. Also, a bad dish affects one diner; a bad save-file bug affects everyone at once, everywhere, at 3 a.m.

45.4.3 PLAIN - a worked example

1. The phases, in order, with what each one actually means.
2. **Concept**: a few people, a few weeks to a few months. A document, some sketches, maybe a rough playable thing. Most concepts die here, and that is the cheapest place to die.
3. **Pre-production**: build the risky bits. Answer "can we even do this". Small team, six months to two years on a big project.
4. **Vertical slice**: one small part of the game at full final quality. Ten minutes that look and feel exactly like the finished game. It exists to prove the vision and to unlock funding.
5. **Production**: the big, expensive middle. Make all the content. The team is at its largest here.
6. **Alpha**: feature complete. Every system exists, even if content is missing and it is full of bugs.
7. **Beta**: content complete. Everything is in. From here you only fix.
8. **Certification**: for consoles, the platform holder tests the build against a long checklist. See section 45.14.
9. **Gold and launch**: the build is locked and manufactured or uploaded.
10. **Live operations**: patches, seasons, events, balance changes, and sometimes years of new content.

Scale	Team size	Time	Budget range
Solo hobby	1	6 to 24 months	0 to 20k
Small indie	2 to 8	1 to 3 years	50k to 1M
Mid-size (AA)	30 to 80	2 to 3 years	5M to 30M
AAA	200 to 600	4 to 6 years	80M to 300M

11. Those budget figures are in United States dollars and exclude marketing, which on a AAA title is often as much again as development.

45.4.4 PLAIN - what is really happening inside

1. The reason production is expensive is content, not code.
2. A gun in a modern shooter is not one job. It is a concept sketch, a high-detail model, a low-detail game model, several texture maps, a rig, reload and inspect and fire animations, muzzle flash effects, four or five layered sounds, a tuning pass on damage and recoil, a first-person and a third-person version, and an entry in the user interface.
3. That is roughly two to six person-weeks for one weapon, and a game may have forty.
4. Multiply that pattern across characters, environments, vehicles and cutscenes and you get the two hundred million dollars.
5. Code is a smaller share, but it gates everything: nobody can build a level until the level tools work.
6. This is why the tools programmers are quietly the most valuable people on a big team. A one-second improvement in the editor's save time, times 200 people, times 50 saves a day, is 2.8 hours of team time per day.
7. And this is why the schedule always slips at the end. Content work is parallel and estimable. Integration, bug fixing and certification are serial and are not.

45.4.5 TECHNICAL - the engineer's version

1. Real documented budgets, from court filings and leaks rather than press releases, which is why they are trustworthy.

Game	Reported cost	Source and year
The Last of Us Part II	~220M, ~200 devs	Court exhibit, 2023
Horizon Forbidden West	~212M, ~300 devs	Court exhibit, 2023
Grand Theft Auto V	~265M with marketing	Reported, 2013
Cyberpunk 2077	~316M with marketing	CD Projekt, 2021
Stardew Valley	one person, 4.5 years	Developer, 2016

2. The Sony figures became public in June 2023 through poorly redacted documents filed in the United States Federal Trade Commission case about the Microsoft acquisition of Activision Blizzard. The Cyberpunk 2077 number is roughly 1.2 billion Polish zloty of combined development and marketing spend, converted at the rates of the time, and should be treated as approximate.
3. Team composition on a typical 250-person AAA project is roughly: 45 to 60 artists, 30 to 45 engineers, 20 to 30 designers, 20 to 30 animators, 10 to 15 audio, 30 to 60 QA, plus production, writing, user interface and localization. External outsourcing studios may add another 200 people who never appear on the internal headcount.
4. Costs per head, fully loaded with salary, benefits, hardware, software and office, run roughly 120,000 to 250,000 US dollars per person-year in North America and Western Europe, less elsewhere. A 250-person team for four years at 160,000 is 160 million dollars, which is where these budgets come from.
5. Now the honest part, because the reader asked for honesty.
6. **Crunch** means long mandatory overtime near milestones. Sixty to eighty hour weeks for months are documented at many studios. It is often unpaid for salaried staff. It correlates with burnout, health damage and people leaving the industry in their thirties.
7. The industry has argued about it publicly since the 2004 "EA Spouse" open letter, which led to overtime lawsuits and settlements at Electronic Arts. It recurred publicly around Rockstar Games in 2018 and CD Projekt Red in 2020.
8. The structural cause is not villainy. It is that games are fixed-date, fixed-scope products with an unpredictable final integration phase, sold in a market with a hit-driven revenue distribution. Cutting scope is the only real fix and it is the hardest decision to make.
9. **Layoffs** became the defining industry story of the 2020s. Trackers recorded roughly 10,500 games industry job losses in 2023 and roughly 14,600 in 2024, with further large numbers through 2025 and 2026. Studios have been closed within months of shipping well-reviewed games.
10. **Unionization** started to take hold in the same period, with organized bargaining units at several large studios from 2023 onward. Whether this changes crunch materially is still open.
11. Where experts disagree: some argue budgets are unsustainable and the AAA model must shrink; others argue the top few titles will keep growing and the middle will disappear. The evidence in 2026 supports both, because the middle has in fact been shrinking while the top has grown.

45.4.6 WORDS - remember these

1. Vertical slice — a small piece at final quality — a fully polished representative segment used to validate the design and secure funding.
2. Alpha — every feature exists — feature complete, content incomplete.
3. Beta — everything is in — content complete, bug fixing only.
4. Gold master — the locked build — the final candidate submitted for manufacture or platform release.
5. Crunch — mandatory long hours — sustained overtime at milestone boundaries, often uncompensated for salaried staff.

6. Live operations — running the game after launch — the ongoing content, balance, event and monetization pipeline for a service game.

45.5 Game engines

45.5.1 PLAIN - in simple words

1. An **engine** is the pile of reusable machinery that every game needs and that is not about your particular game.
2. It draws things. It plays sounds. It reads controllers. It works out what hit what. It loads files. It runs on Windows and consoles and phones.
3. It also gives you an **editor**: a program where you place objects, light a scene and press play, without writing code for any of it.
4. What it does not give you is a game. It gives you the ability to make one.
5. Studios use one instead of writing their own for a plain reason: writing your own costs several engineer-years before you can show anything.
6. Worse, you must then keep it working on five platforms as those platforms change, forever, with a team of maybe three people.
7. Big studios that do write their own do it because they need something the commercial engines do not do well: enormous open worlds, very specific physics, or an existing pile of tools twenty years deep.

45.5.2 PLAIN - a picture in your head

1. Think about building a house.
2. You could make your own bricks, mill your own timber and wind your own wire. People did, once.
3. Today you buy standard bricks, standard wire in standard colours, standard window units and standard plumbing fittings.
4. The standard parts are not what makes your house yours. The plan is.
5. But the standard parts decide a lot: what shapes are cheap, what is awkward, and what will simply not fit.
6. An engine is the standard parts. It makes some games cheap to build and others painful, and you feel that pressure the whole way through.

Where this comparison breaks: bricks do not change their price after you have built half the house. Engine licences can, and in 2023 one of them tried to. Section 45.5.5 tells that story exactly.

45.5.3 PLAIN - a worked example

1. What is actually inside an engine, listed as the parts you would otherwise have to write.

Part	What it does
Renderer	Turns a scene into pixels
Physics	Moves bodies, finds collisions
Audio	Mixes and positions sound
Input	Reads keyboard, mouse, pads
Scene system	Holds and finds objects
Asset pipeline	Imports and converts content
Scripting	Runs gameplay code safely
Editor	Lets non-programmers build
Build system	Packages for each platform
Networking	Replicates state to clients

2. A bare version of the first four is roughly 50,000 to 150,000 lines of code and two to four engineer-years for a competent small team.
3. A version with an editor, a working asset pipeline and console support is a decade of work by a real team. That is the honest scale.
4. Which is why a two-person studio using Godot ships in eighteen months, and a two-person studio writing an engine ships never.

45.5.4 PLAIN - what is really happening inside

1. Two ways of organizing the objects in a game world dominate, and the difference is about how the computer's memory works.
2. A **scene graph** is a tree. A car is a node; its wheels are children of the car; a mirror is a child of a door which is a child of the car.
3. Moving the car moves everything under it automatically, because each child position is stored relative to its parent.
4. This is natural for humans and it matches how artists think. It is what Unity's GameObject hierarchy and Godot's node tree give you.
5. Its weakness is speed. Each object is a separate lump of memory with pointers to its children, scattered across the heap.
6. When the game walks 10,000 objects to move them, it jumps all over memory, and the processor spends most of its time waiting for memory rather than computing.
7. An **entity-component-system**, always abbreviated ECS, is the answer.
8. An **entity** is just a number, an identity with no data of its own.
9. A **component** is a small plain lump of data, such as a position or a health value, stored in one big array with all the other positions.
10. A **system** is a function that runs over every entity that has a particular set of components, walking those arrays straight through.
11. Because all the positions sit next to each other in memory, the processor can fetch them in order and predict what comes next. That is the whole trick, and it can be several times faster.

45.5.5 TECHNICAL - the engineer's version

1. The cache arithmetic that makes ECS worth the awkwardness. A cache line on every current x86-64 and ARM64 processor is 64 bytes.
2. Suppose 10,000 objects, each a 256-byte object, and you want to add velocity to position: 12 bytes read, 12 bytes written per object.
3. In the object-oriented layout, each object touches at least one 64-byte line, usually two because the fields straddle. Memory traffic is at least 10,000 x 64 = 640 KB, in an access pattern the hardware prefetcher cannot predict.
4. In the component layout, positions are one contiguous 120 KB array and velocities another. Total traffic is 240 KB, perfectly sequential, fully prefetched. That is under half the traffic and far better latency hiding.
5. Blizzard's Overwatch is the best-documented shipped example. In Timothy Ford's GDC 2017 talk "Overwatch Gameplay Architecture and Netcode" the client is described as having about 46 systems and about 103 component types, with only three systems involved in gameplay networking.
6. Now the engines themselves, as of August 2026.

Engine	Language	Cost model
Unity 6	C#	Free under 200k revenue
Unreal Engine 5	C++ and Blueprints	Free under 1M lifetime
Godot 4	GDScript, C#, C++	MIT licence, free
In-house	usually C++	Salaries and years

Engine	Language	Cost model

7. **Unity** was announced at Apple's Worldwide Developers Conference in June 2005 as a Mac-only engine. It now targets more than nineteen platforms. It scripts in C#; the old UnityScript was deprecated in August 2017. Unity Personal is free below 200,000 US dollars of annual revenue and funding. Unity Pro is required above that; its list price was 2,200 dollars per seat per year from 1 January 2025 and 2,310 dollars per seat per year from 12 January 2026. Unity Enterprise is required above 25 million dollars.
8. The 2023 episode, stated precisely. On 12 September 2023 Unity announced a **Runtime Fee**: a per-install charge on games above revenue and install thresholds, to begin 1 January 2024. The reaction was severe, because the fee applied retroactively to already-shipped games built under different terms, was based on installs rather than sales, and had no clear way to count them.
9. Unity revised the terms on 22 September 2023. Chief executive John Riccitiello retired on 9 October 2023. Matthew Bromberg became chief executive in May 2024. On 12 September 2024, exactly one year after the announcement, Unity cancelled the Runtime Fee outright and returned to seat-based subscriptions, doubling the Personal threshold from 100,000 to 200,000 dollars and raising Pro by 8 percent and Enterprise by 25 percent instead.
10. The lesson is not "Unity is bad". It is that a licence is part of your technology stack, it is a contract with a company that has shareholders, and it can change under a project that is already three years in. Read the terms, keep an exit plan, and prefer terms fixed at the version you ship.
11. **Unreal Engine** licensing has moved the other way over time. In March 2014 Unreal Engine 4 was 19 dollars per month plus a 5 percent royalty. In March 2015 it became free with a 5 percent royalty above 3,000 dollars per calendar quarter. In May 2020, retroactive to 1 January 2020, the exemption became 1,000,000 dollars of lifetime gross revenue per title. Revenue from the Epic Games Store is royalty-free. Since 2024 there is also a seat licence, 1,850 dollars per seat per year, for companies over 1 million dollars doing non-game work such as film and architecture. Unreal Engine 5.7 shipped in November 2025.
12. **Godot** is MIT licensed and has been open source since February 2014. There is no fee, no royalty and no threshold, and the licence cannot be revoked for a version you already have. The latest stable release at the time of writing is 4.7.1 from 13 July 2026. It is developed by the Godot Foundation on donations, which is its own risk: a much smaller budget than either commercial competitor.
13. **In-house engines** still dominate the top end: RE Engine at Capcom, Decima at Guerrilla, Frostbite at Electronic Arts, Anvil at Ubisoft, id Tech at id Software, Creation at Bethesda, Source 2 at Valve, and Rockstar's RAGE. The reason is usually a genre the commercial engines handle badly, plus twenty years of tooling that a team already knows.
14. Where experts disagree: some argue in-house engines are a strategic asset; others argue they are a tax that diverts engineers from the game. Several large publishers have moved to Unreal Engine 5 since 2022, which is evidence for the second view but not proof.

45.5.6 WORDS - remember these

1. Engine — the reusable machinery — the platform layer, renderer, physics, audio, tools and pipeline shared across titles.
2. Scene graph — a tree of objects — a hierarchy of transforms where child transforms are expressed relative to their parent.
3. ECS — data in flat arrays with functions over them — entity-component-system architecture separating identity, plain data and behaviour for cache locality.
4. Cache line — the smallest chunk memory moves in — 64 bytes on current x86-64 and ARM64 processors.
5. Royalty — a share of revenue paid to the engine maker — a percentage of gross above a stated threshold, as in Unreal's 5 percent above 1 million.
6. Runtime fee — a charge per install — the per-installation model Unity announced in September 2023 and cancelled in September 2024.

45.6 Assets and the content pipeline

45.6.1 PLAIN - in simple words

1. Everything in a game that is not code is an **asset**: models, pictures, sounds, animations, fonts, levels, text.
2. Assets are made in other programs. A model is made in a modelling tool, a texture in a painting tool, a sound in an audio tool.
3. Those programs save in formats designed for editing, not for playing. They are big, flexible and slow to read.
4. So there is a step in the middle that converts everything into the exact form the target machine wants. That step is the **asset pipeline**.
5. It compresses textures into a format the graphics chip can read directly. It simplifies models into several versions for different distances. It packs thousands of small files into a few big ones.
6. It also computes things in advance that would be too slow to compute while playing: how light falls on a static wall, where characters can walk, how sound travels in a room.
7. Doing all that for an entire game is why a full build takes hours.
8. In Unreal Engine this conversion step is called **cooking**, and the phrase "cook the content for PlayStation" means "convert every asset into the exact form that machine wants".

45.6.2 PLAIN - a picture in your head

1. Think of a professional kitchen before service.
2. The raw ingredients arrive as whole vegetables, whole fish, sacks of flour. Flexible, but useless during a busy service.
3. So in the morning the kitchen does **prep**: chopping, portioning, making stocks, part-cooking things that take an hour.
4. During service nothing takes an hour, because the hour already happened.
5. Prep is the asset pipeline. Raw ingredients are the source files. The prepped trays are the cooked assets.
6. And if the chef changes the menu at 5 p.m., all the prep for that dish is wasted and must be redone. That is why artists hate late changes.

Where this comparison breaks: a kitchen preps for one evening. A game preps for every machine it will ever run on, and each target wants different prep, so the same source is prepped four or five times in parallel.

45.6.3 PLAIN - a worked example

1. Follow one texture from painting to screen.
2. An artist paints a 4096 by 4096 colour map in a painting program and saves a 32-bit-per-pixel file.
3. Raw size: 4096 x 4096 x 4 bytes = 67,108,864 bytes, which is 64 mebibytes for one texture.
4. The pipeline generates **mipmaps**, smaller copies at half size, quarter size and so on down to one pixel. That adds about 33 percent, so about 85 mebibytes.
5. Then it compresses to a format the graphics chip can sample directly. On PC and console that is usually BC7, which is one byte per pixel.
6. BC7 size: 4096 x 4096 x 1 = 16 mebibytes, plus mipmaps, about 21.3 mebibytes. That is a quarter of the raw size and it stays compressed in video memory, which is the point.
7. On a phone the same texture is compressed to ASTC instead, because mobile graphics chips do not read BC7.

Format	Target	Bits per pixel
RGBA8 uncompressed	none	32
BC1 (DXT1)	PC, console	4

Format	Target	Bits per pixel
BC7	PC, console	8
ETC2 RGBA	Android, GLES 3	8
ASTC 4x4	mobile, Apple	8
ASTC 6x6	mobile, Apple	3.56

8. Now do that for 4,000 textures. At 64 mebibytes raw each, the source is 256 gibibytes; cooked to BC7 with mipmaps it is about 85 gibibytes; after packaging and further lossless compression, perhaps 45 gibibytes on disk.
9. And the compression itself is not free. High-quality BC7 encoding is roughly 0.5 to 5 seconds per 4K texture on one core. Four thousand of them is 30 minutes to 5 hours of single-core work, which is why studios run distributed build farms and shared caches.

45.6.4 PLAIN - what is really happening inside

1. The full pipeline, in order, for a 3D character.
2. **Model**: the artist builds a mesh, a set of points joined into triangles. A hero character might be 60,000 to 150,000 triangles.
3. **UV unwrap**: the artist flattens the surface onto a square so a two-dimensional picture can be painted onto it.
4. **Texture**: several pictures are painted, not one. Base colour, roughness, metalness, a normal map that fakes small bumps, sometimes ambient occlusion and emissive.
5. **Rig**: a skeleton of bones is built inside the mesh, and each vertex is assigned weights saying which bones move it and how much.
6. **Animate**: the animator poses the skeleton over time, or motion capture data is cleaned up and retargeted onto it. Each clip is a list of bone rotations at keyframes.
7. **Export**: everything is written to an interchange format, usually FBX, or increasingly glTF 2.0 or USD.
8. **Import**: the engine reads that file and converts it. Triangles are reordered for the graphics chip's vertex cache. Normals and tangents are computed. Animation curves are compressed, often to about a tenth of their size, by dropping keys within an error tolerance.
9. **Level of detail**: simplified copies are generated. A common ladder is 100 percent, 50 percent, 25 percent and 12 percent of the triangles, swapped by screen size.
10. **Bake**: things that never change are computed once. Light maps for static geometry, a navigation mesh for where characters can walk, occlusion data for what can see what.
11. **Package**: everything is written into a few large archive files, sorted so that things loaded together sit next to each other on disk, and compressed with a fast codec.
12. **Ship**: the archive plus the executable is what the player downloads.

45.6.5 TECHNICAL - the engineer's version

1. Tools in common use: Blender (free, GPL) and Autodesk Maya for modelling and animation; ZBrush for sculpting; Adobe Substance 3D Painter and Designer for textures; Houdini for procedural content and effects; Photoshop or Krita for two-dimensional art; Reaper, Pro Tools and iZotope RX for audio.
2. Interchange formats: FBX (Autodesk, proprietary but ubiquitous), glTF 2.0 (Khronos, 2017, open and runtime-friendly), USD (Pixar, open-sourced 2016, now standard for large scene assembly), Alembic for baked geometry caches, OBJ for simple static meshes.
3. Cooking, precisely, means: convert every source asset into a platform-specific, engine-specific binary that can be memory-mapped or streamed with minimal parsing, and drop all editor-only data.
4. Why full builds take hours, itemized for a large project:

Stage	Typical duration
Code compile and link	20 to 90 minutes
Shader compilation	1 to 6 hours
Texture compression	30 to 180 minutes
Lightmap baking	1 to 12 hours
Packaging and compress	20 to 90 minutes

5. Shader compilation is usually the worst offender because of **permutations**. One material shader with 12 independent boolean features is 2 to the power 12, which is 4,096 variants, and a real project has hundreds of such shaders. Large titles ship tens of thousands to hundreds of thousands of compiled pipeline states.
6. The countermeasures are a **derived data cache** (a shared server storing the result of every conversion keyed by a hash of the input plus the settings, so nobody converts the same asset twice), distributed compilation systems such as Incredibuild or FASTBuild, and incremental cooking that touches only changed assets. With a warm cache, an incremental cook is minutes rather than hours.
7. Runtime compression codecs matter as much as the format. Oodle Kraken, Zstandard and LZ4 are the common choices; consoles decompress in hardware, with the PlayStation 5 Kraken decompressor rated at roughly 5.5 gigabytes per second of compressed input and the Xbox Series X BCPack path similar in spirit.
8. Virtualized geometry changes the level-of-detail story. Unreal Engine 5's Nanite, introduced in 2021, stores meshes as a hierarchy of triangle clusters and selects detail per cluster per pixel at run time, removing hand-authored LODs for static opaque geometry. It does not remove the need for LODs on skinned characters, foliage or translucent materials, which is a caveat often skipped in marketing material.
9. Virtual texturing solves the matching problem for textures: only the pages of a texture actually visible at the current screen resolution are resident in video memory, so a scene can reference far more texture data than fits.
10. Observation tools: Unreal's `-run=DerivedDataCache` and cook logs, Unity's Build Report and Addressables analysis, `ls -la` on the packaged archives, RenderDoc to see which textures a frame actually sampled, and the platform's own memory profiler to see what is resident.

45.6.6 WORDS - remember these

1. Asset — anything that is not code — content data authored in an external tool and consumed by the engine.
2. Cooking — converting content for one machine — transforming source assets into a platform-specific runtime binary form.
3. Mipmap — smaller copies of a texture — a precomputed pyramid of half-resolution levels used to avoid aliasing at distance.
4. Level of detail — simpler versions for far away — a set of decimated meshes selected by projected screen size.
5. Baking — computing it in advance — precomputing lighting, navigation or occlusion data at build time rather than at run time.
6. Derived data cache — a shared store of conversion results — a server keyed by input hash so that each conversion is performed once for the whole team.
7. Shader permutation — one variant of a shader — a compiled pipeline state for one specific combination of features and platform.

45.7 Physics and collision

45.7.1 PLAIN - in simple words

1. A physics system answers two questions, sixty times a second: where is everything now, and did anything touch anything.
2. Each movable object is a **rigid body**: a shape that does not bend, with a mass, a position, a speed and a spin.
3. Forces such as gravity change its speed. Its speed changes its position. Doing that arithmetic step by step is called **integration**.
4. Finding touches is done in two passes, because checking every pair is far too slow.
5. The first pass, the **broad phase**, cheaply throws away pairs that are obviously nowhere near each other, using a grid or a tree of boxes.
6. The second pass, the **narrow phase**, does the real shape test on the few pairs that survived.
7. When two things do touch, the system pushes them apart and changes their speeds. How bouncy that is depends on a number between 0 and 1 called **restitution**.
8. Ropes, hinges, doors and ragdolls are the same machinery plus rules that say "these two bodies must stay joined here". Those rules are constraints.

45.7.2 PLAIN - a picture in your head

1. Imagine a large hall with 1,000 people walking around, and you must report every collision.
2. Checking every pair means 1,000 x 999 / 2 = 499,500 checks. At sixty times a second that is 30 million checks a second, for nothing.
3. So you chalk the floor into a grid of squares. Each person reports which square they are in.
4. Now you only compare people in the same square or in touching squares. Almost every pair is eliminated without looking at them.
5. That is the broad phase. The remaining few hundred pairs get the careful look, which is the narrow phase.

Where this comparison breaks: people are all about the same size, so one grid square size works. Game objects range from a bullet to a cathedral, and one grid size cannot serve both. Real engines therefore use trees of boxes that adapt, not a flat grid.

45.7.3 PLAIN - a worked example

1. Two balls hit head-on. Ball A weighs 2 kg and moves at +5 metres per second along x. Ball B weighs 1 kg and is still. Restitution is 0.5.
2. The collision normal `n` points from A to B, so it is $(1, 0, 0)$.
3. Relative velocity along the normal: `v_rel = (v_B - v_A) . n = 0 - 5 = -5`. Negative means they are approaching, so we must respond.
4. Impulse magnitude uses this formula:

```
      -(1 + e) * v_rel
j = --------------------
       1/m_A + 1/m_B
```

5. Put the numbers in: `j = -(1 + 0.5)(-5) / (1/2 + 1/1) = 7.5 / 1.5 = 5`. The unit is kilogram metres per second.
6. Apply it: `v_A' = 5 - j/m_A = 5 - 5/2 = 2.5 m/s`, and `v_B' = 0 + j/m_B = 0 + 5/1 = 5.0 m/s`.
7. Check momentum. Before: 2 x 5 + 1 x 0 = 10. After: 2 x 2.5 + 1 x 5 = 10. Conserved, as it must be.
8. Check restitution. Separation speed after is 5 - 2.5 = 2.5, and the approach speed was 5. The ratio is 0.5, which is exactly `e`.
9. Check energy. Before: 0.5 x 2 x 25 = 25 joules. After: 0.5 x 2 x 6.25 plus 0.5 x 1 x 25 = 6.25 + 12.5 = 18.75 joules. 6.25 joules were lost as heat and sound, which is what a restitution below 1 means.

45.7.4 PLAIN - what is really happening inside

1. The integration step. The naive version, called explicit Euler, is `x = x + v*dt` then `v = v + a*dt`. It quietly adds energy on every step and a spring simulated this way will slowly explode.
2. Games use **semi-implicit Euler** instead: update velocity first, then use the new velocity to update position. One line swapped, and the system is stable enough for gameplay.
3. **Verlet integration** stores the previous position instead of velocity: `x_new = 2*x - x_prev + a*dt*dt`. It handles distance constraints very well, which makes it the standard choice for cloth, rope and hair.
4. The broad phase in a real engine is usually a bounding volume hierarchy: a tree where each node is a box containing its children's boxes. Testing a ray or a box against it costs about log of the object count.
5. The narrow phase uses exact tests for simple pairs, such as sphere against sphere, and a general algorithm for convex shapes.
6. Contacts are not resolved one at a time in isolation, because a box resting on the floor has four contacts that must agree. The solver runs over all contacts several times per step, nudging each towards consistency.
7. That is why a tall stack of boxes in a game jitters. The solver has a fixed number of iterations and simply runs out of time before the stack agrees.

45.7.5 TECHNICAL - the engineer's version

1. Broad-phase structures in use: uniform grids and spatial hashes for same-size objects; sweep and prune, which keeps objects sorted along an axis and exploits frame-to-frame coherence; quadtrees and octrees; and dynamic AABB trees, which is what Box2D, Bullet, PhysX and Jolt all use.
2. Narrow phase: analytic tests for sphere, capsule and axis-aligned box pairs; the separating axis theorem for oriented boxes and convex polygons; and GJK, published by Gilbert, Johnson and Keerthi in 1988, plus the expanding polytope algorithm to recover penetration depth, for general convex shapes. Concave meshes are decomposed into convex pieces or treated as static triangle soups.
3. Constraint solving is almost always **sequential impulses**, also called projected Gauss-Seidel, popularized by Erin Catto in Box2D and his GDC presentations from 2005 onward. Typical settings are 4 to 10 velocity iterations and 1 to 4 position iterations per step. Baumgarte stabilization or a split-impulse pass removes penetration without injecting energy.
4. Joint types: hinge (revolute), ball and socket (spherical), slider (prismatic), fixed, distance, and six-degree-of-freedom with per-axis limits and motors. A **ragdoll** is a set of rigid bodies matched to the skeleton's bones with joints whose limits mimic human range of motion, plus a blend that hands control from animation to physics on death or impact.
5. Continuous collision detection exists because a bullet moving 600 metres per second travels 10 metres in a 16.67 ms step and will tunnel straight through a wall. Solutions are conservative advancement, swept shape tests, or simply raycasting bullets instead of simulating them, which is what most shooters do.
6. Common engines and their defaults:

Engine	Physics	Default rate
Unity 6	PhysX 4 or Unity Physics	50 Hz
Unreal Engine 5	Chaos	30 or 60 Hz
Godot 4	Godot Physics or Jolt	60 Hz
Custom	Bullet, Jolt, Box2D	60 Hz

7. Determinism, stated precisely. IEEE 754-2019 fully specifies addition, subtraction, multiplication, division and square root: given the same inputs, rounding mode and precision, every conforming machine returns the same bits.
8. It does not specify `sin`, `cos`, `exp`, `log` or `pow`, so two maths libraries can legally differ in the last bit. That is the first source of divergence.

9. The others are: compilers reordering floating-point expressions or contracting `a*b+c` into a fused multiply-add with different rounding; fast-math flags; historic x87 80-bit intermediates versus SSE 64-bit; different SIMD widths taking different code paths; and multi-threaded solvers where the order contacts are summed changes between runs.
10. The consequence for multiplayer: you cannot assume two machines simulating the same inputs reach the same state. A one-bit difference in one frame becomes a visibly different world within seconds, because contact resolution amplifies small differences.
11. If you need determinism, you must either use fixed-point arithmetic, or lock down compiler flags, library versions, thread counts and instruction sets across every platform you ship on. This is why deterministic lockstep networking, described in section 45.11, is hard.

45.7.6 WORDS - remember these

1. Rigid body — a solid that does not bend — a body with mass, inertia tensor, linear and angular velocity, and no internal deformation.
2. Integration — stepping the maths forward — advancing state by `dt` using a scheme such as semi-implicit Euler or Verlet.
3. Broad phase — the cheap first filter — spatial culling that produces a candidate pair list, typically via a dynamic AABB tree.
4. Narrow phase — the careful shape test — exact intersection tests producing contact points, normals and penetration depths.
5. Restitution — how bouncy it is — the coefficient `e` relating separation speed to approach speed along the contact normal.
6. Impulse — an instant change of momentum — the quantity `j` applied along the contact normal to resolve a collision.
7. Tunnelling — moving straight through a wall — a fast body crossing thin geometry within one timestep, fixed by continuous collision detection.

45.8 Game AI

45.8.1 PLAIN - in simple words

1. Game AI is the code that decides what a character that is not you does next.
2. It has two halves: **decide what to do**, and **work out how to get there**.
3. The simplest way to decide is a **state machine**. The guard is in one state at a time: idle, patrol, investigate, chase, attack, flee. Events move it between states.
4. When there are twenty states, the arrows between them become unmanageable, so bigger games use a **behaviour tree**: a tree of small tasks read from the top, left to right, until one succeeds.
5. Another way is a **utility system**: score every possible action with a number and do the highest-scoring one.
6. The most elaborate is a **planner**: describe the goal, describe what each action requires and produces, and let the machine search for a sequence.
7. The second half, getting there, is **pathfinding**, and the standard algorithm has been the same since 1968.

45.8.2 PLAIN - a picture in your head

1. Think about how you decide what to do at work.
2. Some of it is a state machine: you are in a meeting, or at your desk, or at lunch, and only certain moves make sense from each.
3. Some of it is a checklist read top down: is anything on fire, no; is anyone waiting for me, no; then do the next task on the list. That is a behaviour tree, and the order encodes priority.

4. Some of it is scoring: three things could be done, and you weigh urgency against effort and pick one. That is a utility system.
5. And occasionally you plan: to get the report signed you need the data, which needs the export, which needs access. Working backwards from the goal is what a planner does.
6. All four are used in games, often in the same game, for different characters.

Where this comparison breaks: you have a goal of your own. A game character does not. Its goal is that you enjoy fighting it, which is not the same thing as winning, and that changes every design decision underneath.

45.8.3 PLAIN - a worked example

1. A-star pathfinding, by hand, on a five by five grid. `S` is the start, `G` is the goal, `#` is a wall. Moves are up, down, left and right, each costing 1.

```
y=4  .  .  .  .  .
y=3  .  .  #  .  .
y=2  S  .  #  .  G
y=1  .  .  #  .  .
y=0  .  .  .  .  .
     x0 x1 x2 x3 x4
```

2. A-star keeps a score for each square: `f = g + h`.
3. `g` is the real cost from the start to that square, counted so far.
4. `h` is a guess of the cost from that square to the goal. Here it is the Manhattan distance, `|4 - x| + |2 - y|`, which never overestimates because you cannot reach the goal in fewer steps than that.
5. Always expand the open square with the lowest `f`.
6. Start: S at (0,2) has `g=0`, `h=4`, `f=4`.
7. Expand S. Neighbours (0,1) and (0,3) get `g=1, h=5, f=6`. Neighbour (1,2) gets `g=1, h=3, f=4`.
8. Lowest `f` is (1,2) at 4, so expand it. (2,2) is a wall. (1,1) and (1,3) get `g=2, h=4, f=6`.
9. Now four squares are tied at `f=6`. Ties are broken towards higher `g`, which pushes the search forwards rather than sideways.
10. Expanding those gives (1,0) and (1,4) at `f=8`, and later (0,0) and (0,4) at `f=8`.
11. From (1,4) the search runs along the top: (2,4) `g=4 h=4 f=8`, (3,4) `g=5 h=3 f=8`, (3,3) `g=6 h=2 f=8`, (3,2) `g=7 h=1 f=8`, and finally the goal at `g=8, h=0, f=8`.
12. The answer is an eight-step path around the top of the wall. Note the heuristic said 4 at the start and the truth was 8. That is allowed, and that gap is exactly why the search had to look around.

Algorithm	Squares expanded	Path length
Breadth-first / Dijkstra	~22 of 22	8
A-star, Manhattan h	~13 of 22	8
Greedy best-first	~9 of 22	may exceed 8

13. A-star found the same shortest path as Dijkstra while looking at about forty percent fewer squares. Greedy search looks at fewer still but can return a worse path, because it ignores the cost already paid.

45.8.4 PLAIN - what is really happening inside

1. A-star holds two collections: an **open set** of squares discovered but not yet expanded, and a **closed set** of squares already expanded.
2. The open set is a priority queue ordered by `f`, so taking the best square is fast.
3. Each square records which square it came from. When the goal is reached, you follow those links backwards to build the path.

4. Grids are convenient for teaching and terrible for large 3D worlds, because the number of squares grows with area and the paths look like staircases.
5. Real 3D games use a **navigation mesh**: the walkable floor is covered with a few thousand convex polygons instead of a million squares.
6. A-star then runs over polygons, and a second step called the funnel algorithm pulls the resulting corridor tight into a straight-line path.
7. Finally, **steering** turns that path into motion: the character seeks the next corner, slows as it arrives, avoids other characters, and blends with its animation so the feet do not slide.

45.8.5 TECHNICAL - the engineer's version

1. A-star was published in 1968 by Peter Hart, Nils Nilsson and Bertram Raphael as "A Formal Basis for the Heuristic Determination of Minimum Cost Paths" in IEEE Transactions on Systems Science and Cybernetics. It is optimal if the heuristic is **admissible**, meaning it never overestimates, and needs no node reopening if the heuristic is also **consistent**.
2. Heuristics by grid type: Manhattan for four-way movement; octile, `dx + dy + (sqrt(2) - 2) * min(dx, dy)`, for eight-way; Euclidean for navigation meshes. Using Euclidean on a four-way grid is admissible but weaker, so the search expands more nodes.
3. Variants shipped in real games: hierarchical A-star over a coarse graph then a fine one; jump point search for uniform grids; theta-star for any-angle paths; D-star Lite for replanning in changing worlds; and flow fields, where one Dijkstra pass from the goal produces a direction for every cell, which is how real-time strategy games move 500 units at once without 500 searches.
4. The standard open-source navigation toolkit is Recast and Detour by Mikko Mononen: Recast voxelizes the level and extracts walkable convex polygons; Detour does the queries, path corridors and local avoidance. Unity and Unreal both ship navigation mesh baking built on the same ideas.
5. Decision architectures with their shipped examples: hierarchical finite state machines everywhere; behaviour trees popularized by Damian Isla's GDC 2005 talk on the Halo 2 AI; utility scoring used in The Sims from 2000; and goal-oriented action planning described in Jeff Orkin's GDC 2006 paper "Three States and a Plan: The A.I. of F.E.A.R.", where the state machine has only three states and a planner sequences them at run time.
6. Steering behaviours come from Craig Reynolds. His 1987 SIGGRAPH paper "Flocks, Herds, and Schools: A Distributed Behavioral Model" introduced boids with three rules: separation, alignment and cohesion. His 1999 GDC paper added seek, flee, arrive, pursue, evade, wander, obstacle avoidance and path following. Flocking that looks alive is those three rules and nothing else.
7. Now the honest part. Game AI is not optimizing for winning. It is optimizing for the player's experience, and the two are usually opposed. A perfect aimbot enemy is trivial to write and miserable to play against.
8. Deliberate handicaps that ship in real games: enemies miss the first one or two shots on purpose; only one or two enemies attack at a time while the rest circle, a pattern used widely in third-person action games; attacks have long telegraph animations so you can react; enemies announce their intentions out loud, which is a user interface disguised as dialogue; vision cones and hearing radii are far worse than a human's; racing games rubber-band opponents to keep the pack close.
9. The Pac-Man ghosts from 1980 remain the best teaching example. Four ghosts with four short, deterministic targeting rules read as four personalities, and none of them contains anything a modern programmer would call intelligence.
10. Separating fact from marketing: established fact is that shipped game AI is overwhelmingly hand-authored logic. Active research is reinforcement learning agents such as OpenAI Five for Dota 2 in 2018 and DeepMind's AlphaStar for StarCraft II in 2019, which beat strong humans but are expensive to train, hard to tune and generally not fun opponents. Marketing claim is "AI-powered characters" in trailers, which in 2026 almost always means a language model generating dialogue lines, not the combat behaviour.

45.8.6 WORDS - remember these

1. Finite state machine — one mode at a time — a set of states with guarded transitions, the simplest decision architecture.
2. Behaviour tree — a prioritized checklist as a tree — composites, decorators and leaf tasks ticked from the root each evaluation.
3. Utility AI — score everything and pick the best — action selection by evaluating scoring curves over world variables.
4. GOAP — describe the goal, let it plan — goal-oriented action planning, an A-star search over world states using action preconditions and effects.
5. Heuristic — an educated guess of remaining cost — the h term in A-star, admissible if it never overestimates.
6. Navigation mesh — the walkable floor as polygons — a convex polygon decomposition of traversable surfaces used as the pathfinding graph.
7. Steering — turning a path into movement — force-based local motion such as seek, arrive, separation and obstacle avoidance.

45.9 Rendering a game frame, specifically

45.9.1 PLAIN - in simple words

1. In one line, the drawing pipeline is: points in space, moved into camera view, turned into triangles, filled in as pixels, shaded, depth-tested and written to a picture.
2. A game adds a lot of decisions around that, because it must do all of it in under 16.67 milliseconds.
3. First it works out what it does not need to draw at all. That is **culling**. Anything behind you or outside the camera's cone is thrown away immediately, and anything fully hidden behind a wall as well.
4. Second it groups what is left so the processor talks to the graphics chip as few times as possible. Each of those conversations is a **draw call** and they are expensive on the processor side.
5. Third it sorts. Solid things are drawn nearest first, so far-away pixels are rejected cheaply. See-through things must be drawn furthest first, so the blending is right.
6. Then it draws shadows, then the world, then the see-through things, then a stack of full-screen effects, and finally the user interface on top.

45.9.2 PLAIN - a picture in your head

1. Think of a stage crew setting up a theatre scene in one minute.
2. First they check the script: what is actually visible from where the audience sits. Everything else stays in the wings. That is culling.
3. Then they carry things on in loads rather than one item at a time, because every trip costs time regardless of what is on it. That is batching.
4. They set the lights before the actors walk on, because moving lights during the scene is expensive. That is the shadow pass.
5. Glass and gauze go on last and in the right order, because you can see through them and the order changes what you see.
6. And the surtitles are projected on top of everything at the end, in their own crisp resolution. That is the user interface.

Where this comparison breaks: the stage crew works once and the scene runs. The renderer redoes the entire setup from nothing, sixty times a second, with the camera in a new place each time.

45.9.3 PLAIN - a worked example

1. A real frame breakdown, taken from a profiler on a mid-range PC at 2560 by 1440 with a 60 frames per second target. The numbers are typical rather than from one specific game.

Stage	Time (ms)	Share
Shadow map passes	2.4	14 percent
Depth prepass and G-buffer	3.1	19 percent
Lighting and reflections	3.8	23 percent
Transparency and effects	1.6	10 percent
Post-processing stack	2.7	16 percent
Upscale and interface	1.1	7 percent
Present and slack	1.9	11 percent

2. Total is 16.6 ms, just inside the 16.67 ms budget.
3. Read the first row again. Shadows cost 2.4 ms, which is more than the entire user interface and post-processing of many older games, and they are just one lighting feature.
4. This is why the graphics settings menu exists. Dropping shadow resolution from 2048 to 1024 per cascade typically saves 30 to 50 percent of that row.

45.9.4 PLAIN - what is really happening inside

1. **Frustum culling** tests each object's bounding sphere or box against the six planes of the camera's viewing pyramid. A sphere fully outside any one plane is rejected. This is a handful of arithmetic operations per object.
2. **Occlusion culling** removes what is hidden behind other things. It is harder, because you must know what is in front before you know what is hidden. Common approaches are precomputed visibility per region, a low-resolution software depth buffer rendered on the processor, and graphics-chip queries whose answers arrive a frame or two late.
3. **Batching** merges objects that share a material into one draw call. **Instancing** goes further: send one mesh and a list of 5,000 positions, and the chip draws all 5,000 in one call.
4. **Sorting** solid objects front to back lets the depth test discard hidden pixels before shading them. On modern hardware a depth prepass, which draws depth only and then shades with the depth test set to equal, guarantees every pixel is shaded exactly once.
5. **Forward rendering** shades each object against every light that touches it. Cost grows as objects times lights. It handles transparency and hardware anti-aliasing naturally.
6. **Deferred rendering** first writes surface properties into several full-screen buffers, together called the **G-buffer**, then computes lighting once per screen pixel. Cost is decoupled from scene complexity, which is why it won for scenes with many lights.
7. Deferred rendering's costs are memory bandwidth and transparency. See-through surfaces cannot be written into a single G-buffer pixel, so they are drawn afterwards in a separate forward pass.

45.9.5 TECHNICAL - the engineer's version

1. G-buffer bandwidth arithmetic. At 1920 by 1080 with four 32-bit render targets plus a 32-bit depth buffer, that is 5 x 4 bytes x 2,073,600 pixels, about 41.5 megabytes written per frame, or 2.5 gigabytes per second at 60 frames per second, before the lighting pass reads it all back.
2. At 3840 by 2160 the same layout is about 166 megabytes per frame and about 10 gigabytes per second of writes. This is why 4K deferred rendering is a bandwidth problem before it is a compute problem.
3. Modern engines mostly use clustered or tiled deferred and forward variants, which bin lights into screen tiles or view-space clusters so each pixel only evaluates the lights that reach it. Unreal Engine 5 and Unity's High Definition Render Pipeline both do this.
4. Draw call budgets. Under Direct3D 11 the per-call driver cost made 1,000 to 3,000 calls per frame a common ceiling. Direct3D 12, Vulkan and Metal cut that cost sharply, and with instancing, bindless resources and GPU-driven indirect draws, 10,000 to 30,000 batched calls per frame is normal in 2026.
5. Transparency ordering has no cheap correct answer. Per-object sorting fails for intersecting or concave transparent geometry. The options are alpha testing, which is opaque and sortable; per-triangle

sorting, which is expensive; depth peeling, which costs one pass per layer; or weighted blended order-independent transparency, published by McGuire and Bavoil in 2013, which is approximate but cheap and widely shipped.

6. Shadows in practice are cascaded shadow maps for the sun: the view frustum is split into 3 or 4 depth ranges, each rendered from the light into a 1024 to 2048 square depth texture. Each cascade is another full scene traversal of shadow casters. Point lights need six faces. Filtering ranges from simple percentage-closer filtering to variance and moment methods.
7. A typical post-processing stack, in execution order: depth of field, motion blur, bloom, auto-exposure, tone mapping (commonly an ACES curve), colour grading through a 3D lookup texture, temporal anti-aliasing, upscaling, sharpening, then film grain and vignette.
8. Upscalers such as DLSS, FSR and XeSS are inserted after the main scene and before the user interface, so the interface renders at native output resolution and stays sharp. Getting that order wrong makes text blurry, and it is a common bug in early builds.
9. The interface is drawn last, usually as batched textured quads from a texture atlas with a signed distance field font, in a separate orthographic pass, sometimes at a different resolution from the scene.
10. Capture and profiling tools: RenderDoc for a full frame capture on almost any API; PIX on Windows for Direct3D 12 timing; NVIDIA Nsight Graphics and Radeon GPU Profiler for hardware counters; Xcode's Metal debugger on Apple platforms; Unity's Frame Debugger; Unreal's `stat gpu`, `stat unit` and `ProfileGPU` console commands.

45.9.6 WORDS - remember these

1. Frustum culling — skip what is off-screen — rejecting bounding volumes against the six camera planes.
2. Occlusion culling — skip what is hidden — rejecting objects proven to be behind other opaque geometry.
3. Draw call — one instruction to draw something — a submitted command with a bound pipeline state, costing processor time regardless of triangle count.
4. Instancing — one command, many copies — drawing N copies of a mesh from one call with per-instance data.
5. G-buffer — the sheet of surface facts — the set of render targets holding albedo, normal, roughness, metalness and depth in deferred rendering.
6. Deferred rendering — light once per pixel, not per object — decoupling shading from geometry by writing surface data first.
7. Post-processing — the full-screen effects at the end — the ordered chain from depth of field through tone mapping to anti-aliasing and upscaling.

45.10 Audio in games

45.10.1 PLAIN - in simple words

1. Sound is a pressure wave in air, and a computer stores it as a long list of numbers, typically 48,000 numbers per second per channel.
2. A game does not play files. It plays **events**: "footstep on gravel", "pistol fire", "door creak". One event may pick one of eight recordings at random and change its pitch slightly so it does not sound repetitive.
3. Those recordings are packed into **sound banks**, loaded and unloaded in groups, usually one bank per level or per character.
4. A sound in the world has a position. The mixer works out how loud it should be from the distance, and how much goes to each speaker from the angle.
5. If a wall is in the way, the game fires a ray from the source to your ears, and if it is blocked, it turns the volume down and removes the high frequencies, because that is what walls do to sound.
6. All the sounds are grouped into **buses**: music, effects, dialogue, interface. Each bus has its own volume, and rules such as "duck the music by 6 decibels while anyone is speaking".

7. Music in a modern game is not one track. It is layers that fade in as danger rises, and blocks that swap at musical boundaries.

45.10.2 PLAIN - a picture in your head

1. Imagine a live sound engineer at a theatre with a mixing desk.
2. Every microphone and every playback source is a channel on the desk. Channels are grouped into buses: cast, orchestra, effects.
3. She rides the faders as the scene changes, pulls the music down when somebody speaks, and adds reverb when the scene moves to a cathedral.
4. She also has a limit: the desk has 64 channels. If a 65th sound is needed, something quiet must be dropped.
5. A game audio engine is that desk, run by rules instead of a person, updated sixty times a second.

Where this comparison breaks: the theatre engineer knows the script. The game engine does not know what will happen next, so every rule must be written to handle any order of events, including 200 sounds at once.

45.10.3 PLAIN - a worked example

1. Audio memory arithmetic, which is what decides whether a sound is kept decompressed in memory or streamed from disk.
2. Uncompressed 48 kHz, 16-bit stereo is 48,000 x 2 x 2 = 192,000 bytes per second. One minute is 11.5 megabytes. One hour is 691 megabytes.
3. The same audio in Vorbis at 128 kilobits per second is 16,000 bytes per second: 0.96 megabytes per minute, 57.6 megabytes per hour, a saving of about 12 to 1.
4. So the rule of thumb is: short and frequent sounds are decompressed into memory once and played from there; long music and dialogue are streamed from storage and decoded on the fly.

Content	Typical handling	Cost
Footsteps, gunfire	Decompressed in RAM	~50 KB each
Ambience loops	Compressed in RAM	~1 MB each
Music tracks	Streamed from disk	2 buffers
Voice lines	Streamed, bank per scene	2 buffers

5. A console game commonly budgets 100 to 500 megabytes of memory for audio, which is a real constraint against a script with 40,000 recorded lines.

45.10.4 PLAIN - what is really happening inside

1. The audio system does not run on the frame clock. It runs on its own high-priority thread with its own deadline.
2. That deadline is the buffer size. A 512-sample buffer at 48 kHz must be filled every 512 / 48,000 seconds, which is 10.67 milliseconds. A 256-sample buffer is 5.33 milliseconds.
3. If the audio thread misses that deadline, the hardware plays whatever was in the buffer, and you hear a click or a stutter. There is no equivalent of a dropped frame that merely looks bad.
4. This is why audio must never wait for the game. The game posts events into a queue, and the audio thread reads them when it wakes.
5. Each buffer fill is: gather the active voices, decode any compressed ones, apply per-voice volume, pitch and filters, sum them into buses, apply bus effects such as reverb and compression, and write the result out.
6. Voice limits exist because that work is per voice. A budget of 32 to 128 simultaneous voices is normal, and beyond it the quietest and most distant sounds are stolen.

45.10.5 TECHNICAL - the engineer's version

1. Distance attenuation. The physical law is inverse square, which is minus 6 decibels per doubling of distance. Games rarely use it directly; they use authored curves with a minimum distance, inside which volume is constant, and a maximum distance, beyond which the sound is culled entirely.
2. Occlusion versus obstruction: **occlusion** is a full barrier between source and listener and attenuates both the direct and reverberant paths; **obstruction** blocks the direct path only while sound still arrives via reflections. Both are usually implemented as raycasts driving a low-pass filter cutoff and a gain.
3. Spatialization ranges from cheap stereo panning, through vector-based amplitude panning for surround layouts, to head-related transfer function convolution for binaural headphone output. Shipping implementations include Sony's Tempest 3D AudioTech on PlayStation 5, Dolby Atmos, Windows Sonic and Valve's Steam Audio.
4. Adaptive music has two standard techniques. **Vertical layering** plays several stems at once and crossfades them with game intensity. **Horizontal re-sequencing** switches between composed blocks at musical boundaries, using short transition segments so the change lands on a beat.
5. Middleware exists because building all of this well takes years. The two dominant products are FMOD Studio from Firelight Technologies and Wwise from Audiokinetic, which Sony acquired in 2019. Both give composers and sound designers an editor that produces data the game loads, so audio changes do not require a programmer or a rebuild.
6. Wwise pricing per title, from the published rate card, as an example of how middleware is sold:

Tier	Budget cap	First platform
Indie	250k USD	Free
Pro	2M USD	7,000 USD
Premium	none	22,000 USD
Platinum	none	40,000 USD

7. Additional platforms cost extra on the paid tiers, which is why a small team shipping on five platforms must budget for audio middleware as a real line item rather than an afterthought.
8. Streaming considerations: each streamed voice needs at least two buffers so one can be filled while the other plays, plus a seek-and-decode budget. On consoles with a hardware decompressor, audio streaming competes with texture streaming for the same input-output bandwidth, and audio must win, because a late texture is a blurry wall and a late sound is a click.
9. Diagnostic tools: the FMOD Studio and Wwise live profilers connect to a running game and show every active voice, bus level and memory pool; Unreal's `stat audio` and `au.Debug` commands; Unity's Audio Profiler module; and `pw-top` or `pactl` on Linux for the system side.

45.10.6 WORDS - remember these

1. Sound bank — a group of sounds loaded together — a packaged set of audio assets and event metadata with a shared lifetime.
2. Voice — one sound currently playing — an active mixer channel consuming decode and processing budget.
3. Attenuation — getting quieter with distance — a gain curve over distance, physically minus 6 dB per doubling.
4. Occlusion — a wall in the way — attenuation and low-pass filtering applied when the direct path to the listener is blocked.
5. Bus — a group fader — a mix group through which many voices are summed before further processing.
6. Middleware — audio tools bought in — FMOD or Wwise, giving designers an authoring environment decoupled from engine code.

45.11 Multiplayer and netcode

45.11.1 PLAIN - in simple words

1. Here is the problem, and it cannot be solved, only hidden.
2. Information takes time to travel. Light in a glass fibre goes about 200,000 kilometres a second, which is fast but not instant.
3. Mumbai to Singapore and back is roughly 60 milliseconds of round trip in practice. Mumbai to Frankfurt is roughly 120. Mumbai to the United States east coast is roughly 240.
4. So when you press fire, the server learns about it at best 30 milliseconds later, and you learn its answer 30 milliseconds after that.
5. Meanwhile your screen must keep drawing at 60 frames a second. It cannot wait for the answer.
6. So the game lies to you, carefully. It shows you what it believes will happen, and corrects itself later when the truth arrives.
7. Everything in netcode is a variation on that one trick, plus the question of who is allowed to decide the truth.
8. The answer to that question is: a server that nobody plays on, owned by the game company. That is an **authoritative server**, and it won because any other arrangement lets a player's own machine cheat.

45.11.2 PLAIN - a picture in your head

1. Imagine two people playing chess by post, with letters taking three days.
2. If each waits for the other's letter before thinking, a game takes a year.
3. So each of them keeps a board at home and moves their own piece immediately, guessing what the opponent will do.
4. When the letter arrives, they compare. If the guess was right, nothing changes. If it was wrong, they must undo their guessed moves and redo them from the true position.
5. That undo-and-redo is exactly what your game does, sixty times a second, with the postal delay measured in milliseconds instead of days.
6. And there is a referee whose board is the only one that counts. If your board disagrees with the referee's, yours is wrong by definition.

Where this comparison breaks: chess players can wait, and a game cannot. Also, in chess both boards are complete. In a game the referee deliberately does not tell you about pieces you cannot see, because if it did, you could cheat.

45.11.3 PLAIN - a worked example

1. The full sequence for one shot, with real numbers. Round trip time is 80 milliseconds, so one-way delay is 40 milliseconds. The server runs at 64 ticks per second, so a tick is 15.6 milliseconds. Your interpolation delay is 100 milliseconds.

```
t=0    you press fire; client predicts the shot at once
t=0    input #4471 stamped and sent over UDP
t=40   server receives input #4471
t=40   server rewinds all hitboxes by 40 + 100 = 140 ms
t=40   server tests the ray against the rewound world
t=47   next server tick packages the result
t=87   you receive "hit confirmed, input #4471 processed"
t=87   client reconciles: state matches, nothing visible
```

2. Now the same event from the victim's point of view. On their screen they ran behind a wall at their local time t=-20 milliseconds, that is, 20 milliseconds before you pressed fire.
3. The server still counted the hit, because it rewound the world to where you saw them, and there they were in the open.

4. How far can that be? At a running speed of 6 metres per second, 140 milliseconds is 0.84 metres. That is most of a body width, which is exactly why it feels like being shot through a wall.
5. The 140 milliseconds is not padding. It is 40 milliseconds of network delay plus the 100 milliseconds of deliberate interpolation delay your client uses to smooth other players' movement.

45.11.4 PLAIN - what is really happening inside

1. **Client-side prediction.** Your client never sends "I am at position X". It sends "I held forward and pressed fire, and this is input number 4471". Then it immediately applies that input locally so the screen responds now.
2. It keeps every unacknowledged input in a list.
3. **Server reconciliation.** The server replies with "here is your true state after I processed input 4471". Your client snaps to that state and then replays inputs 4472 onwards on top of it.
4. If the prediction was correct, the replay lands exactly where you already were and you see nothing. If it was wrong, you see a small correction.
5. **Entity interpolation.** Other players are not predicted. Instead your client deliberately renders them slightly in the past, between the last two snapshots it received, so their movement is smooth even though the snapshots arrive unevenly.
6. **Lag compensation.** The server records where every hitbox was for the last half second or so. When your shot arrives, it rewinds to what you were actually looking at, tests the hit there, then restores the present.
7. That is the entire architecture. Prediction hides your own latency. Interpolation hides everyone else's jitter. Lag compensation makes aiming fair. The cost is paid by the person being shot.

45.11.5 TECHNICAL - the engineer's version

1. The techniques above were set out publicly in Yahn Bernier's 2001 paper for Valve, "Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization", building on the prediction work in QuakeWorld from 1996. Almost every shooter since uses this model.
2. Blizzard's Overwatch, described in Timothy Ford's GDC 2017 talk, runs 16 millisecond command frames, about 62.5 per second, tightened to about 7 milliseconds in tournament configuration. The client clock runs ahead of the server by roughly half the round trip plus one buffered command frame, so at 160 milliseconds of latency it predicts about 96 milliseconds ahead. Above about 220 milliseconds round trip it stops predicting hits.
3. **Tick rate** is how often the server simulates. 64 ticks is one every 15.625 milliseconds; 128 ticks is one every 7.8125 milliseconds. Halving the tick interval halves the average quantization error on event timing, from about 7.8 to about 3.9 milliseconds, and doubles both server processor cost and downstream bandwidth. That is a money decision, not a technical one, which is why third-party services ran 128-tick Counter-Strike servers while official matchmaking ran 64.
4. Counter-Strike 2, released in September 2023, introduced **sub-tick** input: the client timestamps each input with its exact moment inside the tick, and the server uses that timestamp rather than snapping to the tick boundary. Valorant instead runs 128-tick servers. Whether sub-tick fully matches a true 128-tick server is genuinely disputed by players and analysts, and there is no neutral published measurement that settles it.
5. **Deterministic lockstep** is the other architecture. Only player commands are transmitted; every machine runs an identical simulation. Bandwidth is independent of the number of units, which is why real-time strategy games use it. The canonical write-up is Mark Terrano and Paul Bettner's GDC 2001 paper "1500 Archers on a 28.8", about Age of Empires.
6. Lockstep requires bit-identical simulation across every machine, for the reasons given in section 45.7: unspecified transcendental functions, compiler reordering, fused multiply-add, and non-deterministic thread ordering will all desynchronize it. It also needs a command delay of two to five turns, typically 100 to 250 milliseconds, so that all commands arrive before the turn executes. That delay is why strategy games feel slightly sluggish by design.

7. Rollback netcode for fighting games, popularized by Tony Cannon's GGPO from 2006, is lockstep plus speculation: assume the opponent pressed nothing, and when the real input arrives, restore the saved state and re-simulate the intervening frames. It needs a fast full state save and restore every frame, which constrains how the game is written.
8. **Bandwidth.** A worked example: 64 players, 20 snapshots per second, 200 bytes per snapshot after delta compression and relevancy culling, gives 64 x 20 x 200 = 256,000 bytes per second, about 2.05 megabits per second outbound per server, before packet headers.

Genre	Snapshot rate	Down per client
Competitive shooter	60 to 128 Hz	200 to 800 kbps
Battle royale, 100 players	20 to 30 Hz	150 to 500 kbps
MMO	5 to 20 Hz	30 to 150 kbps
Strategy, lockstep	5 to 10 Hz	5 to 30 kbps

9. **Delta compression** sends only what changed since the last snapshot the client acknowledged, a model established by Quake 3 in 1999. Combined with quantization it is dramatic: a position stored as three 32-bit floats is 12 bytes; quantized to 16-bit fixed point over a bounded world it is 6 bytes; as a delta from the previous value with variable-length bit packing it is often 1 to 3 bytes.
10. **Relevancy culling**, sending a client only what it could plausibly perceive, saves bandwidth and is also the single most effective anti-cheat measure that exists, because a wallhack cannot draw data it never received. Riot's Fog of War in Valorant, shipped in 2020, is the best-known example.
11. **Why UDP.** TCP, standardized in RFC 793 in 1981 and now RFC 9293 from 2022, guarantees an ordered reliable byte stream. If packet 5 is lost, packets 6, 7 and 8 sit in the kernel buffer until 5 is retransmitted. That is head-of-line blocking, and it costs at least one round trip. For position updates it is exactly wrong: by the time packet 5 arrives it is stale and you already hold newer truth.
12. UDP, RFC 768 from 1980, has no ordering, no retransmission and no congestion control. Games build their own reliability layer and choose per message: unreliable for positions, since the next update supersedes this one; reliable and ordered for chat and inventory; reliable but unordered for one-off events. QUIC, RFC 9000 from 2021, offers independent streams over UDP and is used for some game backend traffic, but action games generally still roll their own.
13. Practical UDP detail: since there is no connection, NAT devices time out idle mappings, commonly after 30 to 120 seconds, so games send small keep-alive packets.
14. **Matchmaking** is a multi-objective search over skill, latency, party size, role and queue time, widening its tolerances as the wait grows. Skill systems descend from Arpad Elo's chess rating from the 1960s, through Glicko, to Microsoft's TrueSkill from 2007, which models skill as a distribution with an uncertainty that shrinks as it observes you.
15. **NAT traversal** matters because most home connections, and every carrier-grade NAT connection, have no reachable public address. STUN, currently RFC 8489 from February 2020, which obsoletes RFC 5389 of 2008 and RFC 3489 of 2003, lets a host discover the public address and port its NAT assigned. TURN, currently RFC 8656 from February 2020, relays traffic when a direct path cannot be established. ICE, RFC 8445 from 2018, gathers all candidate addresses and systematically tests pairs.
16. Hole punching works when both sides send outbound packets simultaneously so each NAT creates a mapping the other can use. It fails when both ends use symmetric NAT, which allocates a different external port per destination. A relay is then the only option, and a single-digit to low-double-digit percentage of connection attempts need one in practice.
17. **Relays** such as Steam Datagram Relay carry game traffic over a provider's private backbone. The benefits are that the server's real address is never exposed, which prevents denial-of-service attacks on individual players; routing is often better than the public internet; and NAT problems disappear.

18. The unfixable part: a player far from any datacentre cannot be given a competitive experience. No amount of netcode removes 240 milliseconds. This is a real and often-ignored fairness issue for players in regions without local server presence.

45.11.6 WORDS - remember these

1. Authoritative server — the referee whose word counts — a server that simulates canonically and treats client messages as inputs, never as state.
2. Client-side prediction — acting before permission — applying local input immediately and retaining it for later replay.
3. Reconciliation — correcting the guess — snapping to the server state and replaying unacknowledged inputs on top of it.
4. Entity interpolation — drawing others slightly in the past — rendering remote entities between two received snapshots with a fixed delay buffer.
5. Lag compensation — rewinding to what the shooter saw — server-side historical hitbox rewind by the shooter's latency plus interpolation delay.
6. Tick rate — how often the server thinks — simulation steps per second, 64 ticks being 15.625 ms and 128 ticks 7.8125 ms.
7. Delta compression — send only what changed — encoding a snapshot as a difference from the last one the client acknowledged.
8. Deterministic lockstep — send commands, not state — identical simulation on every machine, requiring bit-exact reproducibility.

45.12 Cheating and anti-cheat

45.12.1 PLAIN - in simple words

1. Cheating in an online game means making your copy of the program do something the designers did not intend, to gain an advantage.
2. An **aimbot** reads the game's memory to find where other players are, then points your weapon at them for you.
3. A **wallhack** draws other players through walls. It works only because your machine was told where they are.
4. A **speed hack** changes the clock the game reads, so your character moves faster than everyone else's.
5. **Packet manipulation** means interfering with the messages themselves: a lag switch drops your outgoing packets so others cannot hit you.
6. Some cheating is not technical at all: paying someone to play your account, deliberately losing to face weaker opponents, or trading wins.
7. The one rule that explains all defences is this: your machine belongs to the attacker. Anything the game checks on your machine, the attacker can remove.
8. So the only real defence is to check on the server, and to never send your machine information it is not entitled to.

45.12.2 PLAIN - a picture in your head

1. Think of an examination hall.
2. Asking candidates to mark their own papers is the client trusting itself. Some will cheat, and you will never know.
3. Collecting the papers and marking them centrally is server-side validation. Now the cheating must happen in the room, not in the marking.
4. Not printing the answers on the back of the paper is relevancy culling. You cannot copy what you were never given.
5. Invigilators watching for suspicious behaviour are statistical detection.

6. And searching candidates at the door, going through their pockets, is kernel-level anti-cheat. It works, and people reasonably object to it.

Where this comparison breaks: an exam lasts three hours. Anti-cheat software that runs at boot is searching your pockets every day whether or not there is an exam, which is precisely the objection people raise.

45.12.3 PLAIN - a worked example

1. How server-side validation actually looks, for movement.
2. The client sends: "input 4471, held forward, delta time 15.6 milliseconds".
3. The server does not read a position. It runs the same movement code the client ran, from the state it already holds.
4. Maximum run speed is 5.2 metres per second. In 15.6 milliseconds the maximum possible movement is 5.2 x 0.0156 = 0.081 metres.
5. If the client claims a delta time of 100 milliseconds while only 15.6 have passed on the server's clock, that is a speed hack, and it is rejected by comparing the client's claimed time budget against elapsed server time.
6. The same pattern applies everywhere: fire rate against the weapon's cooldown, damage against the weapon's table, line of sight against the server's own geometry, and inventory against the server's own database.

Cheat	Defence that works
Aimbot	Statistical aim analysis
Wallhack	Do not send hidden positions
Speed hack	Server-side time budget
Damage edit	Server owns the damage table
Item duplication	Atomic server transactions

45.12.4 PLAIN - what is really happening inside

1. Statistical detection looks at how you aim rather than at your files.
2. Human aim has tremor, overshoot, correction and a reaction time that never drops below about 150 milliseconds for a genuinely unexpected event.
3. A machine has none of those. Its angular snap to a target is too fast, too precise, and too consistent across distances.
4. Detectors measure the distribution of flick times, the time between a target becoming visible and the first shot, the hit rate as a function of range, and the smoothness of the mouse path.
5. The output is a probability, not a verdict, which is why bans are issued in waves rather than instantly: batching hides which specific detection fired, so cheat authors cannot binary-search their way around it.
6. False positives are the hard part. At a 0.1 percent false positive rate across 10 million players, 10,000 innocent people are banned. This is why detections are usually required to be independently corroborated.

45.12.5 TECHNICAL - the engineer's version

1. Categories in more detail: memory-reading external cheats; injected internal cheats hooking the rendering or input path; triggerbots; radar overlays; automated recoil control macros in hardware mice; lag switches and packet delay; and account services such as boosting and win trading.
2. Two categories defeat all software-only defences. **DMA cheats** use a second computer with a PCIe direct memory access card reading the gaming machine's memory over the bus, so nothing runs on the machine being watched. **Screen-reading cheats** capture the video output with a capture card, run

object detection on a second machine, and drive a hardware device that emulates a mouse. Both are a live problem in 2026 and neither is detectable by inspecting the game machine's software.

3. Kernel-level anti-cheat runs a driver at ring 0 so it can see and block things user-mode code cannot: other drivers loading, handles opened to the game process, hooked system calls, and tampered kernel structures. Shipped examples include Riot Vanguard, introduced with Valorant in 2020 and extended to League of Legends in 2024; Easy Anti-Cheat, now owned by Epic Games; BattlEye; and Activision's Ricochet, introduced in 2021.
4. The honest debate, both sides in full. For: cheats themselves run in the kernel, and a user-mode watcher simply cannot see a kernel-mode cheat, so without a kernel presence the defender has already lost the privilege race. Games with kernel anti-cheat measurably have fewer visible cheaters.
5. Against: it is a signed driver with total access to your machine, shipped by a game company, usually loading at boot whether or not you play. A bug in it is a system-wide vulnerability. This is not hypothetical: the `mhyprot2.sys` driver shipped with Genshin Impact was abused by ransomware operators to disable antivirus software, and the July 2024 CrowdStrike incident showed what a single faulty kernel-mode update does at scale.
6. It also excludes people. Kernel anti-cheat generally does not work under Proton on Linux or the Steam Deck unless the developer explicitly enables the vendor's Linux support, which many competitive titles decline to do.
7. The platform direction is changing. Microsoft announced the Windows Resiliency Initiative in November 2024 and from 2025 began moving endpoint security products out of kernel mode into a supported user-mode platform, with security vendors involved in the design. As of August 2026 this is a live transition rather than a completed one, and how anti-cheat fits into it is not yet settled.
8. Hardware-backed identity raises the cost of ban evasion. Valorant requires TPM 2.0 and Secure Boot on Windows 11, which lets bans attach to something harder to change than an account or a network address.
9. Machine learning detection is real and shipped: Valve described VACnet, a classifier trained on human-reviewed cases, at GDC in 2018, and comparable systems exist at other publishers. It is a filter that raises human review efficiency, not an oracle.
10. The arms race is structural and permanent. Cheat development is a paid industry with subscription pricing, private builds to avoid signature detection, and customer support. Anti-cheat cannot win; it can only make cheating expensive, unreliable and short-lived enough that most people do not bother.

45.12.6 WORDS - remember these

1. Aimbot — the machine aims for you — automated target acquisition driven by memory-read entity positions or screen-space detection.
2. Wallhack — seeing through walls — rendering entity positions the client received but should not have been told about.
3. Server-side validation — the server checks everything — simulating client inputs authoritatively and rejecting physically impossible claims.
4. Relevancy culling — do not send what cannot be seen — filtering network state by visibility, which prevents wallhacks at the source.
5. Kernel-level anti-cheat — a watcher with full system access — a ring 0 driver monitoring for tampering, with the privilege and stability trade-off that implies.
6. Ban wave — punishing in batches — delayed enforcement that obscures which detection triggered, slowing the cheat authors' feedback loop.

45.13 What actually happens when you double-click a game

45.13.1 PLAIN - in simple words

1. The short version of the ordinary part: the operating system reads the file's header, works out how much memory the program needs, copies the code and data into that memory, fills in the addresses of the system libraries the program uses, and jumps to its first instruction.
2. That is true for every program. A game then does a great deal more before you see anything.
3. It usually starts a **launcher** first, which checks for updates, verifies that your files have not been altered, and logs you in.
4. It starts the anti-cheat service, if there is one.
5. Then it starts the real game program, which must find a graphics card, ask the driver for a device, and set up the chain of pictures it will draw into.
6. Then it must turn its shader source into machine code for your specific graphics chip. On the first run this can take minutes, and if it is done lazily while you play, you get the famous first-run stutter.
7. Then it loads a small amount of content for the menu, and only then does the loop from section 45.3 start running.
8. Choosing a level starts a second, much bigger load, and finally you play.

45.13.2 PLAIN - a picture in your head

1. Think of an aircraft before a flight.
2. Opening the door is the double-click. Nothing is running yet.
3. Ground power, then auxiliary power, then engines: that is the loader, the launcher and the engine start.
4. The pre-flight checklist is the device and swap chain creation. Every item must be confirmed before the next.
5. Loading passengers and fuel is asset loading. It is the slowest part and it scales with how far you are going.
6. Taxiing is the main menu: you are moving, everything is running, but the real work has not started.
7. Take-off is entering the level.

Where this comparison breaks: an aircraft does the same checklist every time. A game's first run is dramatically different from its second, because the shader cache, the file system cache and the driver cache are all empty the first time.

45.13.3 PLAIN - a worked example

1. The twenty-five step trace, with realistic timings for a Windows PC with an NVMe solid-state drive, a mid-range graphics card and a broadband connection. Cold means first ever run; warm means a later run.

Step	What happens	Warm time
1	Shell creates the process	20 ms
2	Loader maps the launcher	30 ms
3	Launcher checks for updates	200 to 2000 ms
4	File integrity spot check	500 ms to 30 s
5	Anti-cheat service starts	50 to 500 ms
6	Launcher spawns the game	50 ms
7	Loader maps the game image	100 to 400 ms
8	Runtime and static init	20 to 100 ms
9	Read config and command line	5 to 20 ms
10	Memory pools reserved	10 to 50 ms
11	Job system threads start	5 to 20 ms
12	Archives mounted, index read	50 to 300 ms
13	Enumerate adapters, device	100 to 500 ms

Step	What happens	Warm time
14	Create swap chain buffers	50 to 200 ms
15	Allocate render targets	50 to 200 ms
16	Load or compile shader states	2 s warm, minutes cold
17	Audio device and mixer graph	50 to 200 ms
18	Enumerate input devices	10 to 100 ms
19	Sockets, DNS, account login	200 to 1500 ms
20	Load front-end bundle	500 to 3000 ms
21	Main menu, loop now running	first frame
22	You pick a level, load list	10 ms
23	Stream, decompress, upload	3 to 30 s
24	Warm-up frames force work	1 to 5 s
25	Gameplay loop settles	steady state

2. Warm total to the main menu is roughly 4 to 8 seconds. Cold total can be several minutes, almost all of it in step 16.
3. Step 16 is the one worth understanding. Shader source is compiled at build time into an intermediate form, but the final translation into your exact graphics chip's instructions can only happen on your machine, because the developer does not know which chip you have.
4. A game may have 20,000 to 200,000 pipeline states. If each takes 20 milliseconds to build, 50,000 of them is about 17 minutes of one core, or roughly one minute spread across sixteen cores.
5. If the game builds them lazily instead, the first time a new effect appears the frame stops for 20 to 200 milliseconds. That is the stutter. Consoles never have it, because the chip is known and the compilation happened at build time.

45.13.4 PLAIN - what is really happening inside

1. A **save file** is a written-down copy of the parts of the world the designer decided matter.
2. It is never the whole world. Nobody stores every leaf position. It stores your position, your inventory, quest flags, which doors are open, which enemies are dead, and a version number.
3. Turning live objects into a flat sequence of bytes is called **serialization**, and reading it back is deserialization.
4. The formats used, in rough order of how common they are:

Format	Example use	Trade-off
Chunked binary	Most console games	Fast, opaque
Tagged binary	Minecraft NBT	Flexible, compact
JSON, often gzipped	Many indie games	Readable, large
SQLite database	Mobile, big worlds	Robust, heavier
Engine key-value	Unity PlayerPrefs	Settings only

5. Minecraft's NBT, standing for Named Binary Tag, is a tagged tree of typed values, gzip-compressed, with world data grouped into region files that each hold a 32 by 32 block of chunk columns.
6. The correct way to write a save is: write to a temporary file, flush it to the storage device, then rename it over the old one. Rename is atomic on every mainstream file system, so a power cut leaves either the old save or the new one, never half of each. Games that write in place corrupt saves.
7. Saves must carry a version number, because patch 1.20 must still load a save written by 1.00, which means keeping an upgrade path for every old version. This is why some games break saves at a major update and why players are right to be annoyed about it.

8. Online games sign or checksum saves and usually store them server-side, so that editing them is pointless. Single-player games often do not, which is why save editors exist and are harmless.

45.13.5 TECHNICAL - the engineer's version

1. On Windows the image is a PE file: the loader maps sections according to the section table, applies base relocations if the preferred base is taken, resolves the import address table, runs TLS callbacks, then calls the entry point. On Linux it is an ELF file handled by `ld.so`. Section 45.13 does not repeat that machinery; it only notes where it sits in the sequence.
2. Graphics initialization in Direct3D 12 is: create a DXGI factory, enumerate adapters, `D3D12CreateDevice`, create command queues, then `CreateSwapChainForHwnd` with 2 or 3 buffers and a chosen swap effect, usually flip-model. In Vulkan it is `vkCreateInstance`, physical device selection, `vkCreateDevice`, then a surface and a `VkSwapchainKHR`.
3. Pipeline state objects bundle shaders, blend, depth, rasterizer and vertex layout into one immutable object, which is why they are compiled up front rather than assembled per draw. Both APIs support a pipeline cache blob that can be saved to disk and reloaded, which is what makes the warm run fast. Steam additionally distributes precompiled Vulkan pipeline caches for Proton titles so a Steam Deck rarely compiles at play time.
4. Asset streaming on modern consoles uses hardware decompression and a direct storage path: the PlayStation 5 Kraken decompressor is rated at about 5.5 gigabytes per second of compressed input, and Microsoft's DirectStorage on Windows, with GPU decompression from version 1.1 in 2022, brings a similar path to PC.
5. Useful observation commands: Process Monitor to watch every file the game opens in order; `strace -f` or `ltrace` on Linux; Windows Performance Recorder or `xperf` for a boot-to-first-frame trace; `nvidia-smi` or GPU-Z to watch video memory fill during load; and the engine's own load-time trace, such as Unreal Insights with the LoadTime channel.

45.13.6 WORDS - remember these

1. Launcher — the small program that runs first — an updater, integrity checker and authentication front end that spawns the real executable.
2. Swap chain — the queue of pictures — the set of back buffers created at startup and rotated between renderer and display.
3. Pipeline state object — a frozen graphics configuration — an immutable bundle of shaders and fixed-function state compiled ahead of a draw.
4. Shader cache — the compiled results kept for next time — an on-disk store of driver-compiled pipeline blobs keyed by hardware and driver version.
5. Serialization — turning live objects into bytes — writing state in a versioned, replayable form, and reading it back.
6. Atomic rename — the safe way to save — writing to a temporary file and renaming over the target so no partial file is ever visible.

45.14 Consoles versus PC

45.14.1 PLAIN - in simple words

1. A PC is a different machine for every person. A console is the same machine for everybody.
2. That single difference is worth more than it sounds.
3. On a console the developer knows exactly how fast the processor is, exactly how much memory there is, exactly how fast the storage is, and exactly which graphics chip will run the shaders.
4. So the developer can tune to that machine, compile shaders in advance, use the memory to the last megabyte, and rely on a fixed loading speed.

5. On a PC the same code must work on thousands of combinations, through a driver that has to guess, with unknown amounts of memory, and with shaders compiled on your machine while you wait.
6. In exchange for that constraint, the console maker takes control: your game must pass their tests before it can be sold.
7. That is why a console game with weaker numbers on paper can look better than a PC game on stronger hardware.

45.14.2 PLAIN - a picture in your head

1. Think of a tailor.
2. A bespoke suit is measured for one body. It fits perfectly and it fits nobody else. That is a console game.
3. Off-the-peg suits come in twelve sizes and must fit everybody roughly. They need more fabric in places, and they hang a little loose. That is a PC game.
4. The bespoke suit is not made of better cloth. It just wastes none.

Where this comparison breaks: a PC owner can buy far better cloth. A 1,500 dollar PC in 2026 genuinely outperforms any console, and gets modding, backwards compatibility and higher frame rates too. The console advantage is efficiency per unit of hardware, not raw capability.

45.14.3 PLAIN - a worked example

1. The 2026 landscape, with United States prices as of August 2026. Note that prices in this generation went up rather than down, which is historically unusual.

Machine	Graphics	Price
PlayStation 5 (2020)	36 CU, 10.3 TFLOPS	649.99
PlayStation 5 Pro (2024)	60 CU, 16.7 TFLOPS	899.99
Xbox Series X (2020)	52 CU, 12.1 TFLOPS	649.99
Xbox Series S (2020)	20 CU, 4 TFLOPS	399.99
Nintendo Switch 2 (2025)	1,536 Ampere cores	449.99

2. PlayStation raised prices in August 2025 and again on 2 April 2026. Xbox raised prices in October 2025. The Nintendo Switch 2 launched at 449.99 on 5 June 2025 and has held that price.
3. Memory tells the same story as compute. The PlayStation 5 has 16 gigabytes of GDDR6 at 448 gigabytes per second, shared between processor and graphics chip. The Pro keeps 16 gigabytes but raises bandwidth to 576 gigabytes per second and adds 2 gigabytes of DDR5 for the system. The Switch 2 has 12 gigabytes of LPDDR5X at about 102 gigabytes per second docked.
4. Unified memory is the quiet advantage. On a PC, data must be copied across the PCI Express bus from system memory to video memory. On a console there is one pool and no copy.

45.14.4 PLAIN - what is really happening inside

1. Certification is a checklist, not a review of whether the game is good.
2. Sony calls its list the Technical Requirements Checklist. Microsoft calls its list the Xbox Requirements. Nintendo's process is called Lotcheck.
3. The checks are behavioural. What happens if the controller battery dies mid-save. What happens if the network drops during a purchase. What happens if the user signs out. Does the game suspend and resume within the required time. Does it use the platform's exact button names. Does it declare its save data size correctly. Does it survive an eight-hour soak test.
4. A submission takes on the order of one to three weeks per platform per build, and a failure means fixing and resubmitting. Studios therefore lock content weeks before launch to leave room for one or two failures.

5. Those durations are studio experience rather than published figures, and they vary by platform, by title and by relationship.
6. Age ratings are a separate process again: ESRB in North America, PEGI in Europe, CERO in Japan, USK in Germany, and the IARC questionnaire for digital storefronts.

45.14.5 TECHNICAL - the engineer's version

1. The concrete reasons a fixed target wins, in order of importance: precompiled shaders, so no run-time compilation and no first-run stutter; unified memory, so no bus copies; low-overhead graphics APIs closer to the hardware than the PC equivalents; a known memory budget minus a known system reservation; guaranteed storage bandwidth with hardware decompression; and a single optimization target, so every hour of tuning benefits every player.
2. The APIs differ: PlayStation uses its own GNM and GNMX; Xbox uses Direct3D 12 with console-specific extensions and a fixed driver; Nintendo Switch and Switch 2 use NVN. All three expose more control and less driver interpretation than desktop Direct3D 12 or Vulkan.
3. System reservations are real and published in developer documentation rather than in marketing: the Xbox Series X reserves about 2.5 gigabytes of its 16, and the Switch 2 reserves about 3 gigabytes of its 12 by reported figures, leaving the rest to the game.
4. Handhelds are the growth area. The Steam Deck, released 25 February 2022 with an OLED model in November 2023, uses an AMD APU with a 4-core Zen 2 processor and 8 RDNA 2 compute units, about 1.6 teraflops, 16 gigabytes of LPDDR5, and a 1280 by 800 display, in a 4 to 15 watt power envelope.
5. SteamOS is Arch-based Linux, and Windows games do not run on Linux natively. **Proton** makes them run. It is not emulation: the processor instructions are native x86-64 and execute at full speed. Only the calls into Windows are translated.
6. Proton is Wine, the open reimplementation of the Windows API begun in 1993, plus DXVK which translates Direct3D 9, 10 and 11 into Vulkan, plus VKD3D-Proton which translates Direct3D 12 into Vulkan, plus FAudio for XAudio2, plus Steam-specific shims. Shader bytecode is translated too, from DXBC and DXIL into SPIR-V.
7. Typical overhead is 0 to 15 percent, and a minority of titles run faster than on Windows because of driver differences. The thing Proton cannot fix is kernel anti-cheat, which must be explicitly enabled for Linux by the game's developer.
8. Valve announced a **Steam Machine** in November 2025 for release in 2026: a small SteamOS box with a semi-custom 6-core AMD Zen 4 processor up to 4.8 GHz, an RDNA 3 graphics part, 16 gigabytes of system memory plus 8 gigabytes of dedicated graphics memory, running the same Proton stack. At the time of writing in August 2026 the price had not been announced.
9. The honest summary: consoles buy predictability and efficiency; PCs buy ceiling and freedom. Which is better depends entirely on whether you value a guaranteed experience or a configurable one, and reasonable people split on it.

45.14.6 WORDS - remember these

1. Fixed platform — one machine for everyone — a single known hardware configuration allowing exact tuning and ahead-of-time compilation.
2. Certification — the platform holder's checklist — a technical requirements pass covering behaviour, terminology, storage and stability.
3. Unified memory — one pool shared by processor and graphics — no PCI Express copies between system and video memory.
4. System reservation — the slice the console keeps — memory and processor time reserved for the operating system and background features.
5. Compatibility layer — translating one platform's calls to another — Proton, which maps Windows and Direct3D calls to Linux and Vulkan without emulating the processor.
6. Lotcheck — Nintendo's certification — the platform's technical and content compliance review before release.

45.15 Emulation

45.15.1 PLAIN - in simple words

1. An **emulator** is a program that pretends to be a different computer.
2. The game it runs was compiled for a processor your machine does not have, talking to chips your machine does not have.
3. So the emulator reads the old machine's instructions one at a time and does whatever they were supposed to do, using your machine's instructions.
4. It also pretends to be the old machine's graphics chip, sound chip, cartridge slot and controller ports.
5. There are two ways to run the instructions. **Interpretation** reads and acts on each instruction every time it is encountered. Simple, accurate, slow.
6. **Recompilation** translates a whole block of old instructions into new instructions once, keeps the translation, and reuses it. Complicated, much faster.
7. There is a third shortcut. Instead of emulating the old machine's system software, the emulator can spot a call to a known system function and do the equivalent thing natively. That is **high-level emulation**.

45.15.2 PLAIN - a picture in your head

1. Interpretation is a live interpreter at a conference, translating sentence by sentence, every time, including sentences the speaker repeats.
2. Recompilation is having the speech translated once and printed, so the repeated parts are read from the page.
3. High-level emulation is noticing that the speaker just said a standard phrase, and saying the equivalent standard phrase in the target language rather than translating it word by word.
4. Accuracy is the trade in each case. The live interpreter catches every nuance. The printed version cannot adapt if the speaker goes off script.

Where this comparison breaks: a speech does not care about timing to the microsecond. Old consoles absolutely did, and many games depend on exactly when a chip responded, not just on what it returned.

45.15.3 PLAIN - a worked example

1. Why emulating a slow machine can need a fast one.
2. The Super Nintendo ran at about 3.58 MHz. A modern processor runs at about 4 GHz, roughly 1,100 times faster in clock terms alone, and does far more per clock.
3. Yet a fully cycle-accurate SNES emulator such as byuu's bsnes and higan needed roughly a 3 GHz machine to reach full speed.
4. The reason is not the instructions. It is the synchronization. A cycle accurate emulator must interleave the processor, the picture processing unit, the sound processor and the cartridge coprocessors many times per emulated instruction, and each switch costs far more than the instruction itself.
5. A less accurate emulator that runs each chip in blocks and synchronizes rarely is ten to fifty times faster and breaks a small number of games that rely on exact timing.

Approach	Speed cost	Accuracy
Cycle-accurate interpreter	500 to 1500x	Highest
Block interpreter	20 to 100x	Good
Dynamic recompiler	2 to 10x	Varies
High-level API emulation	1 to 3x	Lowest

6. Those ratios are approximate and depend heavily on the console being emulated.

45.15.4 PLAIN - what is really happening inside

1. An interpreter's inner loop is: read the instruction at the program counter, decode it, look up a handler, run the handler, update the registers and flags, advance the program counter, add the instruction's cycle cost to a counter, and check whether any hardware event is due.
2. That is 10 to 100 host instructions to do the work of one guest instruction.
3. A dynamic recompiler instead scans forward from the program counter until a branch, translates that whole block into host machine code once, stores it in a cache keyed by address, and jumps into it.
4. The block then runs at close to native speed on every later visit.
5. Self-modifying code and code overwritten by a memory bank switch force the cached block to be thrown away and retranslated, which is why emulating older machines with heavy bank switching is harder than it looks.
6. Emulating the graphics chip is where the biggest shortcut lives. If the emulator translates the old chip's drawing commands into modern graphics calls instead of simulating its pixel pipeline, it can also render at four or eight times the original resolution, which the original hardware could never do.

45.15.5 TECHNICAL - the engineer's version

1. Terms, precisely. **Interpretation** decodes per execution. **Threaded interpretation** removes dispatch overhead by chaining handler addresses. **Dynamic binary translation**, also called dynarec or JIT, compiles per basic block or per trace. **Static recompilation** translates the whole binary ahead of time and mostly fails on general programs because indirect jumps make it impossible to tell code from data reliably, though per-title static recompilation projects have succeeded, notably for Nintendo 64 titles from around 2023.
2. **High-level emulation** replaces the guest system's own libraries with native implementations, which is essential when the firmware cannot legally be distributed. **Low-level emulation** runs the real firmware and is more accurate but requires the user to supply it.
3. Accuracy has axes: instruction accuracy, cycle accuracy, sub-instruction bus timing, sample-accurate audio, and scanline or dot accurate video. Choosing a lower point on any axis buys speed and breaks some titles.
4. The legal position, stated neutrally and without advice.
5. Writing an emulator has been upheld in United States courts. In Sony Computer Entertainment v. Connectix, decided by the Ninth Circuit in 2000, intermediate copying of the PlayStation BIOS for reverse engineering was held to be fair use. Sony v. Bleem, also 2000, held that using screenshots in comparative advertising was fair use.
6. Copying game data you do not own is copyright infringement essentially everywhere. Whether dumping a cartridge you do own is lawful varies by country, and in several jurisdictions the act of circumventing copy protection to make the dump is itself unlawful regardless of ownership, under the United States Digital Millennium Copyright Act of 1998 and the European Union Copyright Directive of 2001.
7. Firmware and BIOS images are copyrighted works in their own right; distributing them is infringement whatever the emulator's status.
8. Enforcement in practice: in March 2024 Nintendo settled with Tropic Haze, developer of the Switch emulator Yuzu, for 2.4 million United States dollars, and the project shut down. Nintendo's complaint centred on circumvention of encryption rather than on emulation as a concept.
9. Where people disagree: emulator authors and archivists argue the tool is lawful, general-purpose and necessary; rights holders argue that emulators of current hardware exist principally to enable piracy. Both claims can be true of different projects at the same time.
10. Preservation is the strongest argument, and it has numbers behind it. A study published in July 2023 by the Video Game History Foundation with the Software Preservation Network found that about 87 percent of video games released in the United States before 2010 are out of commercial circulation entirely.
11. The mechanisms of loss are concrete: digital storefronts close, as the Wii Shop Channel did in 2019

and the Nintendo 3DS and Wii U eShops did in March 2023; optical discs suffer disc rot; cartridge save batteries last ten to twenty years; and flash memory loses charge over decades.

12. Museums and libraries have argued for broader legal exemptions to preserve and provide access to games. Some exemptions exist for on-site preservation work; remote access for researchers remains contested.

45.15.6 WORDS - remember these

1. Emulator — a program pretending to be another machine — software modelling a foreign instruction set and peripheral hardware.
2. Interpretation — decode every instruction every time — a fetch-decode- execute loop in software, simple and slow.
3. Dynamic recompilation — translate once, run many times — just-in-time compilation of guest basic blocks into host machine code.
4. High-level emulation — reimplement the system calls — replacing guest firmware routines with native host implementations.
5. Cycle accuracy — matching the original timing exactly — modelling per-clock behaviour including inter-chip contention.
6. ROM — a copy of the original game data — a dump of cartridge or disc contents, copyrighted material unless the rights holder permits otherwise.

45.16 Making your first game, practically

45.16.1 PLAIN - in simple words

1. Build something tiny. Then build something slightly less tiny.
2. The reason is not that you cannot handle a big idea. It is that finishing is a separate skill from building, and it is the rarer one.
3. A tiny game teaches you the whole pipeline: input, a loop, collision, a win condition, a lose condition, sound, a menu, a build, and publishing.
4. A big game teaches you one slice of that and then dies.
5. The proven ladder is: Pong, then Breakout, then Snake, then Asteroids, then a one-level platformer, then something of your own.
6. Each of those should take days, not months. If it is taking months, the scope is wrong.
7. Then publish it. Publishing a bad small game teaches you more than not publishing a good big one.

45.16.2 PLAIN - a picture in your head

1. Nobody learns to cook by attempting a five-course dinner for twenty.
2. You learn by making one dish, badly, then again, less badly.
3. The dinner party teaches you timing, which the single dish does not, so you do eventually need one. But you need it after ten dishes, not before.
4. A game jam is the dinner party: a fixed 48 hours, a theme announced at the start, and something submitted at the end regardless of quality.

Where this comparison breaks: a bad dish is eaten and forgotten. A published game stays on the internet under your name, which is exactly why some people never publish. Publish anyway. Nobody is looking as hard as you fear.

45.16.3 PLAIN - a worked example

1. A thirty-day plan. It assumes about two hours a day, which is realistic alongside study or a job.

Days	Task	Deliverable
1 to 2	Engine, one tutorial	A window, a moving box
3 to 4	Core verb only	Move and jump feel good
5 to 7	One obstacle, win, lose	A 30-second playable loop
8 to 9	Playtest with 3 people	Written notes, a cut list
10 to 13	Fix the top three problems	The loop reads clearly
14 to 16	Add exactly one system	The game has some depth
17 to 18	Second playtest	Notes, cut list again
19 to 21	Sound and feedback pass	It feels good to touch
22 to 24	Five levels or a curve	Content, not features
25 to 26	Menu, pause, settings	Ship-shaped
27 to 28	Build, test on a clean PC	An installable build
29	Store page, screenshots	Page ready
30	Publish on itch.io	Released

2. Notice that only days 3 to 7 and 14 to 16 add features. Everything else is testing, fixing, content and shipping. That ratio is correct and it surprises everyone the first time.

45.16.4 PLAIN - what is really happening inside

1. **Scope** is the number one killer of hobby projects, and the reason is arithmetic, not willpower.
2. Every feature costs its own build time plus its interaction with every other feature. The number of interacting pairs grows roughly as the square of the feature count.
3. Five features have ten pairs to make work together. Fifteen features have
105. That is why the second half of a project feels ten times harder than the first.
4. So the discipline is a **cut list**: when you have a new idea, write it down on a "version two" list, and carry on with what you were doing.
5. Rough multipliers for common additions, approximate but directionally right: multiplayer multiplies total work by about three; a save and load system by about 1.3; procedural generation by about two; a narrative mode by more than any of them.
6. The classic failure is choosing a massive online role-playing game or an open world as a first project. It has ended more hobby projects than any technical difficulty.
7. Prototype with untextured boxes. If a game is not fun as grey boxes, art will not save it. Art makes a good game feel better; it cannot make a boring game interesting.
8. Playtest by handing someone the controls and saying nothing. Every time you want to explain something, that is a design bug. Write it down instead of speaking.

45.16.5 TECHNICAL - the engineer's version

1. Choosing a tool by goal, as of August 2026:

Goal	Tool	Why
Understand the machinery	raylib or SDL3	No magic, all yours
Ship a 2D game quickly	Godot 4	Free, small, 2D-first
Reach mobile and consoles	Unity 6	Widest platform support
Highest-end 3D visuals	Unreal Engine 5	Best default look
Web game in a browser	Godot or a JS engine	HTML5 export

2. Code-first libraries worth knowing: SDL3 and raylib in C, LOVE in Lua, pygame in Python, MonoGame in C#, Bevy in Rust, and three.js in JavaScript. These give you a window, input, drawing and sound, and nothing else, which is exactly the point when learning.

3. Free asset sources with usable licences: Kenney's public domain 2D and 3D packs; OpenGameArt; Poly Haven and AmbientCG for CC0 textures, models and environment maps; Quaternius for CC0 low-poly models; Adobe's Mixamo for free rigged characters and animations; Freesound for Creative Commons audio; and Google Fonts. Read every licence and keep an attributions file from day one, because reconstructing it later is miserable.
4. "Juice" is the disproportionately effective polish layer: screen shake, a two to six frame hit pause, particles on impact, layered sound, tweened interface movement, and squash and stretch on jumps. The reference talk is "Juice It Or Lose It" by Martin Jonasson and Petri Purho from 2012.
5. Where to publish, with real numbers:

Platform	Cost to publish	Revenue share
itch.io	Free	You choose, 10% default
Steam	100 USD per game	30%, 25% over 10M
Epic Games Store	Free, curated	12%
Google Play	25 USD once	15% under 1M per year
Apple App Store	99 USD per year	15% under 1M per year

6. The Steam Direct fee of 100 United States dollars per title is refundable once the title reaches 1,000 dollars of adjusted gross revenue. Steam's share drops to 25 percent above 10 million dollars of lifetime revenue and 20 percent above 50 million.
7. Game jams are the most reliable way to finish anything. Ludum Dare has run since 2002, Global Game Jam since 2009, and the GMTK Jam is now among the largest. A 48-hour deadline enforces scope better than any amount of personal discipline.
8. A practical technical checklist before publishing: test on a machine that has never had your development tools installed; check that the game runs from a path with a space and a non-English character in it; handle alt-tab, minimize and display resolution change; provide key rebinding; check it on a 16:10 and an ultrawide display; and make sure the game can be quit with the keyboard alone.

45.16.6 WORDS - remember these

1. Scope — how big the thing is — the total feature and content set, and the dominant risk in any small project.
2. Cut list — the ideas you deliberately postpone — a written backlog that protects the current milestone from new ideas.
3. Prototype — the ugly playable test — a minimal implementation built to answer one design question, then usually discarded.
4. Playtest — watching someone else play — structured observation of a naive player without designer intervention.
5. Juice — cheap polish that feels expensive — screen shake, hit pause, particles and easing applied to feedback moments.
6. Game jam — a short deadline with a theme — a fixed-duration event, typically 48 to 72 hours, used to force completion.

45.98 Common wrong ideas

1. Wrong: games are made by one genius programmer. Right: a modern game is made by a team where roughly one person in five writes code, and the largest groups are artists and quality assurance. Even the famous solo games, such as Stardew Valley, took a single person four and a half years and still used other people's engine, tools and audio libraries.
2. Wrong: more polygons means better graphics. Right: past a certain density extra triangles are smaller than a pixel and cost performance for no visible gain. What actually makes an image look good is

lighting, material response, texture quality, animation and art direction. A 2015 game with good art direction routinely looks better than a 2026 game without it.

3. Wrong: the engine makes the game. Right: the engine provides the machinery. Every game in the same engine can look and feel completely different, because the design, the art, the tuning and the content are the game. An engine can make a bad game easier to build; it cannot make it good.
4. Wrong: lag is always your internet connection. Right: what you feel as lag is the sum of your input device, your frame rate, your render queue depth, your display, your distance to the server, the server's tick rate, and the game's interpolation delay. A player with a perfect connection on a distant, low-tick server will feel worse than a player with a mediocre connection nearby.
5. Wrong: a higher average frame rate always feels smoother. Right: evenly paced frames feel smoother than unevenly paced ones at the same average. 112 frames per second average with one 100-millisecond hitch feels worse than a steady 90.
6. Wrong: game physics is real physics. Right: it is a stable, cheap approximation tuned for feel. Jump arcs, friction and gravity in most games are deliberately unphysical because physical values feel bad to play.
7. Wrong: game AI is machine learning. Right: shipped game AI is almost entirely hand-authored state machines, behaviour trees, utility scoring and pathfinding, deliberately weakened so it is fun to fight. Learned agents exist in research and beat humans, and are not what is in your game.
8. Wrong: emulators are illegal. Right: writing and using an emulator has been upheld as lawful in United States courts. Copying game data or firmware you do not own is what is unlawful, and the two questions are separate.
9. Wrong: kernel anti-cheat means the game is secure. Right: it raises the cost of software cheating, and it cannot see a direct-memory-access cheat on a second computer or a screen-reading cheat driving a hardware mouse. Server-side validation and not sending hidden data remain the only defences that cannot be bypassed on the client.
10. Wrong: consoles are just locked-down PCs. Right: the fixed target changes what is possible. Precompiled shaders, unified memory, a known memory budget and one optimization target let a console extract far more from the same silicon than a PC of identical specification.

45.99 Chapter summary in 20 lines

1. A game is a program that redraws the world on a clock, whether or not anything changed, unlike an application that redraws only on an event.
2. That single decision produces the frame budget: 16.67 ms at 60 frames per second, 8.33 ms at 120, 6.94 ms at 144.
3. The loop is input, update, render, present, and it never sleeps except when waiting for the display.
4. Physics needs a fixed timestep because integration is not step-size invariant; an accumulator collects real time and spends it in fixed steps.
5. Rendering interpolates between the last two simulation states using the leftover accumulator fraction, which is what makes 60 Hz physics look smooth at 144 frames per second.
6. Frame time is the real measurement; average frames per second hides stutter, which is why 1 percent lows are reported.
7. Games began as laboratory curiosities in 1958 and 1962, became wired logic in 1972, gained processors and cartridges in 1977, crashed in 1983, and were rebuilt around platform control from 1985.
8. Doom in 1993 and 3D acceleration from 1996 moved the bottleneck from pixel fill to triangle throughput, and then to per-pixel computation.
9. A modern AAA game costs 80 to 300 million dollars and employs 200 to 600 people, of whom fewer than a quarter write code; content, not code, is the cost.
10. Engines supply the reusable machinery and an editor. Unity, Unreal and Godot differ chiefly in language, licence and default strengths, and a licence is part of your technology stack, as 2023 demonstrated.

11. Entity-component-system layouts win on speed because contiguous component arrays match how caches and prefetchers work.
12. The asset pipeline converts editor formats into platform-specific runtime formats, a process Unreal calls cooking; shader permutations and lightmap baking are why builds take hours.
13. Physics is broad phase, narrow phase, and impulse-based resolution; momentum is conserved and restitution controls how much energy is lost.
14. Floating-point results are not reproducible across machines in general, which is why deterministic lockstep networking is hard.
15. Game AI is state machines, behaviour trees, utility scoring, planners and A-star pathfinding, deliberately weakened so that fighting it is fun.
16. A rendered frame is culling, batching, sorting, shadow passes, a forward or deferred lighting pass, transparency, post-processing, then interface.
17. Latency cannot be removed, only hidden: client-side prediction, server reconciliation, entity interpolation and lag compensation together explain why you get shot after reaching cover.
18. The client can never be trusted; server-side validation and not sending invisible data are the only defences that cannot be patched out.
19. Launch is loader, launcher, anti-cheat, device and swap chain creation, shader compilation, streaming and finally the loop, which is why the first run stutters and consoles do not.
20. Start tiny, cut ruthlessly, playtest silently, and publish. Scope, not skill, is what ends most projects.

46

CHAPTER

KB

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

46.0 What this chapter gives you

1. You will be able to say exactly what machine learning is, in one sentence, without using the word "intelligence", and explain how it differs from ordinary programming with a concrete example of both.
2. You will be able to tell the honest history: the perceptron, two winters, the support vector machine years, and why 2012 changed everything.
3. You will be able to fit a straight line to five points by hand, define what a parameter is, and compute the error of your fit two different ways.
4. You will be able to explain gradient descent to someone who has never seen a derivative, then work four steps of it on paper with real numbers.
5. You will be able to use the words sample, feature, label, epoch, batch, iteration and step correctly, and never confuse them again.
6. You will be able to read a training curve and say whether the model is underfitting, overfitting, or learning properly.
7. You will be able to name the main classical methods, say when each is the right tool, and explain why neural networks are not always the answer.
8. You will be able to describe one artificial neuron exactly, compute its output by hand, and prove in two lines why a network with no activation function is worthless.
9. You will be able to explain backpropagation in plain language as blame assignment, then do a complete backward pass through a small network with real numbers.
10. You will be able to say what a trained model physically is, and why training costs so much more than using the result.

46.1 What machine learning actually is

46.1.1 PLAIN – in simple words

1. In ordinary programming you know the rule, and you write it down.
2. You want to convert Celsius to Fahrenheit, so you write: multiply by 1.8, then add 32. Done. The rule was in your head and now it is in the file.
3. Machine learning is for the other case: you do not know the rule, but you have a pile of examples of the rule working.
4. You have ten thousand emails, each already marked "spam" or "not spam". You cannot write the rule that separates them. Nobody can.
5. So you do something different. You write down a **shape** for a rule, with blank spaces in it where numbers should go.

6. Then you write a procedure that fills in those numbers, by trying, checking against the examples, and adjusting.
7. When the procedure stops, the blanks are full. Those numbers are the rule.
8. Nobody wrote them. Nobody can read them. But they work on the examples, and usually on new cases too.
9. That is the whole of machine learning. A rule-shaped thing with adjustable numbers, plus a procedure that tunes the numbers to match examples.
10. At heart this is curve fitting: finding the settings that make a formula pass close to a set of points.
11. Saying "it is just curve fitting" is not dismissive. It is accurate, and the accuracy is what lets you reason about it.
12. Curve fitting in a space of a billion adjustable numbers, on data covering most of what humans have written, does astonishing things. It is still curve fitting.

46.1.2 PLAIN - a picture in your head

1. Picture a sound mixing desk with a hundred sliders on it.
2. Each slider changes the sound a little. You do not know what any single one does, and no slider has a label.
3. Someone plays you a recording of how the finished mix should sound.
4. Your job is to move the sliders until what comes out of the speakers matches the recording.
5. You cannot solve this by thinking. You have to move sliders and listen.
6. So you make a small move, listen, and ask one question: closer or further?
7. If closer, keep going that way. If further, go back the other way.
8. Do this over and over, for all hundred sliders, thousands of times, and the mix creeps towards the target.
9. At the end you have a hundred slider positions that produce the right sound. You still cannot say what any one slider does.
10. The sliders are the parameters. The recording is the training data. The listening is the loss. The nudging is the optimizer.

Where this comparison breaks: a real sound engineer knows what each slider does, so they aim. Nothing here aims. Also, real learning does not move one slider and listen. It computes, in one shot, the direction every slider should move, all at once. That computation is called backpropagation and it is the single most important idea in the chapter. Finally, a mix only has to sound right once. A model has to work on sounds it has never heard, which is a much harder demand and the source of nearly every practical problem in the field.

46.1.3 PLAIN - a worked example

1. Take spam detection. First, the hand-written way.

```
def is_spam_handwritten(email):
    subject = email.subject.lower()
    if "free" in subject and email.link_count > 3:
        return True
    if "viagra" in email.body.lower():
        return True
    if email.sender not in email.user_contacts:
        if email.exclamation_count > 5:
            return True
    return False
```

2. Every line of that was decided by a person. You can read it, argue with it, and fix it. Those are real advantages and they are often undervalued.
3. Now the failure. A spammer writes "FR33" instead of "free". The rule misses.

4. You add a rule for "FR33". They write "F.R.E.E". You add another rule.
5. This is a race you lose, because there are more spellings than there are hours in your life.
6. Now the learned way. You do not write rules. You write a shape.
7. The shape: give every word a number, called its weight. Add up the weights of the words present. If the total is above zero, call it spam.
8. That shape has one blank per word in the vocabulary. Say 50,000 blanks.
9. You feed in 10,000 emails already marked spam or not, and let the tuning procedure fill the blanks. Afterwards the numbers look like this:

Word	Learned weight	Effect
viagra	+4.10	strong spam signal
unsubscribe	+1.35	mild spam signal
meeting	-2.60	strong not-spam
the	+0.01	says nothing
fr33	+3.80	learned, never typed

10. Nobody told the system about "fr33". It appeared in spam in the training data, so it got a large positive weight automatically.
11. That is the real difference. The hand-written rule needed a human to notice each trick. The learned rule needs only examples containing the trick.
12. Test one email with the words "free", "meeting" and "unsubscribe", with weights +2.20, -2.60 and +1.35. Total: 2.20 - 2.60 + 1.35 = +0.95.
13. Above zero, so it says spam. It is probably wrong, because "meeting" should have carried more. That is a failure of the training data, not of the idea.
14. Both approaches fail. The difference is how they fail and how you fix them. You fix the first with a code change. You fix the second with more data.

46.1.4 PLAIN - what is really happening inside

1. Every machine learning system has exactly four pieces. Learn these four and nothing later will surprise you.
2. Piece one, the **model**: the shape of the rule, with blanks. A straight line, a tree of yes-or-no questions, or a network with a billion blanks.
3. Piece two, the **parameters**: the numbers that go in the blanks. Before training they are random or fixed. After training they are the answer.
4. Piece three, the **loss**: a single number saying how wrong the current parameters are on the examples. Lower is better. Zero would be perfect.
5. Piece four, the **optimizer**: the procedure that changes the parameters to make the loss smaller.
6. Training is a loop over these four. Predict with current parameters. Measure the loss. Work out which way each parameter should move. Move them a little. Repeat, often millions of times.
7. Notice what has been swapped. In ordinary programming, the logic is the thing you write and the data is what flows through it.
8. In machine learning, the data is the thing you provide and the logic is what comes out. The data has become the specification.
9. That single swap explains almost everything strange about the field. Bad data gives a bad model, in exactly the way a bad specification gives bad software, and for exactly the same reason.
10. It also explains why debugging feels so different. There is no line to step through. There is only a distribution of examples, and a number.

46.1.5 TECHNICAL - the engineer's version

1. The term was coined by Arthur Samuel of IBM, in his 1959 paper "Some Studies in Machine Learning Using the Game of Checkers", in the IBM Journal of Research and Development. His program ran on an IBM 701.
2. The standard formal definition is Tom Mitchell's, from his 1997 textbook Machine Learning: a program learns from experience E with respect to task T and performance measure P, if its performance at T, measured by P, improves with E.
3. Formally: choose a hypothesis class H, a set of functions parameterized by a vector theta. Choose a loss function L. Find theta that minimizes the average loss over the training data.
4. That last step is called **empirical risk minimization**. "Empirical" because you minimize measured loss on data you have, not true loss on data you do not have. The gap between the two is the whole subject of generalization.
5. The linear spam model above is logistic regression on a bag-of-words feature vector. It was the industry standard for spam filtering from the late 1990s, alongside naive Bayes, which Paul Graham popularized with his 2002 essay A Plan For Spam.
6. Side by side:

Aspect	Ordinary program	Learned model
You supply	logic	labelled examples
System supplies	output	logic (as numbers)
Debug by	reading code	inspecting data
Fails by	wrong branch	wrong distribution

7. Established fact: the "rule-shaped thing plus optimizer" description above is complete and correct for essentially every supervised learning system in production today, including the largest language models.
8. Marketing claim: that these systems reason, understand, or think. Those words have no agreed technical definition here, and vendors use them because they are unfalsifiable. Treat them as advertising until defined.
9. Active research: whether large models build internal structures that deserve to be called world models or algorithms rather than statistics. The field of mechanistic interpretability studies this. It is genuinely open, and serious researchers disagree, so do not let anyone tell you it is settled.

46.1.6 WORDS - remember these

1. Machine learning — filling in blanks in a rule from examples — fitting the parameters of a hypothesis class by minimizing empirical risk.
2. Model — the shape of the rule, with blanks — the hypothesis class, a parameterized family of functions.
3. Parameter — a number the system may change during training — a component of the weight vector theta, learned by optimization.
4. Loss — one number saying how wrong you are — a scalar objective function measuring disagreement between prediction and target.
5. Optimizer — the procedure that changes the numbers — the algorithm that updates theta to reduce the loss, usually a gradient method.
6. Curve fitting — finding settings that make a formula pass near the points — function approximation from finite samples. An accurate description, not an insult.

46.2 A short honest history

46.2.1 PLAIN - in simple words

1. This field is older than most people think, and it has failed publicly at least twice. Knowing that keeps you sane during the hype.
2. In 1943 two researchers, Warren McCulloch and Walter Pitts, wrote down a simple mathematical model of a nerve cell. That is where it starts.
3. In 1957 and 1958 Frank Rosenblatt built the **perceptron**, a machine that learned to tell simple pictures apart by adjusting numbers.
4. The press coverage was wild, and far ahead of the reality. That is a pattern you will see again in your lifetime.
5. In 1969 two respected researchers proved the perceptron could not do certain very simple things. Funding collapsed. This is called the first AI winter.
6. The fix, an algorithm for training networks with more layers, was invented several times between 1970 and 1974, and mostly ignored.
7. In 1986 it was published clearly, in a famous journal, and the field came back. Then it faded again in the early 1990s. Second winter.
8. Through the late 1990s and 2000s a different family of methods, based on clean mathematics rather than networks, did better in practice.
9. In 2012 a neural network won a big image contest by such a large margin that the argument ended in a single afternoon.
10. The thing that made 2012 possible was not a new idea. It was cheap parallel arithmetic hardware, plus a very large labelled dataset.

46.2.2 PLAIN - a picture in your head

1. Think of a seed planted in soil that is too poor to grow it.
2. The seed is the idea of learning by adjusting numbers. It was planted in 1958 and it was a good seed.
3. The soil is three things: computing power, data, and algorithms.
4. In 1958 there was almost no computing power, almost no data, and only a one-layer algorithm. The seed sprouted and died.
5. In 1986 the algorithm arrived. Compute was better, data was still thin. The seed sprouted higher and died again.
6. By 2012 all three were rich at once: graphics chips gave the compute, the internet gave the data, and the 1986 algorithm was still sitting there unchanged. The seed grew into a tree.
7. Almost nothing about the seed changed across those fifty-four years.

Where this comparison breaks: the algorithms did improve, and pretending otherwise is unfair to a great deal of careful work. Better activation functions, better initialization, better normalization and better optimizers each moved the field, and without them the 2012 result would not have trained at all. The honest version: compute and data were the necessary conditions, and a decade of unglamorous engineering was what turned "possible in principle" into "trains on a Tuesday".

46.2.3 PLAIN - a worked example

1. The overclaim of 1958, in detail, because it teaches you how to read news about this field.
2. Rosenblatt built the Mark I Perceptron at the Cornell Aeronautical Laboratory in Buffalo, New York, funded by the Office of Naval Research.
3. It had a 20 by 20 grid of 400 light sensors, so it saw a 400-pixel picture.
4. Its adjustable numbers were physical. They were potentiometers, small dials, turned by electric motors during training. You could watch the machine learn by watching the dials move.
5. It could learn to tell simple shapes apart. It genuinely worked, and that was a real achievement.
6. In July 1958 The New York Times reported the Navy's expectation that the machine would be able to "walk, talk, see, write, reproduce itself and be conscious of its existence".
7. It could sort simple shapes. The gap between those two sentences is the entire lesson of this section.

8. Eleven years later, Marvin Minsky and Seymour Papert published the book Perceptrons, in 1969, showing what one layer could not do.
9. Their killer example was XOR: output 1 if exactly one of two inputs is 1. Here is the whole problem.

```
  x2
   1 |  1        0        1 means "should output 1"
     |
   0 |  0        1
     +-------------
        0        1     x1

A single straight line must put both 1s on one side.
No straight line can. That is the entire objection.
```

10. A one-layer perceptron can only draw a straight dividing line. XOR needs two lines, or a bent one. So one layer cannot do XOR.
11. That was mathematically correct and it was fatal in practice, because nobody yet had a working way to train two layers.
12. Minsky and Papert did say multi-layer networks were more capable. The reception of the book was harsher than its text, which is common.

46.2.4 PLAIN - what is really happening inside

1. Why did the training method take so long to arrive, when the mathematics is just repeated use of the chain rule from school calculus.
2. Because nobody was looking for it in the place it was found. The reverse accumulation method was worked out for a completely different purpose.
3. In 1970 Seppo Linnainmaa, in his master's thesis at the University of Helsinki, described how to compute the derivative of a whole computation efficiently by walking backwards through it.
4. His subject was rounding error in numerical algorithms. He was not thinking about learning at all. The algorithm was correct and general anyway.
5. In 1974 Paul Werbos, in his Harvard PhD thesis, applied that idea directly to training neural networks. Very few people read it.
6. In 1986 David Rumelhart, Geoffrey Hinton and Ronald Williams published "Learning representations by back-propagating errors" in Nature. Same idea, clearly written, in a journal everyone read, with results.
7. The lesson is not about mathematics. It is that an idea has to arrive in a readable form, in a place the right people look, at a moment when the hardware can demonstrate it.
8. The second winter came after 1987 for practical reasons: networks of the day were slow to train, needed careful hand-tuning, and were beaten on real tasks by simpler methods with better theory.
9. What broke the deadlock in 2012 was throughput. A graphics chip does the same arithmetic operation on thousands of numbers at once, which is exactly the shape of a neural network's work.

46.2.5 TECHNICAL - the engineer's version

1. The timeline, with the facts you should be able to quote.

Year	Event	Why it mattered
1943	McCulloch and Pitts model	first formal neuron
1949	Hebb, Organization of Behavior	learning as weight change
1957	Rosenblatt perceptron report	first learning machine
1958	NYT overclaim	first hype cycle
1960	Widrow and Hoff ADALINE	least-mean-squares rule
1962	Novikoff convergence proof	perceptron theory

Year	Event	Why it mattered
1969	Minsky and Papert book	XOR objection
1970	Linnainmaa thesis	reverse-mode derivatives
1973	Lighthill report, UK	funding cut
1974	Werbos thesis	backprop for networks
1980	Fukushima Neocognitron	ancestor of the CNN
1982	Hopfield networks	physics enters the field
1986	Rumelhart, Hinton, Williams	backprop reaches everyone
1989	LeCun, digits at Bell Labs	first real CNN application
1989	Cybenko approximation proof	one layer is enough, in theory
1995	Cortes and Vapnik, SVM	networks fall out of fashion
1997	Hochreiter and Schmidhuber	LSTM for sequences
1998	LeCun LeNet-5	cheque reading in production
2001	Breiman random forests	strong, simple, tabular
2006	Hinton deep belief nets	the name “deep learning”
2007	NVIDIA releases CUDA	general compute on GPUs
2009	ImageNet dataset published	a large labelled image corpus
2012	AlexNet wins ILSVRC	the argument ends
2014	GANs; Adam optimizer	generation; default optimizer
2015	ResNet; batch normalization	very deep networks train
2016	AlphaGo beats Lee Sedol	4-1, March 2016
2017	Attention Is All You Need	the transformer
2020	GPT-3; scaling laws	size as a strategy
2022	ChatGPT, 30 November	public arrival
2024	Nobel Prize in Physics	Hopfield and Hinton

2. The 2012 result, precisely. AlexNet, by Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton, won the ImageNet Large Scale Visual Recognition Challenge with a top-5 error rate of 15.3 percent. The second-placed entry, which did not use deep learning, scored 26.2 percent.
3. A gap of nearly 11 percentage points in a contest usually decided by one is why the field turned in a single day.
4. The training set was about 1.2 million images across 1,000 categories. The network had roughly 60 million parameters, most of them in its fully connected layers.
5. It was trained on two NVIDIA GTX 580 cards with 3 GB of memory each, for about five to six days. The 3 GB limit is why the network was split across two cards at all. A hardware constraint shaped the architecture.
6. Chapter 22 explained what a graphics chip is and why it multiplies matrices so quickly. The short version, without repeating it: these chips run thousands of identical arithmetic operations in parallel, and a neural network is thousands of identical arithmetic operations. The fit is exact and it was an accident of history.
7. ImageNet itself, led by Fei-Fei Li and published at CVPR in 2009, was arguably the more important contribution. Without a large, clean, labelled dataset there is nothing to fit.
8. The support vector machine era rests on Corinna Cortes and Vladimir Vapnik’s 1995 paper “Support-Vector Networks”, building on the kernel trick from Bernhard Boser, Isabelle Guyon and Vapnik in 1992. From roughly 1995 to 2010 these methods usually beat neural networks on the benchmarks of the day, and had far better theory.
9. Hinton, Yoshua Bengio and Yann LeCun shared the 2018 Turing Award, presented in 2019. In October 2024 John Hopfield and Geoffrey Hinton received the Nobel Prize in Physics for foundational discoveries enabling machine learning with artificial neural networks.

10. Historical note on naming: "backpropagation" is a shortening of "backward propagation of errors", from the 1986 Nature paper title.

46.2.6 WORDS - remember these

1. Perceptron — the first machine that learned by adjusting dials — a single linear threshold unit with the 1958 update rule, convergent on linearly separable data.
2. AI winter — a period when funding and interest collapsed — the funding contractions of roughly 1974 to 1980 and 1987 to 1993.
3. XOR problem — the simple task one layer cannot do — a function that is not linearly separable, the core objection of Perceptrons, 1969.
4. Backpropagation — the way to find how each number should change — reverse- mode automatic differentiation applied to a layered network.
5. ImageNet — the giant labelled picture collection — the dataset of about 14 million images, and its annual challenge, ILSVRC, run from 2010.
6. AlexNet — the network that won in 2012 and ended the argument — an eight-layer CNN, 60 million parameters, 15.3 percent top-5 error.

46.3 Fitting a line, the foundation of everything

46.3.1 PLAIN - in simple words

1. Everything in this chapter is an elaboration of one small task: draw the best straight line through some dots on a graph.
2. Understand this properly and the rest is bookkeeping.
3. You have pairs of numbers. For each one, an input and an output.
4. Hours studied and exam mark. Size of a flat and its rent. Anything.
5. Plot them. They do not lie on a perfect line, because reality is messy, but they lean in a direction.
6. You want the line that leans the same way and sits as close to all the dots as it can.
7. A straight line is described completely by two numbers. Just two.
8. The first is how steep it is. Call it w. If w is 2, then every step right makes the line go up two.
9. The second is how high it sits. Call it b. It is the height of the line where the input is zero.
10. So the line is: output equals w times input, plus b. Written short: $y = wx + b$.
11. Those two numbers, w and b, are the only things you are allowed to change.
12. A number the system is allowed to change to fit the data is called a **parameter**. This is the definition that carries through the whole book.
13. A straight-line model has two parameters. A large language model has hundreds of billions. There is no difference in kind, only in count.

46.3.2 PLAIN - a picture in your head

1. Picture a long straight stick and a scattering of nails hammered into a board at different heights.
2. You must hold the stick against the board so it passes as close as possible to all the nails at once.
3. You have exactly two freedoms. You can slide the stick up and down, and you can tilt it.
4. Sliding up and down is b. Tilting is w. There is nothing else you can do to a straight stick.
5. Now the useful part. Suppose the nails clearly climb from left to right and your stick is flat. You can see immediately that tilting it will help.
6. Suppose the stick is at the right tilt but all the nails are above it. You can see that sliding it up will help.
7. That "you can see which way to move it" feeling is exactly the gradient. You already have the intuition. Later we just compute it instead of eyeballing.

Where this comparison breaks: with two freedoms you can see the whole board and judge both at once. With a million freedoms you cannot see anything. You can only ask, at your current position, which way is downhill, and take a step. Your entire view of the landscape is the slope under your feet.

46.3.3 PLAIN - a worked example

1. Here is the dataset used for the whole of this chapter. Five points. Learn them, because they come back three more times.

x	y
1	2
2	4
3	5
4	4
5	5

2. Plotted, they rise, but not smoothly. The fourth point dips.

```
y
6 |
5 |         *     *       (3,5) and (5,5)
4 |     *       *         (2,4) and (4,4)
3 |
2 | *                     (1,2)
1 |
0 +-----------------------
    1   2   3   4   5   x
```

3. First, a guess. Try w = 1 and b = 0, the line y = x.
4. Predictions: 1, 2, 3, 4, 5. Actual values: 2, 4, 5, 4, 5.
5. It is too low at the start and too low in the middle. Not good.
6. Now do it properly. The best straight line under the usual error measure has a formula, and we work it by hand.
7. Average of the x values: (1+2+3+4+5) / 5 = 15 / 5 = 3.
8. Average of the y values: (2+4+5+4+5) / 5 = 20 / 5 = 4.
9. Now for each point, take x minus 3, and y minus 4, and multiply them.

x-3	y-4	product	(x-3) squared
-2	-2	4	4
-1	0	0	1
0	1	0	0
1	0	0	1
2	1	2	4

10. Sum of the products: 4 + 0 + 0 + 0 + 2 = 6.
11. Sum of the squares: 4 + 1 + 0 + 1 + 4 = 10.
12. The slope is the first divided by the second: w = 6 / 10 = 0.6.
13. The height is the average y minus w times the average x: b = 4 - (0.6 x 3) = 4 - 1.8 = 2.2.
14. So the best line is y = 0.6x + 2.2. Those two numbers are the trained model. That is the whole model. Two numbers.
15. Its predictions on our five inputs: 2.8, 3.4, 4.0, 4.6, 5.2.
16. Compare to the actual values 2, 4, 5, 4, 5. Close, never exact. That is normal and expected.

46.3.4 PLAIN - what is really happening inside

1. Notice what happened there. We did not search. We solved.
2. For a straight line under squared error, there is a formula that lands on the exact best answer in one calculation. No stepping, no guessing.
3. This is called a closed-form solution. It exists for this problem because the problem is simple enough to solve with algebra.
4. Here is the important fact: for almost every model you will ever use, no such formula exists.
5. Add a bend to the curve, add a threshold, add a second layer, and the algebra stops closing. There is no formula to solve.
6. When you cannot solve, you search. You start somewhere, look at which way is downhill, and step. That is section 46.5.
7. So the line is the perfect teaching example, precisely because you can check the search against the exact answer.
8. We know the truth is w = 0.6, b = 2.2. When gradient descent creeps towards 0.6 and 2.2 later in this chapter, you will know it is working.
9. One more thing to notice. The model has no memory of the five points. After training, all that is left is 0.6 and 2.2. The data is thrown away.
10. That is true of every trained model, from this line to the largest network. The data shaped the parameters and then went home.

46.3.5 TECHNICAL - the engineer's version

1. The method is ordinary least squares. Adrien-Marie Legendre published it in
1805. Carl Friedrich Gauss published in 1809 and claimed use from 1795. The priority dispute is real and unresolved.
2. The closed form for simple linear regression: w equals the sample covariance of x and y divided by the sample variance of x, and b equals the mean of y minus w times the mean of x.
3. In matrix form for many features, the normal equation is theta equals the inverse of X-transpose-X, times X-transpose-y.
4. Computing that inverse directly is a mistake in practice. Numerically stable implementations use a QR decomposition or a singular value decomposition. The function `numpy.linalg.lstsq` uses SVD.
5. Cost: the normal equation is order n times d squared, where n is samples and d is features. At d equal to 100,000 features the d-squared term is fatal, which is one practical reason iterative methods win at scale.

```
import numpy as np
x = np.array([1, 2, 3, 4, 5], dtype=float)
y = np.array([2, 4, 5, 4, 5], dtype=float)
A = np.vstack([x, np.ones_like(x)]).T
w, b = np.linalg.lstsq(A, y, rcond=None)[0]
print(w, b)          # 0.6  2.2
```

6. Terminology, and be careful here. In statistics these two numbers are "coefficients" and the model is a "regression". In machine learning they are "weights" and "bias", and the model is a "linear layer with one output". Same equation, two vocabularies, two research communities.
7. **Parameter** has a precise meaning: a value learned from data by the optimization procedure. A **hyperparameter** is a value you choose before training, which the optimizer never touches. The learning rate is a hyperparameter. The slope is a parameter. Confusing these two is the single most common vocabulary error in the field.
8. Scale check for intuition:

Model	Parameters	Fits on
Straight line	2	paper
LeNet-5, 1998	about 60,000	any laptop
ResNet-50, 2015	25.6 million	one GPU
GPT-2 XL, 2019	1.5 billion	one GPU
GPT-3, 2020	175 billion	a cluster

46.3.6 WORDS - remember these

1. Parameter — a number the system may change to fit the data — a learned component of theta, updated by the optimizer.
2. Hyperparameter — a setting you choose before training starts — a value outside theta, fixed during optimization, tuned on validation data.
3. Weight — how strongly an input pushes the output — the multiplicative coefficient applied to an input feature.
4. Bias — the fixed offset added regardless of input — the additive intercept term, allowing the fit to leave the origin.
5. Slope — how steep the line is — the first derivative of output with respect to input, constant for a linear model.
6. Closed form — an exact answer from one formula — an analytic solution, such as the normal equation, available only for simple models.

46.4 Loss: how you measure being wrong

46.4.1 PLAIN - in simple words

1. To improve something you must first be able to score it.
2. The score here is called the **loss**. It is one number. It says how badly the current parameters do on the examples.
3. Lower is better. Zero would mean perfect on every example.
4. The obvious idea is to add up how far each prediction is from the truth.
5. But there is a trap. If one prediction is 3 too high and another is 3 too low, they cancel to zero, and the model looks perfect when it is not.
6. So you must remove the sign before adding. There are two natural ways.
7. First way: square each miss. A miss of 3 becomes 9. Squares are never negative, so nothing cancels. Average the squares. That is **mean squared error**.
8. Second way: drop the minus sign on each miss. A miss of minus 3 becomes 3. Average those. That is **mean absolute error**.
9. Both are sensible. They are not the same, and the choice changes what the model learns.
10. Squaring punishes big misses much harder. A miss of 10 counts a hundred times more than a miss of 1, not ten times more.
11. So squared error makes the model terrified of large mistakes, and willing to accept many small ones to avoid one large one.
12. Absolute error treats a miss of 10 as exactly ten misses of 1. It does not panic about outliers.
13. The loss is not a description of reality. It is your statement of what you care about. The model will optimize exactly what you asked for, including the parts you did not mean.

46.4.2 PLAIN - a picture in your head

1. Imagine you are a delivery company and you must score your drivers.
2. Scheme one: fine a driver the square of the minutes late. Five minutes late costs 25. Twenty minutes late costs 400.

3. Under that scheme a driver will do almost anything to avoid one very late delivery, and will happily be three minutes late on many.
4. Scheme two: fine one unit per minute late, always. Twenty minutes late costs 20, exactly four times what five minutes costs.
5. Under that scheme a driver optimizes total minutes and will let one delivery run very late if it saves a little time on several others.
6. Neither scheme is wrong. They encode different values.
7. And here is the point: the drivers are not cheating. They are doing exactly what you asked. If the result is bad, your scoring scheme was bad.

Where this comparison breaks: drivers understand the rule and plan around it. A model has no plan and no understanding. It only follows the slope of the number you defined. That is worse in one way, because it will exploit loopholes no honest person would notice, and better in another, because it is not trying to deceive you. The honest version: a model does not "game" the loss. The loss is its entire universe. There is nothing else to game.

46.4.3 PLAIN - a worked example

1. Take the five points again and our fitted line $y = 0.6x + 2.2$.
2. Compute every step of the mean squared error.

Point (x, y)	predicted	error	squared
(1, 2)	2.8	+0.8	0.64
(2, 4)	3.4	-0.6	0.36
(3, 5)	4.0	-1.0	1.00
(4, 4)	4.6	+0.6	0.36
(5, 5)	5.2	+0.2	0.04

3. Error means predicted minus true. Sign kept for now.
4. Sum of the squared column: $0.64 + 0.36 + 1.00 + 0.36 + 0.04 = 2.40$.
5. Divide by the number of points: $2.40 / 5 = 0.48$.
6. Mean squared error is 0.48. Written as a formula:

```
MSE = (1/n) * sum over i of ( (w*x_i + b) - y_i )^2

n = 5
  = (1/5) * (0.64 + 0.36 + 1.00 + 0.36 + 0.04)
  = (1/5) * 2.40
  = 0.48
```

7. Now mean absolute error on the same fit. Drop the signs: 0.8, 0.6, 1.0, 0.6, 0.2. Sum is 3.2. Divide by 5: 0.64.
8. Note that the errors were not squared, so the answer is in the same units as
   y. An MAE of 0.64 means "typically wrong by about 0.64 marks". An MSE of 0.48 is in squared marks, which is not a unit anyone can feel.
9. Taking the square root of MSE gives 0.693, the root mean squared error, back in the original units. That is why people quote RMSE in reports.
10. Now the demonstration that the choice matters. Add a sixth point that is wildly wrong: $x = 6$, $y = 20$. Perhaps a data entry error.
11. Refit with squared error. The best line becomes $y = 2.629x - 2.533$.
12. The slope went from 0.6 to 2.63. One bad point out of six dragged the whole line more than four times steeper.

13. Refit with absolute error. The best line becomes y = 1.5x + 0.5.
14. Still moved, but far less. The absolute-error fit largely ignored the outlier while the squared-error fit chased it.
15. That is not a small technical detail. That is the loss deciding what the model believes.

46.4.4 PLAIN - what is really happening inside

1. Now the idea that makes optimization possible: the **loss surface**.
2. The loss depends on the parameters. Change w, the loss changes. Change b, the loss changes.
3. So imagine a flat map. East-west is the value of w. North-south is the value of b. Every point on that map is one possible model.
4. At every point, compute the loss and use it as a height. Now you have a landscape.
5. High ground means bad parameters. Low ground means good ones.
6. Training is walking downhill on this landscape. That is the entire idea.
7. For our five points, here is the real landscape. Every number is a genuine mean squared error, computed for that pair of w and b.

```
        w=0.0   w=0.3   w=0.6   w=0.9   w=1.2
b=4.4   1.36    2.35    5.32   10.27   17.20
b=3.3   1.69    0.70    1.69    4.66    9.61
b=2.2   4.44    1.47    0.48*   1.47    4.44
b=1.1   9.61    4.66    1.69    0.70    1.69
b=0.0  17.20   10.27    5.32    2.35    1.36
                     * lowest point: w=0.6, b=2.2
```

8. Read it like a contour map. The 0.48 in the middle is the valley floor. That is exactly the answer we computed by formula in section 46.3.
9. Look at the diagonal. Going from top-left to bottom-right, the values stay small: 1.36, 0.70, 0.48, 0.70, 1.36. That is a long shallow trough.
10. Going the other diagonal, values shoot up fast: 1.36 to 17.20.
11. So this valley is not a round bowl. It is a long narrow ravine, steep across and shallow along. Remember that shape. It causes most of the pain in section 46.5.
12. The reason the ravine is tilted is that w and b are not independent here. Making the line steeper and lowering it can partly cancel out.

46.4.5 TECHNICAL - the engineer's version

1. The standard losses, with the situation each belongs to.

Loss	Formula core	Used for
MSE / L2	mean of (pred - y)^2	regression, default
MAE / L1	mean of abs(pred - y)	regression, outliers
Huber	L2 near 0, L1 far out	robust regression
Binary cross-entropy	-y log p - (1-y) log(1-p)	two-class
Cross-entropy	-sum y_k log p_k	many-class

2. Mean squared error is the maximum likelihood estimator under the assumption that the errors are Gaussian with constant variance. That is where it comes from. It is not an arbitrary choice.
3. Mean absolute error is the maximum likelihood estimator under Laplace- distributed errors, and its minimizer is the conditional median rather than the conditional mean. That is exactly why it resists outliers.
4. MSE has a continuous derivative everywhere. MAE has a kink at zero where the derivative jumps from -1 to +1 and is undefined at the point itself. Implementations return 0 or a subgradient there. This is a real, if minor, optimization nuisance.

5. Huber loss, from Peter Huber's 1964 paper, is quadratic within a threshold delta of zero and linear outside it. It gets MSE's smooth gradients near the optimum and MAE's outlier resistance far away. In object detection the same idea appears as smooth L1 loss.
6. For classification you almost never use MSE. Cross-entropy is standard, because its gradient with respect to the pre-activation of a softmax output is simply predicted probability minus target, which is clean and does not saturate. MSE combined with a sigmoid output produces vanishing gradients when the model is confidently wrong, which is the worst possible time.
7. The loss surface of linear regression under MSE is a convex quadratic, so it has exactly one minimum and no local traps. Its Hessian for our five points is the matrix with rows [22, 6] and [6, 2].
8. That matrix has eigenvalues 23.66 and 0.338. The ratio, 70.0, is the condition number. It is the numerical measure of the ravine we saw above, and it directly bounds how fast gradient descent can converge.
9. Observation tools: log the loss every step and plot it. In PyTorch a scalar `loss.item()` per step written to TensorBoard, Weights and Biases, or a plain CSV is enough. Anyone who trains without a loss curve is guessing.

46.4.6 WORDS - remember these

1. Loss — one number saying how wrong you are — a scalar objective evaluated on predictions and targets, to be minimized.
2. Mean squared error — average of the squared misses — the L2 loss, maximum likelihood under Gaussian noise, minimized by the conditional mean.
3. Mean absolute error — average of the misses, signs dropped — the L1 loss, minimized by the conditional median, robust to outliers.
4. Loss surface — the landscape of wrongness over parameter settings — the graph of the objective as a function of theta, in as many dimensions as there are parameters.
5. Outlier — a data point far from the rest — an observation with low probability under the assumed noise model, which dominates squared loss.
6. Cross-entropy — the standard score for choosing between categories — the negative log-likelihood of the correct class under the predicted distribution.

46.5 Gradient descent: walking downhill in fog

46.5.1 PLAIN - in simple words

1. You are standing somewhere on a hillside. Thick fog. You can see nothing beyond your own boots.
2. You want to reach the lowest point in the valley.
3. You cannot see the valley. But you can feel the ground under your feet, and feel which way is downhill.
4. So you do the only sensible thing. You take a small step in the steepest downhill direction. Then you stop and feel again. Then step again.
5. Repeat a few thousand times and you will end up at the bottom of something.
6. That is gradient descent. That is the entire algorithm. Everything else in this section is detail.
7. Translate it. The hillside is the loss surface from section 46.4. Your position is the current values of the parameters.
8. "Feeling which way is downhill" means computing, for each parameter, whether increasing it makes the loss go up or down, and by how much.
9. That per-parameter answer is called the **gradient**. It is a list with one number per parameter.
10. The size of your step is called the **learning rate**. You choose it. It is a hyperparameter, and it matters enormously.
11. Every neural network you have heard of was trained this way. There is no more sophisticated principle hiding behind it.

46.5.2 PLAIN - a picture in your head

1. Stay with the fog, but make the ground more interesting.
2. Suppose you are in a long narrow ravine, steep on the sides, almost flat along the bottom, running away into the fog.
3. You feel the ground. The steepest direction is across the ravine, not along it, because the sides are steep and the floor is nearly level.
4. So you step across. Now you are on the other side, slightly lower. You feel again. Steepest is back across.
5. You zig-zag from wall to wall, making painfully slow progress along the valley floor, which is where you actually needed to go.
6. That is the single most common failure of plain gradient descent, and our five-point problem has exactly this shape. Its condition number is 70, which means the ravine is seventy times steeper across than along.
7. The fixes all amount to remembering which way you were already going, so the zig-zags cancel and the along-valley movement adds up. That is momentum.

Where this comparison breaks: a real hiker can see their boots, feel the whole ground under them, and take a step of any size. A model computes the slope at exactly one point and then assumes the ground keeps that slope for the whole step. If the step is long and the ground curves, that assumption is wrong and you can land higher than you started. That is not a metaphor for divergence. It is literally what divergence is.

46.5.3 PLAIN - a worked example

1. First, what "the slope" means for one parameter. Move w a tiny bit and see how much the loss moves.
2. At w = 0.6, b = 2.2, the loss is 0.48. At w = 0.7, b = 2.2, it is 0.58. The loss rose by 0.10 for a rise of 0.1 in w. So the slope is about +1.0.
3. Positive slope means increasing w increases the loss. So to reduce the loss, decrease w. Always move opposite to the slope. That is the rule.
4. The update rule, for every parameter:

```
new value = old value - (learning rate) * (slope for it)
```

5. For mean squared error on a line, the two slopes have exact formulas:

```
slope for w = (2/n) * sum of ( (w*x + b - y) * x )
slope for b = (2/n) * sum of ( (w*x + b - y) )
```

6. Now run it. Start at w = 0, b = 0, which is the flat line y = 0. Learning rate 0.02. Four full steps, all arithmetic shown.
7. Step 1. Predictions are all 0. Errors, predicted minus true, are -2, -4, -5, -4, -5.
8. Slope for w: (2/5) times [(-2)(1) + (-4)(2) + (-5)(3) + (-4)(4) + (-5)(5)] = (0.4) times [-2 - 8 - 15 - 16 - 25] = 0.4 times -66 = -26.4.
9. Slope for b: (2/5) times [-2 - 4 - 5 - 4 - 5] = 0.4 times -20 = -8.0.
10. Update: w = 0 - 0.02 times (-26.4) = +0.528. b = 0 - 0.02 times (-8.0) = +0.16.
11. Both went up, which is right, because the line started far too low and far too flat.
12. Now the whole run in a table. Every number is exact.

Step	w	b	Loss
0	0.0000	0.0000	17.2000
1	0.5280	0.1600	5.5799
2	0.8045	0.2502	2.3494
3	0.9485	0.3037	1.4469
4	1.0227	0.3377	1.1904

Step	w	b	Loss

13. The slopes computed at each of those positions were:

Step	Slope for w	Slope for b
0	-26.400	-8.000
1	-13.824	-4.512
2	-7.200	-2.673
3	-3.711	-1.702
4	-1.874	-1.188

14. Read the loss column. 17.20, then 5.58, then 2.35, then 1.45, then 1.19. It falls fast at first and then slows. That shape is normal and you will see it in every real training run.
15. Read the slope table. The slopes shrink towards zero. That is the ground flattening as you approach the valley floor.
16. But look at w. It has overshot. It is 1.02 and the true answer is 0.6. And b is only 0.34 when the answer is 2.2.
17. That is the ravine. The optimizer raced along the steep w direction and is crawling along the shallow b direction.
18. Continue the run and it does get there, slowly:

Steps run	w	b	Loss
10	1.0870	0.4345	1.0471
100	0.8655	1.2416	0.6472
500	0.6176	2.1365	0.4807
2000	0.6000	2.2000	0.4800

19. Two thousand steps to find two numbers that a formula gave us instantly in section 46.3. That is the price of not having a formula. For any model worth training, there is no formula, so you pay it.
20. Notice also that the loss stops at 0.48 and does not reach zero. 0.48 is the best a straight line can do on this data. The remaining error is not a failure of the optimizer. The model simply is not a perfect description of the world, and no amount of training fixes that.

46.5.4 PLAIN - what is really happening inside

1. Now the learning rate problem, with real numbers from the same dataset.
2. Too small: with a learning rate of 0.0001 instead of 0.02, every step is two hundred times shorter. You would need roughly 400,000 steps instead of 2,000. It works. It just wastes your week.
3. Too large: here is what happens at 0.09.

Step	w	b	Loss
0	0.000	0.000	17.20
1	2.376	0.720	21.60
2	-0.341	0.027	27.22
3	2.696	0.927	34.40
4	-0.766	0.024	43.58
5	3.114	1.154	55.30

4. The loss is going up, and it is going up faster each step. The parameters are flinging from one side of the ravine to the other, further each time.
5. This is **divergence**. Left alone it ends in numbers too large to represent, and the loss prints as `nan`, meaning not a number.
6. Seeing `nan` in a loss log means, in the overwhelming majority of cases, the learning rate was too high. Halve it and restart.
7. For this problem the exact boundary is computable: any learning rate below 0.0845 converges, and anything above diverges. At 0.02 we were safe. At 0.09 we were not.
8. Now a fact worth knowing. That boundary depends on how the input numbers are scaled. Our x values run 1 to 5, so their average is 3, far from zero.
9. Subtract 3 from every x, so they run -2 to +2 with an average of zero. Fit the same model. The condition number drops from 70 to 2, and the safe learning rate rises from 0.0845 to 0.5.
10. Nothing about the problem changed. We only recentred the inputs, and training became about thirty-five times easier.
11. That is why every practical guide tells you to normalize your inputs. It is not superstition. It reshapes the ravine into a bowl.
12. Finally, local minima. In a landscape with hills and dips, you might walk into a small dip that is not the true bottom, and get stuck.
13. In two dimensions this is a serious worry. In a million dimensions it is much less of one, and the reason is worth understanding.
14. To be a local minimum, the ground must curve upward in every single direction at once. With a million directions, that is like flipping a million coins and getting heads every time.
15. What you meet instead are saddle points: up in some directions, down in others. A saddle is not a trap. There is always a way out, though progress near one is slow.

46.5.5 TECHNICAL - the engineer's version

1. The gradient of a scalar loss L with respect to a parameter vector theta is the vector of partial derivatives, written grad L. It points in the direction of steepest increase, so we step along its negative.
2. The update is theta at time t+1 equals theta at time t minus eta times grad L, where eta is the learning rate.
3. Convergence bound for a quadratic objective: plain gradient descent is stable if and only if eta is less than 2 divided by the largest eigenvalue of the Hessian. For our five-point problem that is 2 / 23.66 = 0.0845, which is exactly the boundary observed above.
4. The number of iterations to a fixed accuracy scales with the condition number, the ratio of largest to smallest eigenvalue. Ours was 70 raw and 2 after centring the inputs.
5. The three ways to choose how much data to use per step:

Variant	Data per step	Character
Batch (full)	all n samples	smooth, slow, exact
Stochastic (SGD)	1 sample	noisy, fast, jumpy
Mini-batch	32 to 8192	the practical default

6. Mini-batch is universal in practice. Batch sizes are usually powers of two, which is a convention rather than a requirement, chosen because it maps cleanly onto hardware. Common values are 32, 64, 128, 256.
7. The gradient noise from small batches is not purely harmful. It acts as a mild regularizer and helps escape sharp regions. Very large batches often generalize slightly worse unless the learning rate is scaled up, an effect documented by Keskar and colleagues in 2017 and by Goyal and colleagues in the same year, whose linear scaling rule trained ImageNet in one hour.
8. The optimizer family, in order of invention:

Optimizer	Year	Core idea
Momentum	1964, Polyak	keep a velocity
Nesterov	1983	look ahead, then step
AdaGrad	2011, Duchi	per-parameter scaling
RMSProp	2012, Hinton	decaying squared average
Adam	2014, Kingma and Ba	momentum plus RMSProp
AdamW	2017, Loshchilov	Adam, decoupled decay

9. Momentum keeps a running velocity v: v becomes beta times v plus the gradient, then theta moves by minus eta times v. Typical beta is 0.9, which means the step is roughly an average of the last ten gradients. In a ravine the across-valley components alternate in sign and cancel, while the along-valley components agree and accumulate.
10. RMSProp divides each parameter's step by the square root of a running average of that parameter's recent squared gradients. Parameters with consistently large gradients get shorter steps. Hinton introduced it in a 2012 Coursera lecture and never published it. It is cited as such.
11. Adam combines both: a running mean of the gradient, a running mean of the squared gradient, and a bias correction for the fact that both averages start at zero. Defaults from the paper: beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8. These defaults are used almost unchanged a decade later.
12. AdamW, from Ilya Loshchilov and Frank Hutter, corrects how Adam applies weight decay. It is the standard for transformer training. Typical settings for large language model pretraining are beta2 = 0.95 and weight decay 0.1.
13. Practical starting learning rates, as of 2026, as rough guidance rather than law:

Setting	Optimizer	Typical rate
Small MLP or CNN	SGD + momentum	0.1 to 0.01
CNN fine-tuning	SGD + momentum	0.001
Transformer pretraining	AdamW	1e-4 to 3e-4
Transformer fine-tuning	AdamW	1e-5 to 5e-5

14. On saddle points: Dauphin and colleagues argued in 2014, in "Identifying and attacking the saddle point problem", that saddles rather than local minima are the dominant obstacle in high-dimensional non-convex optimization. This is the standard modern view, though the loss landscape of deep networks remains an area of active research rather than settled theory.

46.5.6 WORDS - remember these

1. Gradient — which way is uphill, and how steeply, for every parameter at once — the vector of partial derivatives of the loss with respect to theta.
2. Gradient descent — repeatedly stepping downhill — iterative first-order minimization by moving against the gradient.
3. Learning rate — how big a step you take — the scalar eta multiplying the gradient in the update rule; a hyperparameter.
4. Divergence — the loss climbing instead of falling — instability from a step size above 2 over the largest Hessian eigenvalue, usually ending in NaN.
5. Momentum — remembering the direction you were already going — an exponentially weighted moving average of past gradients, coefficient beta.
6. Adam — the optimizer most people use by default — adaptive moment estimation, combining momentum with per-parameter gradient scaling.
7. Saddle point — flat ground that is up one way and down another — a stationary point whose Hessian has both positive and negative eigenvalues.

46.6 The vocabulary of training

46.6.1 PLAIN - in simple words

1. This field reuses ordinary words with narrow meanings, and half of all confusion comes from that. Fix the words now.
2. A **sample** is one example. One email. One photograph. One patient record. Also called an instance, an observation, or a row.
3. A **feature** is one measured property of a sample. The number of links in the email. The brightness of pixel 47. The patient's age.
4. A **label** is the right answer for that sample, provided by whoever made the data. "Spam". "Cat". "Recovered".
5. A **dataset** is all the samples together, with their labels.
6. Now the split, which is the most important habit in the field. You cut the dataset into three parts and treat them differently.
7. The **training set** is what the optimizer sees. The parameters are fitted to it. Usually 70 to 80 percent of the data.
8. The **validation set** is held back. You never train on it. You use it to choose between models and settings. Usually 10 to 15 percent.
9. The **test set** is held back even harder. You look at it once, at the very end, to report an honest number. Usually 10 to 15 percent.
10. Why three and not two? Because if you tune your settings by watching the validation score, you are indirectly fitting to it, so it stops being honest. The test set stays clean because you never used it to decide anything.
11. Now the counting words. An **epoch** is one complete pass through the whole training set.
12. A **batch** is a small group of samples processed together, typically 32 to 512 of them.
13. An **iteration**, also called a **step**, is one batch going through the model and producing one parameter update.
14. So: one step handles one batch, and one epoch is many steps.

46.6.2 PLAIN - a picture in your head

1. A student is preparing for an exam and has a book of 500 practice questions with answers at the back.
2. Studying the questions is training. The 500 questions are the training set.
3. The real exam has different questions on the same subject. That is the test set. The student has never seen it.
4. Student A learns the ideas. On the exam, new questions, good marks. That is a model that generalizes.
5. Student B memorizes all 500 questions and their answers word for word. Perfect on practice, lost in the exam. That is **overfitting**.
6. Student C did not study enough and gets both the practice and the exam wrong. That is **underfitting**.
7. The tell is the gap. A big gap between practice performance and exam performance means memorization, not learning.

Where this comparison breaks: a student who memorizes knows they are memorizing. A model cannot tell the difference between a real pattern and a coincidence in the data, and does not experience one as different from the other. Worse, the analogy suggests memorization is always bad. It is not that simple. Very large modern models memorize a great deal of their training data and still generalize well, which classical theory did not predict. That gap between theory and observation is real, and it is active research, not settled.

46.6.3 PLAIN - a worked example

1. Take 50,000 labelled photographs. Split 80 / 10 / 10.
2. Training set: 40,000. Validation set: 5,000. Test set: 5,000.

3. Choose a batch size of 100.
4. Steps in one epoch: 40,000 / 100 = 400.
5. Train for 20 epochs. Total steps: 20 x 400 = 8,000 parameter updates.
6. The model has seen each photograph 20 times, and the parameters have been changed 8,000 times.
7. Now read a real training curve. Both numbers are losses. Lower is better.

```
loss
2.0 |T
    | T
1.5 |  T V
    |   TV
1.0 |    T V
    |     T  V
0.5 |      TT   V
    |        TTT   V     <- validation turns up here
0.2 |           TTTTT       V
    |                TTTT       V
0.0 +-------------------------------------
     0    5   10   15   20   25   30  epoch
     T = training loss    V = validation loss
```

8. Epochs 0 to 10: both fall together. This is real learning. Nothing to do.
9. Epochs 10 to 18: training keeps falling, validation flattens. The model is starting to fit noise. Watch it.
10. Epoch 18 onwards: training still falls, validation rises. This is overfitting, and it is unambiguous.
11. The correct action is to stop at around epoch 18 and keep those parameters. That is **early stopping**, and it is the cheapest regularizer there is.
12. Here is the opposite picture, underfitting: both curves flatten early and high, at say 1.4 and 1.45, and stay there. Small gap, bad numbers.
13. That is not a stopping problem. That means the model is too simple, or the features carry no signal, or the learning rate is wrong.
14. Read the two symptoms as a pair. Big gap, low training loss: too much capacity, add regularization. Small gap, high loss on both: too little capacity, make the model bigger.

46.6.4 PLAIN - what is really happening inside

1. Underneath the two symptoms sits one trade-off, and it has a name: **bias and variance**.
2. Bias, here, does not mean unfairness. It means systematic error from a model too simple to represent the truth.
3. A straight line fitted to data shaped like a curve has high bias. It is wrong in the same way every time, no matter how much data you give it.
4. Variance means sensitivity to the particular data you happened to get.
5. A very flexible model has low bias and high variance. Show it a slightly different sample of data and it produces a noticeably different answer.
6. Total error is roughly bias squared, plus variance, plus noise that nothing can remove.
7. Simple models sit at the high-bias end. Complex models sit at the high-variance end. Classically, the best model is somewhere in between.
8. **Regularization** is anything you do to push a flexible model back towards simplicity, so you get low bias without paying full variance.
9. The four you will actually use:
10. Weight decay, also called L2: add a penalty proportional to the sum of the squared weights. Big weights become expensive, so the model prefers small ones, and small weights make gentler, smoother functions.
11. L1: penalize the sum of the absolute weights instead. This drives many weights to exactly zero, which selects features and gives a sparse model you can read.

12. Dropout: during training, at every step, randomly switch off a fraction of the units, typically half. No unit can rely on any other being present, so the network cannot build fragile chains.
13. Early stopping: stop when validation loss stops improving, as in the curve above.
14. Separately from all of those, **data augmentation**: make more training data by altering what you have in ways that do not change the label. Flip the photograph left to right, crop it, rotate it slightly, change its brightness. A flipped cat is still a cat.
15. Augmentation is often the most effective of the lot, because the real problem is usually not enough data, and this is the cheapest way to get more.

46.6.5 TECHNICAL - the engineer's version

1. Precise definitions, so the words never slip again:

Term	Exact meaning
Sample	one row of the dataset
Feature	one input dimension
Label	the target value for a sample
Batch	samples in one forward pass
Iteration / step	one parameter update
Epoch	n_train / batch_size steps

2. Real scale: ImageNet's ILSVRC training set is 1,281,167 images. At batch size 256 that is 5,004 iterations per epoch. The classic ResNet recipe runs 90 epochs, so about 450,000 parameter updates.
3. Splitting must respect structure. If several rows come from the same patient, the same user or the same day, they must not be split across train and test, or information leaks and your test number is a fantasy. Use grouped or time-based splits when the data has groups or an ordering.
4. K-fold cross-validation splits the data into k parts, trains k times with a different part held out each time, and averages. It gives a far more stable estimate on small datasets, at k times the cost. Common k is 5 or 10. It is rarely used for large deep networks purely because of cost.
5. L2 regularization adds lambda times the sum of squared weights to the loss. Its gradient contribution is 2 lambda w, so each step multiplies every weight by a factor slightly under one. With lambda 0.01 and learning rate 0.02, that factor is 0.99960, and 1,000 steps of pure decay would shrink a weight of 0.6 to 0.402.
6. In Adam, adding L2 to the loss and applying weight decay directly are not equivalent, because Adam rescales the gradient per parameter. AdamW, from Loshchilov and Hutter in 2017, decouples them. This is why AdamW replaced Adam for transformer training, and it is a real bug fix, not a preference.
7. Dropout, from Srivastava, Hinton and colleagues, published in JMLR in 2014, with an earlier 2012 preprint. At training time each unit is kept with probability 1-p. At inference nothing is dropped. Standard implementations use inverted dropout: divide activations by 1-p during training so the expected value matches, and inference needs no adjustment. Typical p is 0.5 for fully connected layers and 0.1 for transformer residual paths.
8. Batch normalization interacts with dropout badly, and modern convolutional networks often use batch normalization with little or no dropout.
9. Standard augmentations by field:

Field	Common augmentations
Images	flip, crop, colour jitter
Images, stronger	mixup, cutmix, RandAugment
Audio	time and frequency masking
Text	back-translation, dropout

Field	Common augmentations

10. The classical bias-variance curve is U-shaped: error falls, then rises with model size. Established fact for classical models.
11. Active research, and important: Belkin and colleagues showed in 2019 that for very over-parameterized models the test error falls again past the interpolation point. This is called **double descent**, and it means the classical U-shape is only the left half of the real picture. The theory is incomplete and the practical implications are still debated.
12. Practically, as of 2026, the standard recipe for very large models is not "find the sweet spot on the U". It is "make the model large, train on far more data, regularize lightly, and stop before the data runs out". That recipe is empirical, and it works.

46.6.6 WORDS - remember these

1. Sample — one example — one row of the dataset, one point in feature space.
2. Feature — one measured property — one input dimension of the model.
3. Label — the right answer for a sample — the supervision target y.
4. Epoch — one full pass over the training data — ceil(n / batch size) optimizer steps.
5. Batch — the group processed together — the set of samples whose gradients are averaged into one update.
6. Overfitting — memorizing the practice questions — low training error with a large gap to validation error; fitting sample noise.
7. Underfitting — not learning enough — high error on both sets; insufficient model capacity or bad optimization.
8. Regularization — anything that pushes towards simpler solutions — a penalty or constraint that reduces effective capacity, trading bias for variance.
9. Data augmentation — making more examples by changing existing ones — label- preserving transformations that expand the effective training distribution.

46.7 Classical machine learning before neural networks

46.7.1 PLAIN - in simple words

1. Neural networks are not the field. They are one family in it, and for a large amount of real work they are the wrong choice.
2. **Linear regression** predicts a number by adding up weighted inputs. That is section 46.3, with more than one input.
3. **Logistic regression** predicts a yes-or-no answer. It adds up weighted inputs the same way, then squashes the total into a probability between 0 and 1. Despite the name, it does classification.
4. A **decision tree** asks a series of yes-or-no questions and follows the answers to a leaf. "Is the income above 40,000? Then, is the age under 25?" You can read it out loud and a person will understand it.
5. One tree is easy to read and usually not very accurate. It clings to accidents in the data.
6. A **random forest** grows hundreds of different trees on different random parts of the data, and lets them vote. The mistakes are different, so they partly cancel.
7. **Gradient boosting** grows trees one after another, where each new tree is trained to fix the errors the current set makes. This is usually the most accurate method available for table-shaped data.
8. **k-nearest neighbours** does not train at all. To classify a new point, it finds the k most similar stored points and takes the majority answer.
9. **k-means clustering** takes unlabelled data and sorts it into k groups by similarity. No right answers are provided.
10. A **support vector machine** draws the dividing line that leaves the widest possible empty corridor between the two classes.

11. Every one of these is still in production somewhere, today, doing useful work.

46.7.2 PLAIN - a picture in your head

1. Think of a toolbox with a hammer, a screwdriver, a spanner and a saw.
2. Linear regression is the tape measure. Cheap, fast, always the first thing you reach for, and it tells you a lot before you do anything clever.
3. A decision tree is a flowchart pinned to the wall. Anybody can follow it and anybody can argue with it.
4. A random forest is asking three hundred people the same question and taking the majority. No individual is reliable. The crowd is.
5. Gradient boosting is one person answering, then a second person who only corrects the first one's mistakes, then a third who corrects what is left.
6. k-nearest neighbours is asking your five closest neighbours what they did.
7. A neural network is a machine tool. Enormously capable, expensive to set up, and absurd for hanging a picture.

Where this comparison breaks: real tools have obvious jobs and you can see when you are using the wrong one. Here you cannot, because every method returns a number for every input, and a badly chosen method still returns confident answers. The only way to know is to try several and measure honestly on held- out data. The honest version: the choice is empirical, and anyone who tells you which method wins before seeing your data is guessing.

46.7.3 PLAIN - a worked example

1. Logistic regression, worked fully. Predict whether a student passes, from hours studied.
2. The model is: z = 0.8 times hours, minus 1.2. Then squash z with the sigmoid function, which is 1 divided by (1 plus e to the minus z).
3. Student studied 3 hours. z = (0.8 x 3) - 1.2 = 2.4 - 1.2 = 1.2.
4. Sigmoid of 1.2 = 1 / (1 + e^-1.2) = 1 / (1 + 0.3012) = 0.7685.
5. So the model says 76.85 percent chance of passing.
6. If the student did pass, the loss is minus the natural log of 0.7685, which is 0.263. Small loss, because the model was mostly right.
7. If the student failed, the loss is minus the log of (1 - 0.7685) = minus log of 0.2315 = 1.463. Much bigger loss. Correctly punished.
8. Now k-nearest neighbours, worked fully. New point at (3, 4). Five stored points with known classes:

Point	Class	Distance from (3,4)
(4, 4)	B	1.00
(2, 5)	B	1.41
(3, 6)	B	2.00
(1, 2)	A	2.83
(6, 1)	A	4.24

9. With k = 3, the three closest are all B, so the answer is B.
10. With k = 5, the vote is three B against two A, so still B, but less confidently.
11. Notice there was no training. The distances were computed at question time, against every stored point. That is why it is slow to use and instant to "train".
12. Notice also that the distances mix the two axes as if they were comparable. If the first axis were rupees and the second were years, this would be nonsense. k-nearest neighbours demands that you scale your features first.

46.7.4 PLAIN - what is really happening inside

1. Now the four kinds of learning, distinguished by what you give the system.
2. **Supervised learning**: every sample comes with the right answer. The system learns to map input to answer. Spam or not. Cat or dog. House price. This is most of the industry.
3. **Unsupervised learning**: no answers at all. The system finds structure in the data itself. Grouping customers. Compressing images. Spotting anomalies.
4. **Self-supervised learning**: the answers are generated automatically from the data by hiding part of it. Take a sentence, hide a word, and the label is the hidden word.
5. That last one deserves attention, because it is how every large language model is trained. Nobody labelled the internet. The data labels itself by the trick of prediction.
6. Self-supervised learning is why scale became possible. Human labelling costs money per example. Hiding a word costs nothing, so the amount of training data is limited only by how much text exists.
7. **Reinforcement learning**: no labels, only rewards. The system acts, receives a score, and must work out which of its actions caused the score.
8. The hard part of reinforcement learning is that the reward arrives late. You lose a game of chess at move 60. Which of the 60 moves was the bad one? That is the credit assignment problem, and it makes reinforcement learning far harder and far less stable than the other three.
9. One clean way to keep them straight: supervised has answers, unsupervised has none, self-supervised manufactures its own, reinforcement has only scores.

46.7.5 TECHNICAL - the engineer's version

1. When each method is right:

Method	Best when	Weak when
Linear regression	few features, linear	curved relations
Logistic regression	need calibrated odds	complex boundaries
Decision tree	must explain decision	needs accuracy
Random forest	tabular, low tuning	wide sparse data
Gradient boosting	tabular, top accuracy	needs tuning
k-nearest neighbours	tiny data, odd shapes	many features
k-means	quick grouping	non-round clusters
SVM	small data, clear margin	over 100k samples
Neural network	images, audio, text	small tabular data

2. Origins, with dates: decision trees as CART, from Breiman, Friedman, Olshen and Stone in 1984, and ID3 from Ross Quinlan in 1986. Random forests from Leo Breiman in 2001. Gradient boosting from Jerome Friedman in 2001. k-means from Stuart Lloyd in 1957, published 1982, named by James MacQueen in 1967. k-nearest neighbours from Evelyn Fix and Joseph Hodges in 1951, with the classic analysis by Cover and Hart in 1967.
3. The practical boosting libraries: XGBoost, from Tianqi Chen and Carlos Guestrin in 2016; LightGBM from Microsoft in 2017; CatBoost from Yandex in
2017. Between them they won a very large share of Kaggle competitions on tabular data from 2016 onwards.
4. The honest note on tabular data, which is what most companies actually have. Established fact: for table-shaped data with mixed numeric and categorical columns, gradient-boosted trees have been the strongest default for a decade, and remain a leading choice in 2026. The 2022 paper by Grinsztajn, Oyallon and Varoquaux, "Why do tree-based models still outperform deep learning on typical tabular data", is the standard citation, and gives the reasons: trees handle uninformative features, irregular target functions and unnormalized columns well, and neural networks handle none of them well.

5. Active research, and genuinely moving: tabular foundation models, starting with TabPFN in 2022 and TabPFN v2 published in Nature in January 2025, are pretrained transformers that make predictions on a new small table without fitting it in the usual sense. On benchmarks of small datasets, roughly under a few tens of thousands of rows, they now often beat tuned gradient boosting. On larger tables and heavily categorical tables, boosted trees still generally lead. This boundary is moving, so check the current benchmarks rather than trusting a book on this point.
6. SVMs use the hinge loss and maximize the margin, the distance from the decision boundary to the closest points, which are the support vectors. The kernel trick, from Boser, Guyon and Vapnik in 1992, computes inner products in a high-dimensional space without ever building the coordinates. The radial basis function kernel is the usual default.
7. Cost matters: the standard SVM solver is between quadratic and cubic in the number of samples, which is why SVMs are rare above roughly 100,000 samples and were displaced partly for that reason.
8. k-means minimizes within-cluster sum of squares. Lloyd's algorithm is not guaranteed to find the global optimum, so you restart it several times. The k-means++ initialization of Arthur and Vassilvitskii, 2007, is the standard default and is what `sklearn.cluster.KMeans` uses.
9. Reinforcement learning landmarks: Q-learning from Chris Watkins in 1989, temporal difference learning from Richard Sutton in 1988, Deep Q-Networks from DeepMind in Nature in 2015, AlphaGo defeating Lee Sedol 4-1 in March 2016, and reinforcement learning from human feedback, which is how chat models are aligned after pretraining and is covered later in this book.

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split

Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.2,
                                      random_state=0)
clf = GradientBoostingClassifier(n_estimators=200,
                                 learning_rate=0.05,
                                 max_depth=3).fit(Xtr, ytr)
print(clf.score(Xte, yte))
```

10. Note the hyperparameters in that snippet: number of trees, learning rate and tree depth. Boosting has a learning rate too, for the same reason gradient descent does, and 0.05 with 200 trees is a reasonable default.

46.7.6 WORDS - remember these

1. Logistic regression — a linear model that outputs a probability — a generalized linear model with a sigmoid link, fitted by minimizing binary cross-entropy.
2. Decision tree — a readable flowchart of yes-or-no questions — a recursive partition of feature space, split by information gain or Gini impurity.
3. Random forest — many different trees voting — bagged trees with random feature subsets at each split, reducing variance.
4. Gradient boosting — each new tree fixes the last one's mistakes — stage-wise additive modelling fitting each learner to the current negative gradient.
5. Support vector machine — the widest empty corridor between two classes — a maximum-margin classifier with hinge loss, optionally kernelized.
6. Supervised learning — learning from labelled answers — estimating a mapping from inputs to targets given paired examples.
7. Self-supervised learning — the data provides its own answers — training on a pretext task built by hiding part of the input, such as next-token prediction.
8. Reinforcement learning — learning from scores, not answers — optimizing a policy to maximize expected cumulative reward from delayed feedback.

46.8 The neuron

46.8.1 PLAIN - in simple words

1. An artificial neuron is a tiny arithmetic unit. It does exactly three things and then it stops.
2. One: it takes several numbers in, and multiplies each by its own weight.
3. Two: it adds all those products together, then adds one more number called the bias.
4. Three: it passes that single total through a fixed non-straight function, and that answer is its output.
5. That is all. Multiply, add, bend. There is no memory, no decision, no state.
6. Steps one and two are just the line from section 46.3 with more inputs: output = w1 x1 + w2 x2 + w3 x3 + b.
7. Step three, the bend, is the only genuinely new part, and section 46.9 explains why it is essential.
8. A weight says how much that input matters, and in which direction. Large positive means "this pushes the answer up hard". Negative means it pushes down. Near zero means the neuron ignores it.
9. The bias is the neuron's own starting opinion, applied before any input is considered.
10. A network is thousands to billions of these, arranged in rows, each row feeding the next.

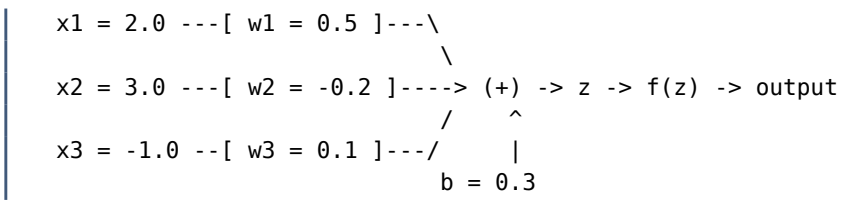
46.8.2 PLAIN - a picture in your head

1. Picture a small committee vote where the members have unequal power.
2. Each member reports a number. Each member's report is multiplied by their voting power, which is the weight.
3. The chair has a fixed opinion held before anyone speaks. That is the bias.
4. Everything is added into one total, and a rule converts the total into the committee's answer. That rule is the activation function.
5. A member with a negative weight is one whose enthusiasm counts against the proposal. That is normal and useful.
6. A member with weight zero has been muted. Their report changes nothing.

Where this comparison breaks: committee members reason about the proposal and about each other. Nothing here does. The weights are fixed numbers that were tuned by an optimizer, not opinions held by anyone. And a real committee can deadlock or refuse to answer. A neuron always produces exactly one number, for every input, instantly, forever.

46.8.3 PLAIN - a worked example

1. One neuron, three inputs. Weights 0.5, -0.2 and 0.1. Bias 0.3.
2. Inputs arrive: 2.0, 3.0 and -1.0.



3. Multiply each input by its weight:
4. 2.0 x 0.5 = 1.0
5. 3.0 x -0.2 = -0.6
6. -1.0 x 0.1 = -0.1
7. Add them: 1.0 - 0.6 - 0.1 = 0.3
8. Add the bias: 0.3 + 0.3 = 0.6. Call this z, the pre-activation.
9. Now apply the bend. Three common choices, all on the same z = 0.6:

Function	Output at z = 0.6
ReLU	0.600
Sigmoid	0.646
tanh	0.537

10. That is a complete neuron evaluation. Six multiplications and additions, then one function call.
11. Read the weights. Input one matters most and pushes up. Input two pushes down, less strongly. Input three barely matters at all.
12. And notice: input two had the largest raw value, 3.0, but contributed less than input one, because its weight was smaller. Size of input and importance of input are different things.

46.8.4 PLAIN - what is really happening inside

1. Geometrically, the weighted sum plus bias defines a plane cutting through the input space.
2. The neuron is measuring which side of that plane the input lies on, and how far from it.
3. The weights set the direction the plane faces. The bias sets how far the plane sits from the origin.
4. So a single neuron with a threshold is a straight-line separator, and nothing more. It is exactly the 1958 perceptron.
5. This is why one neuron cannot do XOR: one plane, two regions, and XOR needs the two "1" corners on the same side, which no plane achieves.
6. Layers fix it, because the second layer separates a space that the first layer has already bent.
7. Now the honest part about the brain. A neuron in your head is a living cell. It fires discrete spikes whose timing carries information. It has thousands of synapses with their own chemistry, it changes its own structure, it is regulated by hormones, and it is affected by nearby cells that are not even neurons.
8. The artificial version is a weighted sum and a bend. It does not spike, has no timing, no chemistry and no internal state.
9. The name came from the 1943 McCulloch and Pitts paper, which was genuinely an attempt to model nerve cells. The name stuck long after the modelling ambition was dropped.
10. The honest version: the brain analogy is historical and, today, mostly marketing. Calling it a neuron tells you where the idea came from. It tells you nothing about how brains work, and nothing about how networks work. Real computational neuroscience uses far more detailed models and does not use these.

46.8.5 TECHNICAL - the engineer's version

1. Formally, for input vector x in R^n, weight vector w in R^n and bias b in R: z = w-transpose x + b, and a = f(z), where f is the activation.
2. z is the pre-activation or logit. a is the activation or unit output.
3. Cost per neuron: n multiply-accumulate operations, counted as 2n floating-point operations, plus one activation evaluation.
4. Storage per neuron: n + 1 parameters. In float32 that is $4(n+1)$ bytes; in bfloat16, $2(n+1)$ bytes.
5. The set of points where z equals zero is a hyperplane of dimension n-1. The vector w is its normal, and the perpendicular distance from the origin is the absolute value of b divided by the Euclidean norm of w.
6. The original 1958 perceptron used the Heaviside step as f, which has a derivative of zero everywhere it is defined and is undefined at zero. That is precisely why perceptrons could not be trained by gradient descent, and why the switch to smooth activations mattered so much.
7. Biological comparison, with real figures:

Property	Biological neuron	Artificial unit
Signal	timed voltage spikes	one real number
Inputs	~1,000 to 10,000	as designed
Firing rate	up to ~200 Hz	not applicable
State	membrane potential	none

8. The human brain has roughly 86 billion neurons, a figure from Suzana Herculano-Houzel's 2009 work, with of the order of 100 trillion synapses. Parameter counts of large models are sometimes compared to synapse counts. That comparison is not meaningful, because the units are not comparable, and you should treat it as a marketing claim.
9. Spiking neural networks and neuromorphic hardware, such as Intel's Loihi 2 and IBM's TrueNorth, do model timing and spikes. They are active research and are not what runs any deployed large model today.

46.8.6 WORDS - remember these

1. Neuron / unit — a weighted sum followed by a bend — a parameterized affine map composed with a fixed non-linearity.
2. Weight — how much an input matters and in which direction — a learned coefficient in the affine map.
3. Bias — the unit's fixed starting offset — the additive constant, shifting the hyperplane off the origin.
4. Pre-activation — the total before the bend — z, the logit; the affine output before f is applied.
5. Activation — the neuron's final output — a = f(z), the value passed to the next layer.
6. Hyperplane — the flat surface a neuron measures distance from — the set where w-transpose x + b equals zero, of dimension n-1.

46.9 Activation functions

46.9.1 PLAIN - in simple words

1. Every neuron ends with a bend, and without the bend the whole network is worthless. Here is the proof, in two lines of algebra.
2. Take two layers with no bend. Layer one computes W1 x + b1. Layer two takes that and computes W2 (W1 x + b1) + b2.
3. Multiply out: W2 W1 x + W2 b1 + b2. Now let W' be W2 W1 and let b' be W2 b1 + b2. Both are just fixed matrices of numbers.
4. The result is W' x + b'. That is one single layer.
5. So a hundred layers with no bend collapse into one layer. Depth buys nothing at all. Not less than expected. Exactly nothing.
6. The bend is what stops the collapse. Once you put a non-straight function between the layers, the multiplication cannot be folded up, and the second layer sees a genuinely reshaped space.
7. The function that provides the bend is called the **activation function**.
8. There are only a handful in common use, and almost all of modern practice is one of them: ReLU, which is "if the number is negative, make it zero; otherwise leave it alone".
9. That is genuinely the whole of ReLU. One comparison and one selection.

46.9.2 PLAIN - a picture in your head

1. Picture folding a flat sheet of paper.
2. A linear layer stretches, rotates and slides the sheet. However many times you do that, the sheet stays flat, and you could have done it in one move.
3. ReLU folds it. Everything below the crease is flattened onto the crease line, and everything above is untouched.

4. Fold, stretch, fold, stretch, twenty times, and the sheet is a complicated crumpled shape that no single stretch could produce.
5. Now a straight cut through the crumpled sheet is, when you unfold it, a complicated curved boundary in the original space.
6. That is exactly what a deep network with ReLU is: many folds, then one straight cut at the end.

Where this comparison breaks: paper folds along a line you choose by hand. Here the crease positions are the weights and biases, and they are learned. Also, paper is two-dimensional, so you can see the folds. In 512 dimensions with thousands of folds there is nothing to see, and the intuition stops helping long before the mathematics does.

46.9.3 PLAIN - a worked example

1. The same input z, through four different activations.

z	sigmoid	tanh
-2.0	0.1192	-0.9640
-1.0	0.2689	-0.7616
0.0	0.5000	0.0000
1.0	0.7311	0.7616
2.0	0.8808	0.9640

z	ReLU	GELU
-2.0	0.00	-0.0455
-1.0	0.00	-0.1587
0.0	0.00	0.0000
1.0	1.00	0.8413
2.0	2.00	1.9545

2. Notice sigmoid never leaves the range 0 to 1, and tanh never leaves -1 to 1. Both are squashers.
3. Notice ReLU has no upper limit. Feed it 1000 and it returns 1000.
4. Notice GELU is almost ReLU for large inputs but is smooth near zero and allows a small negative output. That smoothness is why it is preferred in transformers.
5. Now the vanishing gradient, worked with numbers. The steepness of sigmoid at any point is s times (1 - s), where s is its output.
6. At z = 0, s = 0.5, so the steepness is 0.5 x 0.5 = 0.25. That is the maximum it ever reaches.
7. At z = 6, s = 0.9975, so the steepness is 0.9975 x 0.0025 = 0.0025. A hundred times flatter.
8. Now recall that backpropagation multiplies these steepness values together, once per layer. Best case, every layer contributes 0.25.

Layers of sigmoid	Best-case gradient factor
1	0.25
3	0.0156
5	0.00098
10	0.00000095

9. After ten sigmoid layers, the gradient reaching the first layer is at best about one millionth of the gradient at the output, and in practice far less.

10. So the early layers barely move. They are effectively frozen at their random starting values. The network trains its last two layers and nothing else.
11. That is the vanishing gradient problem, and it is the single technical reason deep networks did not work before roughly 2010.
12. ReLU's steepness is exactly 1 for every positive input. Multiply 1 by itself fifty times and you still have 1. The problem disappears.

46.9.4 PLAIN - what is really happening inside

1. So why did ReLU win, when it looks crude next to a smooth curve?
2. Reason one, the gradient. Slope of exactly 1 on the positive side, so gradients pass through many layers undamaged.
3. Reason two, speed. ReLU is one comparison. Sigmoid needs an exponential, which is many times more expensive, and it is evaluated billions of times.
4. Reason three, sparsity. Roughly half the units output exactly zero for a given input. Zeros propagate nothing and cost nothing, and the resulting sparse representations seem to help.
5. ReLU has one real failure, called the dying ReLU. If a unit's weights land such that its pre-activation is negative for every input in the data, its output is always zero, its gradient is always zero, and it can never recover. It is dead for the rest of training.
6. Leaky ReLU fixes this by giving the negative side a small slope, typically 0.01, instead of zero. A dead unit still receives a trickle of gradient.
7. In practice, plain ReLU with sensible initialization dies rarely enough that leaky ReLU is optional rather than standard, which is a somewhat surprising empirical result.
8. GELU and SiLU are the modern smooth alternatives. Both look like ReLU from a distance, but curve gently through zero and dip slightly negative just below it.
9. There is no strong theory for why that small dip helps. The honest answer is that it measured better on transformer benchmarks and the field adopted it.

46.9.5 TECHNICAL - the engineer's version

1. The reference table:

Name	Definition	Range
Sigmoid	1/(1+e^-z)	(0, 1)
Tanh	$(e^{z-e}-z)/(e^{z+e}-z)$	(-1, 1)
ReLU	max(0, z)	[0, inf)
Leaky ReLU	max(0.01z, z)	(-inf, inf)
GELU	z * Phi(z)	(-0.17, inf)
SiLU / Swish	z * sigmoid(z)	(-0.28, inf)
Softmax	e^z_i / sum e^z_j	(0, 1), sums to 1

2. Where each is used today, as of 2026:

Activation	Typical use
ReLU	CNNs, most MLPs
GELU	BERT, GPT-2, GPT-3
SiLU / SwiGLU	Llama, Mistral, Qwen
Sigmoid	binary output layer only
Tanh	LSTM gates, some RL heads
Softmax	multi-class output layer

3. Derivatives, which is what backpropagation actually needs. Sigmoid: s(1-s), maximum 0.25 at z = 0. Tanh: 1 - tanh squared, maximum 1.0 at z = 0. ReLU: 1 for z above 0, 0 below, undefined at exactly 0, where implementations return 0 by convention.
4. Tanh's maximum derivative of 1.0 versus sigmoid's 0.25 is why tanh was preferred over sigmoid through the 1990s and 2000s. It is zero-centred, which also keeps the mean activation near zero. It still saturates at both ends, so it only postpones the problem.
5. GELU, from Dan Hendrycks and Kevin Gimpel in 2016, is z times the standard normal cumulative distribution function. The common `tanh` approximation is 0.5z(1 + tanh(sqrt(2/pi)(z + 0.044715 z^3))). Both forms are shipped in PyTorch, selected by the `approximate` argument.
6. SiLU, also called Swish, appeared in Elfwing, Uchibe and Doya in 2017 and in Ramachandran, Zoph and Le's neural architecture search paper the same year.
7. SwiGLU, from Noam Shazeer in 2020, is a gated variant using SiLU. It is the feedforward activation in the Llama family and much of the current open model ecosystem. It uses three weight matrices rather than two, so implementations shrink the hidden dimension to keep the parameter count equal.
8. Softmax is not a per-neuron activation. It operates on a whole output vector, turning logits into a probability distribution. Implementations always subtract the maximum logit before exponentiating, to avoid overflow. This is called the log-sum-exp trick and it is an implementation detail every library handles for you.
9. The vanishing gradient problem was identified in Sepp Hochreiter's 1991 diploma thesis and analysed by Bengio, Simard and Frasconi in 1994. The move to ReLU came through Nair and Hinton in 2010 and Glorot, Bordes and Bengio in 2011, and reached everyone through AlexNet in 2012.

46.9.6 WORDS - remember these

1. Activation function — the bend after the weighted sum — the elementwise non-linearity f applied to the pre-activation.
2. ReLU — negative becomes zero, positive passes through — the rectified linear unit, max(0, z), with derivative 1 on the positive side.
3. Vanishing gradient — the learning signal fading out before it reaches the early layers — the product of many derivatives below 1 decaying towards zero exponentially in depth.
4. Saturation — the region where a squashing function stops responding — where the derivative is near zero because the input is far from zero.
5. Dying ReLU — a unit stuck outputting zero forever — a unit whose pre-activation is negative across the entire data distribution, so its gradient is permanently zero.
6. Softmax — turning scores into probabilities that add to one — the normalized exponential over a logit vector, used with cross-entropy.

46.10 Layers and depth

46.10.1 PLAIN - in simple words

1. A **layer** is a row of neurons that all look at the same inputs and all produce outputs at the same time.
2. Neurons in one layer do not talk to each other. They work in parallel, which is exactly why this runs so well on the hardware from Chapter 22.
3. Stack layers so each one's outputs are the next one's inputs, and you have a network.
4. The **input layer** is not really a layer of neurons. It is just the numbers you feed in. Counting it is a convention some books follow and some do not.
5. **Hidden layers** are the middle ones. They are called hidden because nothing outside the network ever sees their values.
6. The **output layer** is the last one, and its size is set by the task. One unit for a single number. Ten units for ten categories.
7. **Width** is how many neurons are in a layer. **Depth** is how many layers there are.

8. "Deep learning" means, roughly, more than two or three hidden layers. There is no official threshold. It is a marketing term that stuck.
9. What depth actually buys you is composition. Early layers find simple patterns, later layers combine those into complicated ones.
10. In a picture network, the first layer finds edges, the third finds corners and textures, the tenth finds eyes and wheels, the last finds faces and cars. Nobody designed that. It emerges from training.

46.10.2 PLAIN - a picture in your head

1. Think of a factory production line rather than one craftsman.
2. Station one takes raw metal and cuts flat pieces. That is all it does, and it does it to everything.
3. Station two takes flat pieces and bends them into brackets.
4. Station three joins brackets into frames. Station four fits frames into cabinets.
5. No station understands the cabinet. Each does one narrow transformation of whatever arrives.
6. Depth is the number of stations. Width is how many workers stand at each station doing that station's job in parallel.
7. Adding stations lets you build more complicated products. Adding workers per station lets you build more different things at each stage.

Where this comparison breaks: a factory line is designed, and someone knows what each station is for. Here nobody assigns jobs. The stations arrive at their own division of labour during training, and often it does not decompose neatly at all. Some units respond to combinations of things that have no name. The tidy edges-to-faces story is real for convolutional image networks and was demonstrated by visualization work in the 2010s, but it is a summary of typical behaviour, not a law.

46.10.3 PLAIN - a worked example

1. A network for handwritten digits. Input is a 28 by 28 greyscale image, which is 784 numbers. Output is 10 scores, one per digit.
2. Shape: 784 inputs, then a hidden layer of 128, then a hidden layer of 64, then 10 outputs.

```
[784 inputs] -> [128 hidden] -> [64 hidden] -> [10 outputs]
     |               |               |              |
  pixels          ReLU            ReLU          softmax
```

3. Parameter count, layer by layer. Each layer has (inputs x outputs) weights plus (outputs) biases.
4. Layer 1: 784 x 128 = 100,352 weights, plus 128 biases = 100,480.
5. Layer 2: 128 x 64 = 8,192 weights, plus 64 biases = 8,256.
6. Layer 3: 64 x 10 = 640 weights, plus 10 biases = 650.
7. Total: 100,480 + 8,256 + 650 = 109,386 parameters.
8. Note how lopsided that is. The first layer holds 92 percent of the parameters, purely because 784 inputs is a lot to connect to.
9. This is a real network. Trained for a few minutes it reaches roughly 97 to 98 percent accuracy on the MNIST test set.
10. Now change the shape and see the trade. Width 512 in one hidden layer: 784 x 512 + 512, then 512 x 10 + 10 = 401,920 + 5,130 = 407,050 parameters.
11. Four times the parameters, one layer fewer, and typically slightly worse accuracy on this task. Depth is usually the more efficient way to spend parameters, though not without limit.

46.10.4 PLAIN - what is really happening inside

1. The **universal approximation theorem** is the famous result here, and it is almost always quoted wrongly. Here it is stated correctly.
2. A feedforward network with a single hidden layer, containing enough neurons, and a suitable non-polynomial activation, can approximate any continuous function on a closed bounded region, to any accuracy you like.
3. Read the conditions again. Continuous. Closed and bounded. Any accuracy you like, but "enough neurons" is not bounded.
4. Now the three things it does not say, which is where the misuse happens.
5. It does not say how many neurons. "Enough" can mean exponentially many in the number of inputs. The theorem is silent on the count.
6. It does not say you can find those weights. It proves such weights exist. It gives no procedure, and gradient descent may never reach them.
7. It does not say anything about new data. Approximating a function on the region you sampled says nothing about behaviour outside it.
8. So the theorem is an existence result. It tells you the model family is rich enough. It does not tell you the model is learnable, efficient or useful.
9. The practical answer to why we use depth at all: for many functions, a deep network needs exponentially fewer units than a shallow one to reach the same accuracy. Depth is about efficiency, not possibility.
10. That is not folklore. There are proven separation results, for example Telgarsky in 2016, exhibiting functions a deep network represents with few units that any shallow network needs exponentially many units to match.

46.10.5 TECHNICAL - the engineer's version

1. A fully connected layer with n inputs and m outputs has n x m weights and m biases, and costs 2nm floating-point operations per sample.
2. Layer terminology in code: `nn.Linear(in, out)` in PyTorch, `Dense(units)` in Keras, `nn.Dense(features)` in Flax. All the same object.
3. Depth in real architectures, with dates:

Model	Year	Depth
LeNet-5	1998	7 layers
AlexNet	2012	8 layers
VGG-19	2014	19 layers
GoogLeNet	2014	22 layers
ResNet-152	2015	152 layers
GPT-3	2020	96 blocks
Llama 3 70B	2024	80 blocks

4. Depth beyond about 20 plain layers did not train at all before residual connections. ResNet, from Kaiming He and colleagues at Microsoft Research in 2015, adds the input of a block to its output, so the block only has to learn a correction. That single change took usable depth from about 20 to over 1,000 and is the most important architectural idea of the decade.
5. Approximation theory: George Cybenko proved the sigmoid case in 1989. Kurt Hornik generalized it in 1991, showing that the multilayer structure, not the choice of activation, is what gives universality. Leshno and colleagues showed in 1993 that a non-polynomial activation is the precise condition.
6. There are also dual results for width. Lu and colleagues showed in 2017 that networks of bounded width and unbounded depth are universal, with a width threshold around the input dimension plus four.

7. Established fact: depth gives exponential representational efficiency for some function classes, and residual connections make great depth trainable.
8. Active research: exactly why depth helps optimization, not just representation, and why over-parameterized networks generalize at all. There is no complete theory. Anyone claiming otherwise is overselling.

46.10.6 WORDS - remember these

1. Layer — a row of neurons computed together — an affine transform followed by an elementwise non-linearity.
2. Hidden layer — a middle layer nothing outside ever sees — any layer that is neither the input nor the output.
3. Width — how many neurons in one layer — the output dimension of that layer.
4. Depth — how many layers deep the network is — the number of composed non-linear transformations.
5. Universal approximation theorem — one wide layer can copy any smooth function — for any continuous f on a compact set and any epsilon, a sufficiently wide single-hidden-layer network exists within epsilon. An existence result only.
6. Residual connection — adding a block's input to its output — a skip connection giving gradients a direct path, from ResNet in 2015.

46.11 The forward pass as matrix multiplication

46.11.1 PLAIN - in simple words

1. Running a network on an input is called the **forward pass**. Numbers go in at one end and come out the other.
2. Doing it one neuron at a time, as in section 46.8, is correct but slow to describe and slower to compute.
3. Every neuron in a layer does the same thing to the same inputs, with different weights. That is exactly what a matrix multiplication is.
4. So stack the weights of all the neurons in a layer as rows of a grid. Then one grid-times-list operation computes the whole layer at once.
5. That grid is the **weight matrix**. Its height is the number of neurons and its width is the number of inputs.
6. The whole forward pass becomes: multiply by a matrix, add a bias list, apply the bend. Repeat once per layer. That is all a network is.
7. Chapter 22 explained the matrix multiplication and the chip that does it. This is the same operation, unchanged, on bigger grids.
8. That is not a coincidence or an analogy. It is the same arithmetic, and it is the entire reason graphics hardware runs this field.

46.11.2 PLAIN - a picture in your head

1. Imagine a spreadsheet where a whole column of results is computed by one formula dragged down.
2. You do not calculate each cell separately. You describe the pattern once and the machine fills the column.
3. A matrix multiplication is that idea in two directions at once, and on hardware built to do thousands of the multiplications simultaneously.

Where this comparison breaks: a spreadsheet computes cells one after another and just hides it from you. A graphics chip genuinely computes thousands at the same instant, in separate arithmetic units. The parallelism is real, not presentational, and that difference is worth roughly a thousandfold in speed.

46.11.3 PLAIN - a worked example

1. A two-layer network. Two inputs, three hidden units with ReLU, one output.
2. Input x = [1.0, 2.0].
3. Layer 1 weight matrix W1, three rows of two, and bias b1:

```
W1 = [  0.5  -0.2 ]        b1 = [  0.1 ]
     [  0.3   0.8 ]             [ -0.1 ]
     [ -0.4   0.1 ]             [  0.2 ]
```

4. Multiply row by row. Row 1: (0.5 x 1.0) + (-0.2 x 2.0) = 0.5 - 0.4 = 0.1. Add bias 0.1, giving 0.2.
5. Row 2: (0.3 x 1.0) + (0.8 x 2.0) = 0.3 + 1.6 = 1.9. Add bias -0.1, giving 1.8.
6. Row 3: (-0.4 x 1.0) + (0.1 x 2.0) = -0.4 + 0.2 = -0.2. Add bias 0.2, giving 0.0.
7. So z1 = [0.2, 1.8, 0.0]. Apply ReLU: nothing is negative, so a1 is unchanged at [0.2, 1.8, 0.0].
8. Layer 2 is one row of three: W2 = [1.0, -0.5, 0.25], and b2 = [0.05].
9. Multiply: (1.0 x 0.2) + (-0.5 x 1.8) + (0.25 x 0.0) = 0.2 - 0.9 + 0 = -0.7. Add bias: -0.7 + 0.05 = -0.65.
10. Output is -0.65. If this were a two-class problem, apply sigmoid: 0.343, so 34.3 percent.
11. Total parameters: (3 x 2 + 3) + (1 x 3 + 1) = 9 + 4 = 13. Total arithmetic: 9 multiply-adds. That is the whole network.
12. Written compactly, the entire forward pass is two lines:

```
a1  = ReLU( W1 @ x  + b1 )
out = sigmoid( W2 @ a1 + b2 )
```

46.11.4 PLAIN - what is really happening inside

1. Now count the work for a real network: the 784 to 128 to 64 to 10 digit classifier from section 46.10.
2. Each weight is used exactly once per input, in one multiply and one add. So count multiply-accumulate operations, then double for floating-point operations.

Layer	Multiply-adds	FLOPs
784 -> 128	100,352	200,704
128 -> 64	8,192	16,384
64 -> 10	640	1,280
Total	109,184	218,368

3. About 218,000 floating-point operations to classify one digit. A modern laptop core does billions per second, so this takes microseconds.
4. Useful rule to memorize: a forward pass costs roughly two floating-point operations per parameter. Here, 218,368 divided by 109,386 is 1.996.
5. That rule holds for dense layers and for transformers, so it lets you estimate the cost of any model you know the size of.
6. Now batching. Instead of one input vector, stack 64 inputs as columns of a matrix. One matrix-times-matrix computes all 64 forward passes.
7. Same total arithmetic, 64 times 218,368 which is about 14 million operations, but each weight is loaded from memory once and used 64 times.
8. Chapter 22 made the point that memory bandwidth, not arithmetic, is usually the limit. Batching is how you escape it, and it is why batch size is a performance knob and not only a statistical one.

46.11.5 TECHNICAL - the engineer's version

1. For a layer with weight matrix W of shape (m, n), input X of shape (n, B) where B is the batch size, and bias b of shape (m, 1): Z = W X + b, broadcast over the batch dimension, then A = f(Z).

2. This is a GEMM, a general matrix-matrix multiply, the operation every BLAS library optimizes above all others. On NVIDIA hardware it runs through cuBLAS or CUTLASS, and on tensor cores where precision allows.
3. Cost: 2 m n B FLOPs. Memory traffic if unbatched and uncached: m n reads for the weights per sample. The arithmetic intensity, operations per byte, rises linearly with B until the compute units saturate.
4. Convention warning: mathematics texts write W x with x as a column vector. PyTorch's `nn.Linear` stores weight of shape (out_features, in_features) but computes `x @ W.T + b` with x of shape (batch, in_features), so the batch dimension is first. Both conventions are in daily use and mixing them up is a common source of shape errors.
5. Real forward-pass costs, per single sample:

Model	Parameters	FLOPs per forward
MNIST MLP above	109 thousand	0.22 million
ResNet-50, 224px	25.6 million	about 8.2 billion
GPT-3, per token	175 billion	about 350 billion

6. Note ResNet-50 breaks the two-FLOPs-per-parameter rule badly, at about 320 FLOPs per parameter. That is because convolution reuses each weight at every spatial position. The rule applies to dense layers, and transformers are mostly dense layers, which is why it works there.
7. The training-cost rule of thumb, from Kaplan and colleagues in 2020, is about 6 N D floating-point operations to train a model of N parameters on D tokens: roughly 2 N for the forward pass and 4 N for the backward pass.
8. Precision matters enormously here, and Chapter 22 gave the per-chip figures. The short summary without repeating them: halving the bits per number roughly doubles the achievable rate on the same silicon, which is why training moved from FP32 to mixed BF16 and inference is moving to FP8 and 4-bit integers.

46.11.6 WORDS - remember these

1. Forward pass — running the input through to the output — evaluating the composed function, layer by layer, with no gradient computation.
2. Weight matrix — the grid holding a layer's weights — a matrix of shape (out_features, in_features).
3. GEMM — the matrix multiply that dominates the work — general matrix-matrix multiply, the core BLAS level-3 routine.
4. Batch dimension — many inputs processed at once — the axis over which samples are stacked to raise arithmetic intensity.
5. FLOP — one floating-point operation — one multiply or one add; a forward pass through dense layers costs about 2 per parameter.

46.12 Backpropagation

46.12.1 PLAIN - in simple words

1. The network produced an answer. The answer was wrong by some amount. Now what?
2. Every weight in the network contributed something to that wrong answer. Some contributed a lot. Some contributed almost nothing. Some pushed in the right direction and were outvoted.
3. What we need is, for each weight, its share of the blame: if this one weight were slightly larger, would the error get better or worse, and by how much?
4. **Backpropagation** is the procedure that works out every weight's share.
5. It works backwards. It starts at the output, where the error is directly measurable, and moves towards the input, one layer at a time.

6. At the output layer, the blame is obvious. Each output weight fed directly into the answer, so its share is easy to compute.
7. Then the trick. Having computed how much blame belongs to each unit in the last hidden layer, that blame is passed further back, split among the weights that fed those units, in proportion to how strongly each fed them.
8. Repeat until you reach the input. Now every weight in the network has a number attached: its share of the error.
9. Then every weight is nudged a little, against its share. Weights that made things worse move down. Weights that helped move up.
10. That is it. Compute forwards, assign blame backwards, nudge everything. Millions of times.
11. The reason this matters so much is efficiency. It computes the share for every weight in one backward sweep, costing about twice one forward pass, regardless of how many weights there are.

46.12.2 PLAIN - a picture in your head

1. A large company delivers a project three weeks late. The chief executive wants to know who is responsible.
2. She does not interview every one of ten thousand employees. That would take longer than the project.
3. Instead she asks four department heads: how much of the delay came through your department? They can answer, because they sit right next to the outcome.
4. Manufacturing says 60 percent, logistics 30 percent, design 10 percent.
5. Now each head asks their own team leads the same question, splitting only their own share. The manufacturing head splits her 60 percent among five teams.
6. Each team lead splits their portion among their engineers. In four rounds, every one of ten thousand employees has a number.
7. Four conversations per level, not ten thousand interviews. The saving is the whole point.
8. And the blame is proportional to influence. Someone whose work barely touched the delivery date receives a share near zero automatically.

Where this comparison breaks, and it matters. First, blame in a company is a judgement, and people argue. Here it is an exact derivative with a single correct value. Second, real blame implies fault. Here a "share" can be negative, which simply means that increasing that weight would reduce the error, so the weight should grow. Third, and most important, the shares are only valid for a tiny nudge. They tell you the effect of changing one weight by a hair while everything else stays fixed. Change all the weights at once by a large amount and the shares no longer describe reality. That is exactly why the learning rate must be small.

46.12.3 PLAIN - a worked example

1. Now the chain rule, which is the exact rule behind the blame splitting.
2. If a affects b, and b affects c, then the effect of a on c is the effect of a on b multiplied by the effect of b on c.
3. If turning a dial doubles a pressure, and doubling the pressure triples a flow, then turning the dial multiplies the flow by six. Effects along a chain multiply.
4. A network is a long chain, so a weight's effect on the loss is the product of the effects at every step between that weight and the loss.
5. Here is the network we will do completely. Two inputs, two hidden units with sigmoid, one output unit with sigmoid, and the loss is half the squared error.

```
x1=0.5 --w11=0.8--> h1 --v1=0.5--\
       \--w21=0.2-> h2 --v2=-0.3--> out -> loss
x2=-1.0-/w12=-0.4, w22=0.6/
biases: b1=0.1, b2=-0.2, c=0.2     target y = 1.0
```

6. Forward pass. Hidden unit 1: z = (0.8 x 0.5) + (-0.4 x -1.0) + 0.1 = 0.4 + 0.4 + 0.1 = 0.9. Sigmoid of 0.9 = 0.710950.
7. Hidden unit 2: z = (0.2 x 0.5) + (0.6 x -1.0) + (-0.2) = 0.1 - 0.6 - 0.2 = -0.7. Sigmoid of -0.7 = 0.331812.
8. Output: z = (0.5 x 0.710950) + (-0.3 x 0.331812) + 0.2 = 0.355475 - 0.099544 + 0.2 = 0.455931. Sigmoid = 0.612048.
9. Loss = 0.5 x (0.612048 - 1.0)^2 = 0.5 x 0.150506 = 0.075253.
10. Backward pass, output layer first.
11. Effect of the output on the loss: predicted minus target = 0.612048 - 1.0 = -0.387952.
12. Effect of the pre-activation on the output, which is the sigmoid derivative: 0.612048 x (1 - 0.612048) = 0.237445.
13. Multiply them. This product is the output unit's blame share, called delta: -0.387952 x 0.237445 = -0.092117.
14. Now each output weight's share is that delta times the activation it multiplied:
15. Share for v1 = -0.092117 x 0.710950 = -0.065491.
16. Share for v2 = -0.092117 x 0.331812 = -0.030566.
17. Share for the output bias c = -0.092117, because a bias multiplies 1.
18. Now push the blame back to the hidden units. Each hidden unit's blame is the output delta times the weight connecting them, times that unit's own sigmoid derivative.
19. Hidden unit 1: -0.092117 x 0.5 = -0.046059. Its sigmoid derivative is 0.710950 x 0.289050 = 0.205500. Product: -0.009465.
20. Hidden unit 2: -0.092117 x -0.3 = +0.027635. Its sigmoid derivative is 0.331812 x 0.668188 = 0.221713. Product: +0.006127.
21. Note that hidden unit 2 got a positive share while unit 1 got a negative one, because its outgoing weight was negative. The sign carried through the multiplication automatically.
22. Now the input weights. Each is its unit's delta times the input it multiplied:
23. w11 = -0.009465 x 0.5 = -0.004733. w12 = -0.009465 x -1.0 = +0.009465.
24. w21 = +0.006127 x 0.5 = +0.003064. w22 = +0.006127 x -1.0 = -0.006127.
25. Biases b1 = -0.009465 and b2 = +0.006127.
26. Every parameter now has a share. Update with a learning rate of 0.5, using new value = old value minus rate times share:

Parameter	Old	Share	New
v1	0.5	−0.065491	0.532745
v2	−0.3	−0.030566	−0.284717
c	0.2	−0.092117	0.246059
w11	0.8	−0.004733	0.802366
w12	−0.4	+0.009465	−0.404733
w21	0.2	+0.003064	0.198468
w22	0.6	−0.006127	0.603064

27. Run the forward pass again with the new numbers. Output is now 0.629935, up from 0.612048, and the loss is 0.068474, down from 0.075253.
28. It moved towards the target of 1.0. One step, and it worked.
29. Continue: after 5 steps the output is 0.688, after 20 steps 0.801, after 50 steps 0.874. Loss falls from 0.0753 to 0.0079.
30. That is training. Everything else is this, at scale.

46.12.4 PLAIN – what is really happening inside

1. Notice something about the arithmetic above. The value 0.710950 was used twice: once going forward, and once coming back.

2. That is the reason the forward pass stores its intermediate values. Every activation computed on the way in is kept, because the backward pass needs it.
3. This is why training uses far more memory than inference. Inference can discard each layer's output as soon as the next layer has consumed it. Training cannot discard anything until the backward pass has been.
4. For large models this stored activation memory frequently exceeds the memory taken by the weights themselves.
5. The other thing to notice: at no point did anyone write down a derivative formula for the whole network.
6. Each small step knew only its own local rule. The sigmoid knows its own derivative. Multiplication knows its own. Addition knows its own.
7. The overall answer came from chaining those local rules together, mechanically.
8. That mechanical chaining is called **automatic differentiation**, and it is what PyTorch, JAX and TensorFlow actually implement. Backpropagation is one specific use of it.
9. Here is how the frameworks do it. As the forward pass runs, every operation records itself and its inputs onto a graph. Multiply here, add there, ReLU there. That is the **computation graph**.
10. When you ask for gradients, the framework walks that graph backwards. At each node it applies that operation's known local derivative rule and multiplies it into what arrived from downstream.
11. The graph is built while the code runs, so an `if` statement or a loop in your model creates a different graph on different inputs, and it still works. That style is called define-by-run, and PyTorch popularized it.
12. So you never write a gradient. You write the forward computation in ordinary code, call `.backward()`, and every parameter's share appears in its `.grad` field.

46.12.5 TECHNICAL - the engineer's version

1. Notation for a network of L layers, with z the pre-activation, a the activation, and delta the error signal at a layer.
2. Output layer: delta_L = grad_a L, elementwise times f'(z_L). For the common pairing of softmax with cross-entropy, this simplifies exactly to predicted probability minus one-hot target, with no activation derivative left over.
3. Recurrence for earlier layers: delta_l = (W_{l+1}-transpose delta_{l+1}), elementwise times f'(z_l).
4. Parameter gradients: dL/dW_l = delta_l a_{l-1}-transpose, and dL/db_l = delta_l. Summed over the batch.
5. Read those four lines again. The forward pass multiplies by W. The backward pass multiplies by W-transpose. That is the whole structural symmetry, and it is why backward costs about the same as forward.
6. Cost: the backward pass is about twice the forward, because each layer computes both the gradient with respect to its input and the gradient with respect to its weights. Total training step is about three times a forward pass, giving the 2 N forward plus 4 N backward that produces the 6 N D rule.
7. Memory: activations for all L layers must be retained. Gradient checkpointing, from Chen and colleagues in 2016, stores only some layers and recomputes the rest during the backward pass, trading roughly 30 percent more compute for a large memory saving. It is standard for large models.
8. Automatic differentiation has two modes. Reverse mode, used here, costs one sweep per output and is efficient when outputs are few and inputs are many. Forward mode costs one sweep per input and is efficient in the opposite case. Training has one scalar loss and billions of parameters, so reverse mode wins by an enormous margin.
9. Reverse mode is Linnainmaa's 1970 result. Applying it to networks is Werbos in 1974 and Rumelhart, Hinton and Williams in 1986.
10. Backpropagation is not finite differences. Nudging each weight and re-running would cost one forward pass per parameter, so 175 billion forward passes for GPT-3 instead of one backward pass. It is also not symbolic differentiation, which would produce an expression that grows explosively with depth.

Automatic differentiation is a third thing: numerically exact, and linear in the size of the computation.

```
import torch
x = torch.tensor([0.5, -1.0])
W1 = torch.tensor([[0.8, -0.4], [0.2, 0.6]], requires_grad=True)
b1 = torch.tensor([0.1, -0.2], requires_grad=True)
h = torch.sigmoid(W1 @ x + b1)
v = torch.tensor([0.5, -0.3], requires_grad=True)
c = torch.tensor(0.2, requires_grad=True)
out = torch.sigmoid(v @ h + c)
loss = 0.5 * (out - 1.0) ** 2
loss.backward()
print(v.grad)     # tensor([-0.0655, -0.0306])
print(W1.grad)    # tensor([[-0.0047, 0.0095], [0.0031, -0.0061]])
```

11. Those printed gradients are exactly the shares computed by hand above. The hand calculation and the framework agree to the digits shown.
12. Practical detail: gradients accumulate by default in PyTorch, so `optimizer.zero_grad()` must be called each step. Forgetting it is one of the most common bugs in the field, and it shows up as training that starts fine and then degrades.

46.12.6 WORDS - remember these

1. Backpropagation — passing the blame backwards through the layers — reverse- mode automatic differentiation applied to a layered network.
2. Chain rule — effects along a chain multiply — the derivative of a composition is the product of the derivatives of its parts.
3. Delta / error signal — one unit's share of the blame — the partial derivative of the loss with respect to that unit's pre-activation.
4. Computation graph — the record of every operation performed — a directed acyclic graph of operations and tensors, built during the forward pass.
5. Automatic differentiation — chaining known local derivative rules mechanically — exact derivative evaluation at machine precision, linear in the cost of the original computation.
6. Gradient checkpointing — recomputing instead of storing — trading extra forward compute for reduced activation memory during backpropagation.

46.13 Training in practice

46.13.1 PLAIN - in simple words

1. The theory so far says: start somewhere, compute gradients, step downhill. In practice, five extra things decide whether it works at all.
2. **Initialization**: what the weights are before training starts. They cannot all be zero, because then every neuron in a layer computes the same thing, gets the same gradient, and stays identical forever. Symmetry must be broken by randomness.
3. But the random numbers cannot be any size. Too large and each layer amplifies its input, so after twenty layers the numbers explode. Too small and each layer shrinks its input, so after twenty layers the signal is gone.
4. The fix is to scale the random numbers by the size of the layer, so each layer roughly preserves the scale of what passes through it.
5. **Input normalization**: rescale your input data so each feature has an average near zero and a similar spread. Section 46.5 showed this making the safe learning rate thirty-five times larger on a two-parameter problem.
6. **Normalization inside the network**: do the same rescaling to the values flowing between layers, not just at the entrance.

7. **Gradient clipping**: if the gradient is enormous, shrink it before taking the step, so one bad batch cannot destroy the model.
8. **Learning rate schedules**: do not use one step size for the whole run. Start small, grow, then shrink towards the end.
9. None of these change what the model can represent. All of them change whether you can find the weights.

46.13.2 PLAIN - a picture in your head

1. Think of a chain of loudspeakers, each feeding the next one's microphone.
2. If every stage amplifies by 1.5, then after twenty stages the signal is 3,300 times louder and everything clips into noise.
3. If every stage attenuates by 0.7, after twenty stages the signal is one thousandth of its original size and is lost in the hiss.
4. What you want is every stage having a gain of almost exactly 1, so the signal arrives at the end intact.
5. Careful initialization is setting each amplifier's gain to about 1 at the start. Normalization layers are automatic gain control, resetting the level at each stage while the system runs.

Where this comparison breaks: an audio chain carries one signal and the right gain is obviously 1. A network carries hundreds of dimensions, and the useful quantity is the variance across them, not the level of one. Also, the gradient flows in the opposite direction to the signal, and it has its own gain that can explode or vanish independently. Initialization schemes have to compromise between keeping the forward pass and the backward pass both well scaled, and in general you cannot have both exactly.

46.13.3 PLAIN - a worked example

1. Xavier initialization, from Glorot and Bengio in 2010: draw weights from a distribution with standard deviation equal to the square root of 2 divided by (inputs plus outputs).
2. He initialization, from He and colleagues in 2015: standard deviation equal to the square root of 2 divided by inputs. It is larger, to compensate for ReLU throwing away half the signal.

Layer shape	Xavier std	He std
784 -> 128	0.0468	0.0505
128 -> 64	0.1021	0.1250
512 -> 512	0.0442	0.0625

3. Rule: Xavier with tanh or sigmoid, He with ReLU and its relatives. Both ship in every framework and are usually the default already.
4. Gradient clipping by norm, worked. Suppose three parameters have gradients 3.0, 4.0 and 12.0.
5. The norm is the square root of $(9 + 16 + 144) =$ square root of $169 = 13.0$.
6. Clip threshold is 1.0. Since 13.0 exceeds 1.0, multiply every component by $1.0 / 13.0 = 0.07692$.
7. New gradients: 0.2308, 0.3077, 0.9231. Their norm is exactly 1.0.
8. The direction is unchanged. Only the length was cut. That is the point: you still step the right way, just not off a cliff.
9. A warmup-plus-cosine schedule, with peak rate 3e-4, 2,000 warmup steps and 100,000 total steps:

Step	Learning rate
0	0
1,000	1.5e-4
2,000	3.0e-4 (peak)
10,000	2.95e-4

Step	Learning rate
50,000	1.55e-4
100,000	0

10. Warmup exists because at step zero the weights are random, the gradients are large and meaningless, and a full-size step would throw the model somewhere useless before it has learned anything. Starting from zero and ramping up avoids that.

46.13.4 PLAIN - what is really happening inside

1. The two normalization layers, and exactly what each normalizes. People confuse these constantly.
2. **Batch normalization**, from Ioffe and Szegedy in 2015: for each feature, compute the average and spread across all the samples in the batch, then rescale that feature so it has mean 0 and variance 1.
3. It normalizes down the batch. Sample 5's value for feature 3 depends on what samples 1 to 64 happened to be.
4. That has a strange consequence: a sample's output depends on which other samples it travelled with. At inference you have no batch, so batch norm keeps a running average from training and uses that instead. This train versus inference difference is a genuine source of bugs.
5. **Layer normalization**, from Ba, Kiros and Hinton in 2016: for each sample, compute the average and spread across that sample's own features, then rescale.
6. It normalizes across the features. Each sample is handled entirely alone, so the batch size does not matter and training and inference behave identically.
7. That independence is why layer normalization won in transformers and sequence models, where sequences have different lengths and batch statistics are unreliable. Batch normalization remains standard in convolutional image networks.
8. One sentence to remember: batch norm normalizes each feature over the batch; layer norm normalizes each sample over its features.
9. Both add two learned parameters per feature, a scale and a shift, so the network can undo the normalization where it is unhelpful.

46.13.5 TECHNICAL - the engineer's version

1. Troubleshooting, which is the most useful table in this chapter:

Symptom	Likely cause	Action
Loss is NaN	learning rate too high	halve it, add clipping
Loss flat from step 0	rate too low, or dead net	raise rate 10x, check init
Loss falls then explodes	rate too high late	add warmup and decay
Train falls, val rises	overfitting	augment, regularize, stop
Both flat and high	underfitting	bigger model, more steps
Loss spikes then recovers	bad batch or outlier	clip gradients, shuffle
Val loss very noisy	val set too small	enlarge it, use EMA
Works on 1 batch, not all	data pipeline bug	overfit 1 batch first
Slower each epoch	memory leak	check retained graphs

2. The single best debugging habit: take one batch of eight samples and train until the loss reaches near zero. If a model cannot memorize eight examples, the bug is in your code, not your hyperparameters. This takes two minutes and saves days.
3. Schedules in common use: step decay, which multiplies by 0.1 at fixed epochs, the classic ImageNet recipe; cosine annealing, from Loshchilov and Hutter in 2016, now the default for transformers; and inverse square root, used in the original 2017 transformer paper with 4,000 warmup steps.

4. Typical warmup lengths for large language models are 1 to 2 percent of total steps, or a fixed 2,000 steps. This is a convention, not a standard.
5. Gradient clipping: clip by global norm, not per-parameter, so the update direction is preserved. A threshold of 1.0 is the near-universal default for transformer training.
6. Mixed precision is standard practice: keep a master copy of weights in FP32, compute in BF16 or FP16, and with FP16 scale the loss up before the backward pass to stop small gradients underflowing to zero. BF16 has the same exponent range as FP32 so it usually needs no loss scaling, which is why it is preferred on hardware that supports it.
7. Observation tools: TensorBoard or Weights and Biases for curves, `torch.cuda.max_memory_allocated()` for memory, `nvidia-smi` for utilization, and `torch.profiler` or Nsight Systems for finding the real bottleneck. Low GPU utilization almost always means the data loader, not the model.
8. Reproducibility caveat: seeding with `torch.manual_seed` is not sufficient for bit-identical runs, because some GPU kernels are non-deterministic by design. `torch.use_deterministic_algorithms(True)` forces determinism where possible, at some cost in speed.

46.13.6 WORDS - remember these

1. Initialization — the random starting values of the weights — a distribution chosen to keep activation and gradient variance stable across depth.
2. Xavier / Glorot initialization — scaling by inputs plus outputs — variance $2/(\text{fan_in} + \text{fan_out})$, suited to tanh and sigmoid.
3. He initialization — scaling by inputs only, larger — variance $2/\text{fan_in}$, compensating for ReLU zeroing half its input.
4. Batch normalization — rescaling each feature across the batch — normalizing over the batch dimension, with running statistics used at inference.
5. Layer normalization — rescaling each sample across its own features — normalizing over the feature dimension, identical in training and inference.
6. Gradient clipping — cutting the step length without changing direction — rescaling the gradient vector when its global norm exceeds a threshold.
7. Warmup — starting with tiny steps and ramping up — a linear increase of the learning rate over the first steps, before the main decay schedule.

46.14 The families of neural network

46.14.1 PLAIN - in simple words

1. Everything so far described the **feedforward** network: layers in a line, every unit connected to every unit in the next layer. It is the default and the fallback.
2. Its weakness is that it treats every input as unrelated to every other. Move a picture one pixel to the right and it is a completely different input.
3. A **convolutional network** fixes this for images. Instead of one weight per pixel, it learns a small patch of weights and slides that same patch over the whole image.
4. So a pattern learned in one corner is recognized everywhere, and the number of weights collapses.
5. A **recurrent network** handles sequences, such as words or sound. It reads one item at a time and keeps a running summary of what it has read.
6. An **autoencoder** learns to squeeze data into a small representation and rebuild it. Useful for compression, denoising and finding structure.
7. A **generative adversarial network** trains two networks against each other: one makes fakes, one detects fakes, and both improve.
8. A **diffusion model** learns to remove noise, and generates by starting from pure noise and removing it repeatedly until an image appears.

9. A **graph neural network** works on data with connections, such as molecules or social networks, by passing messages along the edges.
10. The **transformer** replaced recurrent networks for almost all sequence work after 2017. Chapter 48 covers it properly.

46.14.2 PLAIN - a picture in your head

1. Think of a rubber stamp and an ink pad.
2. A convolutional filter is a stamp with a small pattern cut into it. You press it at every position on the page and record how well the pattern matched there.
3. One stamp finds one thing: a vertical edge, a spot of red, a diagonal line.
4. A layer has many stamps, so you get many maps of where each pattern occurred.
5. The next layer stamps patterns onto those maps, so it finds combinations of patterns. A corner is where a vertical edge map and a horizontal edge map both light up.
6. Layer after layer, the patterns become larger and more specific.

Where this comparison breaks: a stamp is a fixed pattern you carved. These are learned from data, and typically nobody can say in words what most of them detect. The early ones do look like edge detectors and colour blobs, reliably and across independently trained networks, which is a genuine and repeatable finding. The middle ones are much harder to describe, and the neat story that each unit detects one nameable thing is often false.

46.14.3 PLAIN - a worked example

1. One convolution, worked completely. A 5 by 5 image with a vertical edge in it: two dark columns then three bright ones.

```
image                     kernel (vertical edge)
  0   0  10  10  10          -1   0  +1
  0   0  10  10  10          -2   0  +2
  0   0  10  10  10          -1   0  +1
  0   0  10  10  10
  0   0  10  10  10
```

2. The kernel is 3 by 3, so it fits in the image at 3 x 3 = 9 positions.
3. Take the top-left position. The window covers rows 1 to 3, columns 1 to 3: all values are 0, 0, 10 on each row.
4. Multiply each window value by the kernel value in the same place and add:
5. Row 1: (0 x -1) + (0 x 0) + (10 x +1) = 10.
6. Row 2: (0 x -2) + (0 x 0) + (10 x +2) = 20.
7. Row 3: (0 x -1) + (0 x 0) + (10 x +1) = 10.
8. Total: 10 + 20 + 10 = 40. That is one output value.
9. Slide right by one and repeat. Do this for all nine positions:

```
output (3x3)
  40  40   0
  40  40   0
  40  40   0
```

10. Read it. The left columns are strongly positive, marking the edge. The right column is exactly 0, because that window was uniformly bright, and a uniform region produces no edge response.
11. Nine parameters found a vertical edge anywhere in the image. A fully connected layer would have needed 25 weights per output unit and would have had to learn the pattern separately at every position.
12. Now **pooling**, which shrinks the map. Take a 4 by 4 grid and keep the largest value in each 2 by 2 block:

```
  1  3 | 2  4          max-pool 2x2, stride 2
  5  6 | 1  2   ---->        6   4
```

```
------+------              7   9
7  2 | 8  0
1  4 | 3  9
```

13. The result is half the size in each direction, so a quarter of the values. It keeps the strongest response and discards where exactly it occurred, which makes the network tolerant of small shifts.

46.14.4 PLAIN - what is really happening inside

1. Recurrent networks and their fundamental limit, which is worth understanding properly because it explains why transformers exist.
2. A recurrent network reads word one, updates a hidden summary, reads word two, updates it again, and so on. The same weights are used at every step.
3. So the summary after 50 words has been through 50 multiplications by the same matrix.
4. If that matrix shrinks things slightly, the influence of word one is multiplied by a number below 1 fifty times and vanishes. If it grows things slightly, the influence explodes.
5. That is the vanishing and exploding gradient problem again, this time through time rather than through layers, and it is why plain recurrent networks cannot remember much past about ten steps.
6. The **LSTM**, from Hochreiter and Schmidhuber in 1997, fixes it with a separate memory line that information can travel along unchanged, plus three learned gates that decide what to write, what to keep and what to read. That memory path is additive rather than multiplicative, so gradients survive along it. LSTMs handle hundreds of steps.
7. But there is a second limit that no gate fixes: a recurrent network must process step 1 before step 2. It is sequential by construction.
8. That means it cannot use the parallel hardware from Chapter 22 across the sequence. Doubling your chips does not halve your training time.
9. The transformer's central trick is to let every position look at every other position at once, in one matrix multiplication, with no sequential dependency. That is why it took over. Chapter 48 explains how.
10. Established fact: transformers replaced recurrent networks for nearly all language work from about 2018 onwards. Active research: state-space models such as Mamba, from 2023, revisit the recurrent idea with a formulation that can be parallelized during training. Whether they displace attention at scale is genuinely open as of 2026.

46.14.5 TECHNICAL - the engineer's version

1. Summary of the families:

Family	Best for	Key idea
Feedforward MLP	tabular, final heads	dense layers
Convolutional	images, spectrograms	shared local filters
Recurrent / LSTM	short sequences, streams	carried hidden state
Transformer	text, and now most things	all-pairs attention
Autoencoder	compression, anomalies	bottleneck plus rebuild
GAN	fast image synthesis	generator versus critic
Diffusion	high-quality generation	learned denoising
Graph network	molecules, networks	message passing on edges

2. Convolution parameter count: a layer with C_in input channels, C_out output channels and a k by k kernel has C_in x C_out x k x k weights plus C_out biases, independent of image size. A 3x3 layer with 64 in and 128 out has 73,728 weights and is applied at every spatial position.
3. Convolution FLOPs: 2 x H_out x W_out x C_in x C_out x k x k. This is why ResNet-50 costs about 8.2 billion operations per 224 by 224 image with only 25.6 million parameters. Weights are reused, not multiplied.

4. Convolutional milestones: Fukushima's Neocognitron in 1980, LeCun's backpropagation-trained digit network at Bell Labs in 1989, LeNet-5 in 1998 reading cheques in production, AlexNet in 2012, VGG and GoogLeNet in 2014, ResNet in 2015.
5. What is actually implemented: most convolutional layers are lowered to a matrix multiply, either by the im2col transformation or by an implicit GEMM in cuDNN, so the underlying arithmetic is the same GEMM from Chapter 22. Winograd algorithms are used for small kernels to cut the multiply count.
6. Autoencoders: an encoder maps to a bottleneck of lower dimension, a decoder rebuilds, and the loss is reconstruction error. The variational autoencoder, from Kingma and Welling in 2013, adds a probabilistic latent distribution and enables generation. Autoencoder latents are used inside modern image diffusion models to work in a compressed space rather than pixel space.
7. GANs, from Ian Goodfellow and colleagues in 2014, train a generator and a discriminator in a minimax game. They produce images in a single forward pass, which is fast, but training is notoriously unstable and can collapse to producing a few outputs repeatedly. Diffusion models displaced them for quality from about 2021.
8. Diffusion models come from Sohl-Dickstein and colleagues in 2015, made practical by Ho, Jain and Abbeel's denoising diffusion probabilistic models in 2020. A forward process adds Gaussian noise in steps; a network learns to predict and remove it; generation runs the process backwards from noise. Sampling originally took hundreds of steps; distilled and flow-matching variants now reach single-digit steps. This area is moving fast.
9. Graph neural networks aggregate each node's neighbours' features, transform the result, and repeat. Message passing was formalized by Gilmer and colleagues in 2017. They are used in molecular property prediction, and AlphaFold 2 in 2021 combined graph-like reasoning with attention.

46.14.6 WORDS - remember these

1. Convolution — sliding one small pattern over the whole image — a shared, local, translation-equivariant linear operator, implemented as a GEMM.
2. Kernel / filter — the small patch of learned weights — the k by k by C_in tensor slid across the input.
3. Pooling — shrinking the map by keeping the strongest response — a downsampling operation, usually 2x2 max or average with stride 2.
4. Feature hierarchy — edges first, objects later — successive layers representing progressively larger and more abstract patterns.
5. Recurrent network — reading a sequence while keeping a running summary — a network applying shared weights over time steps with a carried hidden state.
6. LSTM — a recurrent unit with a protected memory line — long short-term memory, with input, forget and output gates over an additive cell state.
7. Autoencoder — squeeze it small, then rebuild it — an encoder-decoder trained on reconstruction loss through a bottleneck.
8. Diffusion model — generating by repeatedly removing noise — a model trained to reverse a gradual noising process.

46.15 What a trained model physically is

46.15.1 PLAIN - in simple words

1. After all the training, what do you actually have? A file.
2. The file contains numbers. Nothing else. No code, no logic, no rules you could read.
3. It is a long list of the weights and biases, in a known order, with a small index at the front saying which block of numbers belongs to which layer.
4. To use the model you need two things: this file, and the code that knows the architecture, so it can put each block of numbers in the right place.
5. Neither is useful alone. The file without the architecture is noise. The architecture without the file is an untrained network that outputs rubbish.
6. **Training** is the expensive process of producing the file. **Inference** is the cheap process of using it to answer one question.
7. Training happens once, on a cluster, for weeks. Inference happens billions of times, on one machine, in milliseconds.
8. The asymmetry between those two is the single most important economic fact about this field. Chapter 49 goes into what is inside the file in detail.

46.15.2 PLAIN - a picture in your head

1. Think of a printed dictionary.
2. Compiling it took a team of lexicographers many years, with arguments, revisions and enormous cost.
3. Looking up one word takes you four seconds and costs nothing.
4. The book is the model file. The compilation is training. The lookup is inference.
5. And note: you cannot recover the arguments, the drafts or the sources from the finished book. It contains the conclusions and nothing else.

Where this comparison breaks: a dictionary can be read by a person and checked entry by entry. A model file cannot. There is no entry for anything. The knowledge, if that word even applies, is spread across all the numbers at once, and no single number means anything on its own. That property is called distributed representation, and it is why interpreting these models is hard research rather than an afternoon of reading.

46.15.3 PLAIN - a worked example

1. File sizes, computed directly. Multiply the parameter count by the bytes per number.

Model	Parameters	Precision	File size
MNIST MLP	109,386	FP32	0.44 MB
ResNet-50	25.6 million	FP32	102 MB
Llama-class 8B	8 billion	BF16	16 GB
Llama-class 70B	70 billion	BF16	140 GB
Same 70B, 4-bit	70 billion	INT4	35 GB

2. The arithmetic is simply: FP32 is 4 bytes per number, BF16 and FP16 are 2, INT8 is 1, INT4 is half a byte.
3. That last row is why quantization matters. The same model drops from 140 GB to 35 GB, which is the difference between needing several datacentre cards and fitting on one.
4. Now the compute asymmetry, with the standard estimates. Training cost is about 6 times parameters times training tokens.
5. For an 8-billion-parameter model trained on 15 trillion tokens: 6 x 8e9 x 15e12 = 7.2e23 floating-point operations.

6. Inference cost for one token is about 2 times parameters: 2 x 8e9 = 1.6e10 operations.
7. The ratio is 4.5e13. Training that model costs about as much arithmetic as generating 45 trillion tokens with it.
8. So if the model serves a billion tokens a day, the training cost is repaid in arithmetic terms after about 45,000 days of service.
9. That is why nobody trains their own foundation model unless they are selling access to it, and why fine-tuning an existing file is the normal path.

46.15.4 PLAIN - what is really happening inside

1. Inside the file, the layout is boring by design.
2. A header holds the metadata: the name of each tensor, its shape, its data type, and the byte range where its numbers start and end.
3. Then a single long region of raw numbers, laid out so a program can memory- map the file and read a tensor without copying it.
4. The `safetensors` format, now the default for published models, is exactly that: a JSON header giving names, shapes and offsets, then raw bytes.
5. It replaced Python's `pickle` format for a security reason. A pickle file can execute arbitrary code when loaded, so downloading a model was equivalent to running a stranger's program. `safetensors` cannot execute anything.
6. What is not in the file: the training data, the optimizer state, the code, or any record of how the numbers were reached.
7. The optimizer state is worth noting. During training, Adam keeps two extra numbers per parameter, so the training checkpoint is roughly three times the size of the final model, or four if a master FP32 copy is kept. Those extra numbers are discarded before publication.
8. Inference needs the weights and one small activation buffer. Training needs the weights, the gradients, the optimizer state, and every stored activation. That is why a model you can run on one card may need eight cards to fine-tune.

46.15.5 TECHNICAL - the engineer's version

1. Memory required, per billion parameters, as a planning table:

Purpose	Precision	Memory per 1B
Inference weights	INT4	0.5 GB
Inference weights	BF16	2 GB
Training weights	FP32 master	4 GB
Adam state	FP32, 2 moments	8 GB
Gradients	BF16	2 GB

2. So full fine-tuning at 1 billion parameters needs roughly 16 GB before activations, against 2 GB to run it. A factor of eight, and this is exactly why parameter-efficient methods such as LoRA, from Hu and colleagues in 2021, exist.
3. Formats you will meet: `safetensors` for published weights, ONNX for framework-portable graphs, GGUF for quantized local inference in the llama.cpp ecosystem, and TensorRT engines for compiled NVIDIA deployment.
4. Established fact: a trained model is a fixed function. Given identical inputs, identical settings and deterministic kernels, it returns identical outputs. Chat models appear to vary because a random sampler is applied to their output distribution, not because the model changed.
5. Established fact: the model does not learn from your conversation. Its weights are read-only at inference. Anything that looks like learning within a session is context, not weight change.

6. Marketing claim: that a deployed model "improves as you use it". In almost all cases the file is static and improvements arrive as a new file, trained separately, on data that may include logged interactions. That is a different and much slower loop, and it is worth insisting on the distinction.

46.15.6 WORDS - remember these

1. Model file / checkpoint — the file of trained numbers — a serialized tensor dictionary mapping parameter names to arrays.
2. Inference — using the trained model to answer — a forward pass only, with no gradients, activations discarded as soon as they are consumed.
3. Training — producing the numbers in the first place — repeated forward passes, backward passes and optimizer updates over a dataset.
4. Quantization — storing the numbers in fewer bits — reducing weight precision to INT8 or INT4 to cut memory and bandwidth, at some accuracy cost.
5. Safetensors — the safe modern weight file format — a JSON header of names, shapes and offsets over a raw tensor region, with no code execution.
6. Optimizer state — the extra numbers training needs — momentum and variance estimates, roughly two extra values per parameter for Adam.

46.98 Common wrong ideas

1. Wrong: the network understands what it is doing. Right: it computes a function. Weighted sums and bends, repeated. "Understanding" has no agreed technical definition here, and using the word imports assumptions nobody has earned. Whether internal structures deserve richer descriptions is active research, not a settled fact.
2. Wrong: more layers is always better. Right: past a point, deeper networks get harder to optimize, overfit more and cost more, without gaining accuracy. Plain networks beyond about 20 layers did not train at all until residual connections in 2015, and even now the right depth is chosen by measurement, not by adding more.
3. Wrong: it learned by itself. Right: a person chose the architecture, the loss, the optimizer, the learning rate, the schedule, the regularization, the data and when to stop. The only thing the machine did was solve, by repeated small steps, the exact optimization problem it was handed.
4. Wrong: neural networks work like brains. Right: the resemblance stopped at the 1943 model of a neuron. Real neurons spike, have timing, chemistry and internal state, and change their own structure. An artificial unit is a weighted sum and a bend. The name is historical, and today it is mostly marketing.
5. Wrong: AI is just a big pile of if-statements. Right: there are no branches at all in a dense network's forward pass. It is a fixed sequence of multiplications, additions and one elementwise function, executed in the same order for every input. That is the opposite of branching.
6. Wrong: training finds the best possible weights. Right: it finds some set of weights where the gradient is near zero, reached from where it started. A different random start gives different weights and often similar accuracy. There is no guarantee of optimality for any non-convex model.
7. Wrong: a lower training loss means a better model. Right: training loss can always be driven down by memorizing. The only number that means anything is performance on data the model has never seen and that you never used to make a decision.
8. Wrong: the model stores its training data, so it can be asked to recall it. Right: the data shaped the parameters and was then discarded. Verbatim recall of some training examples does happen, and is a real privacy concern, but it is a side effect of over-fitting particular sequences, not a lookup mechanism.
9. Wrong: bigger datasets always help. Right: more of the same distribution has diminishing returns, and more badly labelled data actively hurts. Data quality, coverage and correct labelling usually beat raw volume.
10. Wrong: gradient descent gets stuck in local minima all the time. Right: in very high dimensions the common stationary points are saddles, not minima, because a minimum requires upward curvature in

every direction at once. The practical obstacles are ill-conditioning, poor scaling and bad learning rates far more often than local minima.

46.99 Chapter summary in 20 lines

1. Machine learning writes a rule-shaped thing with adjustable numbers, and a procedure that tunes those numbers until outputs match examples.
2. It is curve fitting at heart, and saying so is accurate rather than dismissive; it is what lets you reason about the system.
3. The four pieces are always the same: a model, its parameters, a loss, and an optimizer that reduces the loss.
4. The history has two public failures. The perceptron of 1957 to 1958 was overclaimed, limited by Minsky and Papert in 1969, and funding collapsed.
5. Backpropagation existed from Linnainmaa in 1970 and Werbos in 1974 and was ignored until Rumelhart, Hinton and Williams published it in Nature in 1986.
6. In 2012 AlexNet won ImageNet with 15.3 percent top-5 error against 26.2 percent, trained on two consumer graphics cards, and the argument ended.
7. A parameter is a number the system may change; a hyperparameter is one you choose. Fitting $y = wx + b$ to five points has exactly two parameters.
8. Loss is one number saying how wrong you are. Squared error punishes large misses hardest; absolute error resists outliers. The choice changes the fit.
9. The loss surface is a landscape over the parameters, and training is walking downhill on it in fog, feeling only the slope under your feet.
10. Gradient descent steps against the gradient by a learning rate. Too small crawls; too large diverges to NaN. Normalizing inputs reshapes the ravine.
11. Momentum, RMSProp and Adam all exist to fix the same problem: plain descent zig-zags across narrow valleys instead of running along them.
12. Split your data into training, validation and test sets, and decide nothing using the test set. A widening gap between training and validation loss is overfitting, and the cure is early stopping, regularization or more data.
13. Classical methods are not obsolete. Gradient-boosted trees remain a leading choice for table-shaped data in 2026, though tabular foundation models are now competitive on small datasets and the boundary is moving.
14. One neuron is a weighted sum plus a bias, then a non-linear bend. It is not a brain cell and the analogy is historical.
15. Without the bend, stacked layers collapse algebraically into a single layer, so depth would buy exactly nothing. ReLU won because its slope is 1.
16. A forward pass is a matrix multiply, a bias add and a bend, once per layer. It is the same matrix multiply as Chapter 22, which is why GPUs run this field.
17. Backpropagation assigns each weight its share of the error by passing blame backwards layer by layer, then nudges every weight against its share.
18. That is the chain rule applied mechanically, which frameworks implement as reverse-mode automatic differentiation over a recorded computation graph.
19. Convolutional networks share small filters across an image, recurrent networks carry a running summary and cannot be parallelized over time, and the transformer of 2017 replaced them for sequences. See Chapter 48.
20. A trained model is a file of numbers, useless without the architecture that reads it. Training costs roughly $6 N D$ operations once; inference costs about $2 N$ per token, forever. See Chapter 49.

47

CHAPTER

KB

Parameters, Tokens and Embeddings

47.0 What this chapter gives you

1. You will be able to say exactly what one parameter is: a single number in a single position of a single table inside the model.
2. You will be able to take an architecture description and compute the parameter count by hand, and say which parts of the model hold the billions.
3. You will be able to tell parameters apart from hyperparameters, activations, context and tokens, and use all five words correctly in one sentence.
4. You will be able to turn a parameter count into a memory figure, for inference and for training, and explain the gap between them.
5. You will be able to explain quantization, name the common formats, and work out whether a given model fits on a given machine.
6. You will be able to say honestly why parameter count is a weak predictor of how useful a model is, and cite the evidence for that.
7. You will be able to explain byte-pair encoding by working it through by hand on a tiny corpus, and name the three other tokenizer families.
8. You will be able to predict roughly how many tokens a piece of text costs, and explain why Hindi costs several times more than English for one meaning.
9. You will be able to explain why a model cannot count the letters in a word, and why it fails at arithmetic on long numbers, from tokenization alone.
10. You will be able to explain what an embedding is, what similarity between embeddings means, and what a context window really limits.

47.1 What a parameter actually IS in a language model

47.1.1 PLAIN – in simple words

1. A language model is, on the inside, a very large pile of numbers.
2. Nothing else. No sentences stored. No facts written down as text. Numbers.
3. One of those numbers is called a **parameter**.
4. Chapter 46 fitted a straight line to five points. That line had exactly two parameters: a slope and an intercept, both found by gradient descent, which is the process of nudging numbers downhill until the error stops falling.
5. A language model is the same idea with the count raised enormously. Two parameters becomes eight billion, and the shape of the thing being fitted becomes a network of layers instead of a line.
6. Everything else in this section is that one sentence unpacked.
7. So a parameter is not a setting you choose. It is not an option in a menu.
8. It is one plain number, like -0.0113 or 0.0392, sitting in one fixed slot.
9. The slots are arranged in rectangles, called matrices. Rows and columns.

10. A parameter has an address: this matrix, this row, this column.
11. When the model was trained, every one of those numbers was slowly adjusted until the model got good at predicting the next piece of text.
12. When training finished, the numbers were frozen and written to a file.
13. Every time you use the model, it reads the same frozen numbers.
14. Your question does not change them. Your conversation does not change them.
15. When somebody says a model has 7 billion parameters, they mean the file contains 7 billion such numbers.
16. That is the whole idea. A parameter is one learned number in one slot.

47.1.2 PLAIN - a picture in your head

1. Picture an enormous sound mixing desk, the kind in a recording studio.
2. A mixing desk has knobs. Each knob controls how loudly one thing is heard.
3. Turn one knob and the guitar gets louder. Turn another and the drums fade.
4. Now imagine a desk with seven billion knobs instead of forty.
5. Each knob is set to some exact position, like 0.0392 out of a range.
6. Training was a very long process of nudging every knob a tiny amount, over and over, until the whole desk produced a good sound.
7. Shipping the model means writing down every knob position in a file.
8. Loading the model means setting all seven billion knobs back to those exact positions.
9. Running the model means pushing sound through the desk without touching a single knob.

Where this comparison breaks: on a real mixing desk, a human knows what each knob does, and there is a label under it. In a model, almost no single parameter has a meaning you can name. Its effect only exists in combination with millions of others. Also, a mixing desk has knobs a person turns during a performance. Model parameters are never turned during use. They are turned only during training, and then never again unless the model is retrained.

47.1.3 PLAIN - a worked example

1. Here is a genuine kind of object from inside a model: a weight matrix.
2. This is a four-by-four corner of one, with values rounded for printing.

```
A 4 x 4 corner of one weight matrix, values rounded

          col 0     col 1     col 2     col 3
row 0   -0.0113    0.0271   -0.0004    0.0158
row 1    0.0392   -0.0087    0.0221   -0.0310
row 2   -0.0056    0.0143    0.0079    0.0002
row 3    0.0018   -0.0225   -0.0166    0.0407
```

3. Look at row 1, column 2. The value there is 0.0221.
4. That is a parameter. That one number. Not the row. Not the matrix. That.
5. Its full address inside a real model would be something like: layer 7, the query projection matrix, row 1, column 2.
6. There are 16 parameters in this little corner, because it is 4 rows times 4 columns.
7. The real matrix this corner came from would be 4096 by 4096.
8. That is 4096 x 4096 = 16,777,216 parameters in that one matrix alone.
9. A model of this size has hundreds of matrices like it.
10. Every one of those numbers started as a random value before training, and ended up at its final value because training pushed it there.
11. Nobody chose 0.0221. Nobody can explain 0.0221 on its own.

47.1.4 PLAIN - what is really happening inside

1. The model's job at each step is to turn a list of numbers into another list of numbers.
2. The main operation it uses is multiply-and-add, done many times at once.
3. Take an incoming list of 4096 numbers. Call it the input vector.
4. Take one column of the weight matrix, also 4096 numbers.
5. Multiply them pairwise and add the results up. That gives one output number.
6. Repeat for every column. That gives the whole output vector.
7. So each parameter is a multiplier. It says how much one incoming number contributes to one outgoing number.
8. The parameter at row 1, column 2 says: take input number 1, multiply it by 0.0221, and add that into output number 2.
9. That is literally all a weight does. It is a multiplier in one cell of a giant multiply-and-add.
10. There is a second, smaller kind of parameter called a **bias**.
11. A bias is added rather than multiplied. It shifts an output up or down by a fixed amount regardless of the input.
12. Biases are also learned during training, and are also parameters.
13. Some model families use biases everywhere, some use almost none.
14. When the model runs, it walks through layer after layer doing this. The parameters never move. Only the numbers flowing through them change.

47.1.5 TECHNICAL - the engineer's version

1. A **parameter** is one scalar element of a learnable tensor, updated by the optimizer during training and fixed at inference time.
2. In PyTorch terms, a parameter is one element of a tensor wrapped in `torch.nn.Parameter`, which is a tensor with `requires_grad=True` that is registered in the module's parameter list.
3. The words **weight** and **parameter** are used interchangeably in practice, and in almost every sentence you will read they mean the same thing.
4. The precise distinction is that a **weight** is a multiplicative parameter in a linear map, while a **bias** is an additive parameter. Both are parameters. So all weights are parameters, but not all parameters are weights.
5. Normalization scale and shift terms, often called gamma and beta, are also parameters. So are learned position embeddings where a model uses them.
6. In a decoder-only transformer of the Llama family, the linear layers carry no bias terms at all, and RMSNorm carries a scale but no shift. In GPT-2 the linear layers do carry biases and LayerNorm carries both scale and shift.
7. That difference is tiny in count. In GPT-2 small, all biases and all normalization parameters together are 121,344 out of 124,439,808 parameters, which is 0.098 per cent of the model.
8. Parameters are stored in the model's state dictionary under dotted names. Real names from a Llama-family checkpoint look like this:

```
model.embed_tokens.weight            [128256, 4096]
model.layers.0.self_attn.q_proj.weight   [4096, 4096]
model.layers.0.self_attn.k_proj.weight   [1024, 4096]
model.layers.0.mlp.gate_proj.weight     [14336, 4096]
model.layers.0.input_layernorm.weight       [4096]
lm_head.weight                       [128256, 4096]
```

9. The bracketed pair is the tensor shape. The number of parameters in a tensor is the product of its dimensions.
10. Counting them in PyTorch is one line: `sum(p.numel() for p in model.parameters())`. Counting trainable ones only adds `if p.requires_grad`.

11. For a file on disk, `safetensors` stores a JSON header listing every tensor name, dtype and shape, so the count can be read without loading the model. For GGUF files, `gguf-dump` prints the same information.
12. Terminology note, and it is a standard one: in statistics a "parameter" is a quantity of a distribution, and in software a "parameter" is a function argument. Neither meaning applies here. The machine learning meaning is the one in this chapter.

Kind of parameter	Operation	Typical share
Linear layer weight	Multiply	Over 99 per cent
Bias	Add	Under 0.1 per cent
Norm scale (gamma)	Multiply	Under 0.01 per cent
Embedding row	Lookup	0.5 to 25 per cent

47.1.6 WORDS - remember these

1. Parameter — one learned number in the model — one scalar element of a learnable tensor, updated by the optimizer, frozen at inference.
2. Weight — a number that multiplies something — a multiplicative parameter in a linear map, stored as an element of a weight matrix.
3. Bias — a number that is added on — an additive parameter applied after a linear map, one per output dimension.
4. Matrix — a rectangle of numbers — a rank-2 tensor with shape [out_features, in_features] for a linear layer.
5. Tensor — a box of numbers with a shape — an n-dimensional array with a dtype and a shape tuple.
6. State dictionary — the list of all the model's numbers with names — an ordered mapping from parameter name to tensor, saved to disk.
7. Checkpoint — the saved file of a model — a serialized state dictionary, in formats such as safetensors, PyTorch .pt or GGUF.
8. Frozen — not changing any more — parameters with `requires_grad=False`, or any parameter during inference.

47.2 What 7B, 70B and 405B actually count

47.2.1 PLAIN - in simple words

1. The B in 7B means billion. It is a count of parameters.
2. A 7B model contains about seven thousand million individual numbers.
3. A 405B model contains about four hundred and five thousand million of them.
4. Nothing else is being counted. Not files. Not facts. Not words known.
5. To get the total you add up the size of every matrix in the model.
6. Every matrix size is just rows times columns.
7. So the whole count is a long sum of simple multiplications.
8. The numbers are not round because the shapes are not round.
9. A model called "8B" often turns out to hold 8,030,261,248 parameters exactly.
10. The advertised name is rounded. The real count is exact and computable.
11. Once you know four or five shape numbers about a model, you can compute the total yourself in a few minutes.
12. That is what this section does, all the way through, for a real model.

47.2.2 PLAIN - a picture in your head

1. Imagine a large office block with 126 identical floors.
2. Every floor has exactly the same furniture plan: seven big filing cabinets.
3. Some cabinets are wide, some are narrow, but every floor gets the same set.
4. To count the drawers in the whole building you do not walk every floor.
5. You count the drawers on one floor, then multiply by 126.
6. Then you add the reception desk at the bottom and the roof office at the top, because those exist only once.
7. That is exactly how a transformer parameter count works.
8. The floors are the layers. The identical furniture is the same set of matrices repeated in every layer.
9. The reception desk is the embedding table. The roof office is the output layer. Both exist once, not once per floor.

Where this comparison breaks: real buildings sometimes have odd floors. In most transformer models every layer genuinely is the same shape, which is why the multiply-by-number-of-layers trick works exactly. But some newer designs break this on purpose: mixture-of-experts models have layers with many parallel expert blocks, and only some are used per token, so the total count and the count actually used per token stop being the same number.

47.2.3 PLAIN - a worked example

1. Let us count a real model completely: Llama 3.1 8B, released by Meta on 23 July 2024.
2. Five shape numbers describe it, and they are published:

Setting	Value
Layers	32
Hidden size (d)	4096
Feed-forward size	14336
Attention heads	32
Key-value heads	8
Vocabulary size	128256

3. Head size is hidden size divided by heads: 4096 / 32 = 128.
4. Now count one layer. Attention first, four matrices:

```
q_proj   4096 x 4096  = 16,777,216
k_proj   4096 x 1024  =  4,194,304
v_proj   4096 x 1024  =  4,194,304
o_proj   4096 x 4096  = 16,777,216
attention total per layer = 41,943,040
```

5. The k and v matrices are narrower because of grouped-query attention: only 8 key-value heads, so 8 x 128 = 1024 columns instead of 4096.
6. Now the feed-forward part, three matrices:

```
gate_proj   4096 x 14336 = 58,720,256
up_proj     4096 x 14336 = 58,720,256
down_proj  14336 x 4096  = 58,720,256
feed-forward total per layer = 176,160,768
```

7. Two normalization vectors of 4096 each add 8,192 more.
8. One layer therefore holds 41,943,040 + 176,160,768 + 8,192 = 218,112,000.
9. There are 32 layers: 32 x 218,112,000 = 6,979,584,000.

10. The embedding table is vocabulary times hidden size: 128,256 x 4,096 = 525,336,576.
11. The output layer is the same shape again: 525,336,576.
12. The final normalization adds 4,096.
13. Add them all: 6,979,584,000 + 525,336,576 + 525,336,576 + 4,096 = **8,030,261,248**.
14. That is 8.03 billion. The model is sold as "8B". The arithmetic closes.

47.2.4 PLAIN - what is really happening inside

1. Here is the whole 8B model drawn as a structure, with the parameter count written next to every piece.

```
       INPUT TOKEN IDS
             |
             v
  +--------------------------+
  | embedding table          |    525,336,576
  | 128,256 rows x 4,096     |
  +--------------------------+
             |
             v
  +==========================+  <-- repeated 32 times
  |  q_proj  16,777,216      |
  |  k_proj   4,194,304      |
  |  v_proj   4,194,304      |      attention
  |  o_proj  16,777,216      |      41,943,040
  |  ----------------------  |
  |  gate    58,720,256      |
  |  up      58,720,256      |      feed-forward
  |  down    58,720,256      |     176,160,768
  |  ----------------------  |
  |  2 norms      8,192      |
  |  layer total 218,112,000 |
  +==========================+
             |     x 32 layers = 6,979,584,000
             v
  +--------------------------+
  | final norm       4,096   |
  +--------------------------+
             |
             v
  +--------------------------+
  | output head              |    525,336,576
  | 128,256 rows x 4,096     |
  +--------------------------+
             |
             v
    ONE SCORE PER TOKEN         TOTAL 8,030,261,248
```

2. Look at where that total came from and a clear pattern appears.
3. The feed-forward matrices are the biggest single item, by a wide margin.
4. In the 8B model, feed-forward is 70 per cent of all parameters.
5. Attention is about 17 per cent.
6. The embedding and output tables together are about 13 per cent.
7. Normalization is a rounding error: three thousandths of one per cent.
8. Now scale the model up and watch the shares move.
9. In the 70B model, feed-forward rises to 80 per cent and embeddings fall to 3 per cent, because the vocabulary stays the same size while the layers grow.
10. In the 405B model, feed-forward is 81 per cent and embeddings are 1 per cent.

11. So the honest one-line answer to "where do the billions live" is: in the feed-forward blocks, repeated once per layer.
12. That matters practically. When people compress models, they attack the feed-forward matrices first, because that is where the mass is.
13. It also explains why small models feel embedding-heavy. In GPT-2 small, the embedding table alone is 25 per cent of the whole model.

47.2.5 TECHNICAL - the engineer's version

1. The general formula for a decoder-only transformer with grouped-query attention, SwiGLU feed-forward and RMSNorm, with untied embeddings, is:

```
per_layer = 2*d*d                (q_proj and o_proj)
          + 2*d*n_kv*head_dim    (k_proj and v_proj)
          + 3*d*d_ff             (gate, up, down)
          + 2*d                  (two RMSNorm scales)

total = V*d                      (input embedding)
      + L*per_layer              (all layers)
      + d                        (final RMSNorm)
      + V*d                      (output head, if untied)
```

2. Here d is hidden size, d_ff feed-forward size, L layers, V vocabulary size, n_kv the number of key-value heads and head_dim the per-head width.
3. Applying it to the three published Llama 3.1 configurations gives exact totals that match the marketing names:

Model	Layers x d x d_ff	Computed total
Llama 3.1 8B	32 x 4096 x 14336	8,030,261,248
Llama 3.1 70B	80 x 8192 x 28672	70,553,706,496
Llama 3.1 405B	126 x 16384 x 53248	405,853,388,800

4. All three use vocabulary 128,256, head dimension 128, and 8 key-value heads. The 405B model has 128 query heads, which is why its k and v projections are only 16384 x 1024 while q and o are 16384 x 16384.
5. Where the parameters sit, as a percentage of the total:

Block	8B	70B	405B
Feed-forward	70.20%	79.90%	81.25%
Attention	16.71%	17.12%	17.71%
Input embedding	6.54%	1.49%	0.52%
Output head	6.54%	1.49%	0.52%

6. Norm parameters are 0.0033 per cent for 8B, 0.0019 per cent for 70B and 0.0010 per cent for 405B, and are omitted from the table above for width.
7. **Tied embeddings** are an implementation choice, not a standard: the output projection reuses the transposed input embedding matrix, saving V x d parameters. GPT-2 ties them. Llama 3.1 8B, 70B and 405B do not. Several small models such as Gemma tie them, which is why their embedding share is larger than you would guess.
8. Verifying GPT-2 small the same way, including biases, gives exactly 124,439,808 parameters: 38,597,376 token embeddings, 786,432 learned position embeddings, 12 layers of 7,087,872 and a final LayerNorm of 1,536. The commonly quoted "117M" for GPT-2 small comes from an older count that excluded the position embeddings and some terms; 124M is the count that matches the released checkpoint.

9. For mixture-of-experts models the count splits in two. Report the **total** parameter count and the **active** parameter count separately, because only a few experts run per token. A model with 8 experts of which 2 run has roughly a quarter of its feed-forward parameters active at any moment. Quoting only the total for such a model is a marketing choice, not a technical one.
10. Tools: `sum(p.numel() for p in model.parameters())` in PyTorch, `model.num_parameters()` in Hugging Face transformers, the safetensors index JSON header, and `gguf-dump --no-tensors file.gguf` for GGUF metadata.

47.2.6 WORDS - remember these

1. Hidden size — how wide the model is — d_model, the width of the residual stream carried between layers, commonly 4096 or 8192.
2. Feed-forward size — the width of the wide middle part — d_ff, usually between 2.7 and 4 times d_model in modern designs.
3. Layer — one repeat of the model's standard block — one transformer block containing attention, feed-forward and two normalizations.
4. Vocabulary size — how many different tokens exist — the number of rows in the embedding matrix, for example 128,256 in Llama 3.1.
5. Grouped-query attention — sharing key and value work across heads — GQA, where n_kv is smaller than n_heads, shrinking k and v projections and the key-value cache.
6. Tied embeddings — reusing the input table as the output table — weight tying between `embed_tokens` and `lm_head`, saving V x d parameters.
7. Active parameters — the ones that actually run for a token — in a mixture-of-experts model, the subset routed to per token, as against the total stored.

47.3 Parameters versus the four things they get confused with

47.3.1 PLAIN - in simple words

1. Five words get mixed up constantly. Here they are, one line each.
2. **Parameters** are the learned numbers inside the model, frozen after training.
3. **Hyperparameters** are the choices a human made before training started. They are not learned. Nobody trained them.
4. **Activations** are the temporary numbers created while the model runs on one input, and thrown away as soon as the answer is produced.
5. **Context** is the text you are giving the model right now. It is data.
6. **Tokens** are the small pieces that text is chopped into. Also data.
7. Now the sharp test. Which of these are in the model file on disk?
8. Only the parameters. The other four are not in the file.
9. Hyperparameters are recorded next to the file, in a small configuration note, so the loader knows what shape to build. But they were not learned.
10. Activations exist only for a moment while a request is being answered.
11. Context and tokens are whatever you happen to be asking about today.
12. So: parameters are what the model is. The rest is how it was built, what it is thinking about right now, and what you asked.

47.3.2 PLAIN - a picture in your head

1. Think of a bakery.
2. The **recipe decisions** are the hyperparameters: oven temperature, tin size, how long to bake. A person chose these before any baking started.
3. The **baker's learned skill** is the parameters: the exact feel for how long to knead, built up over years and now fixed in the baker's hands.

4. The **ingredients on the counter today** are the context: this flour, this water, this order for a birthday cake.
5. The **individual scoops and spoonfuls** are the tokens: the units the ingredients get measured into.
6. The **half-made dough at each stage** is the activations: sticky, temporary, real, and gone the moment the bread is out.
7. Now the useful part. Changing the oven temperature is changing a hyperparameter, and you have to re-train the baker to suit it. Giving new ingredients is changing the context, and needs no re-training at all.

Where this comparison breaks: a baker learns a little from every loaf. A deployed model does not. Its parameters are identical after your millionth message and after your first. Any apparent learning inside a conversation is the context being carried along, not the parameters changing.

47.3.3 PLAIN - a worked example

1. Here is one scenario with all five words used correctly in it.
2. A team trains a model. They choose 32 layers, hidden size 4096, learning rate 0.0003 and batch size 4 million tokens. Those are **hyperparameters**.
3. Training runs for weeks and produces 8,030,261,248 **parameters**, which are saved to a file of about 16 gigabytes.
4. A user types "Summarize this contract" and pastes 3,000 words of contract. That whole thing is the **context**.
5. The tokenizer chops that context into 4,182 **tokens**, each an integer.
6. The model runs. As those 4,182 tokens flow through 32 layers, it creates millions of temporary numbers at every layer. Those are **activations**, and they are discarded when the answer is finished.
7. Now the one-sentence version the reader asked for, all five in one line:
8. "The 8B model's frozen parameters were produced by a training run whose hyperparameters fixed 32 layers and a learning rate of 0.0003; at inference the 4,182 tokens of the user's context are turned into activations that flow through those parameters and are then thrown away."
9. Read it twice. Every one of the five words is doing a different job.

47.3.4 PLAIN - what is really happening inside

1. Trace the lifetime of each of the five things, in order of when it exists.
2. Hyperparameters exist first, before there is a model at all. A human writes them into a configuration file.
3. The training program reads them and builds an empty model of that shape, filled with random numbers.
4. Training begins. The parameters change, millions of times, over weeks.
5. Training ends. The parameters are written to disk and never change again.
6. Later, somebody loads the file. Now the parameters are in memory.
7. A user sends text. That is the context. The tokenizer converts it to tokens.
8. The tokens flow in. At every layer the model computes new temporary numbers. Those are activations.
9. The activations are what carries the meaning of your specific question. The parameters are what does the transforming.
10. When the reply is finished, the activations are freed. The parameters are still sitting there, byte for byte identical.
11. This is why inference needs more memory than the model file size. The activations, and the cached keys and values for the conversation so far, need room too.
12. And it is why a longer conversation costs more memory but does not make the model bigger. The model is exactly as big as it was.

47.3.5 TECHNICAL - the engineer's version

Thing	Learned?	Lives where
Parameter	Yes, by optimizer	Checkpoint, then GPU
Hyperparameter	No, chosen by human	config.json, run script
Activation	No, computed	Transient GPU buffers
Context	No, supplied	Request payload
Token	No, derived	Integer array

1. **Hyperparameters** split into two families that people wrongly lump together. Architectural hyperparameters fix the shape and therefore fix the parameter count: number of layers, hidden size, feed-forward size, head count, vocabulary size, maximum position count. Optimization hyperparameters do not change the count at all: learning rate, warmup steps, weight decay, batch size, dropout rate, gradient clipping threshold, number of epochs.
2. Sampling settings supplied per request, such as temperature, top-p, top-k and repetition penalty, are sometimes loosely called hyperparameters. They are inference settings. They change decoding, not the model.
3. **Activations** are the intermediate tensors of the forward pass. During inference with no gradient tracking, most are freed as soon as the next layer consumes them, so peak activation memory is a few hundred megabytes for a single sequence, not gigabytes.
4. During training the picture inverts. Activations must be retained for the backward pass. Using the accounting from the 2022 NVIDIA paper Reducing Activation Recomputation in Large Transformer Models, activation memory per layer per sample is about $sbh(34 + 5a*s/h)$ bytes in 16-bit, where s is sequence length, b batch, h hidden size and a head count.
5. For a 7B-class model at sequence length 2048 and batch 1 that is about 912 MiB per layer, so about 28.5 GiB across 32 layers for a single sample, before any recomputation is applied. Gradient checkpointing trades compute for memory to cut that sharply.
6. The **key-value cache** is a special, persistent kind of activation. It is the only activation deliberately kept across decoding steps, and it grows linearly with context length. It is covered with numbers in section 47.4.
7. **Context** is data on the request path. It has no gradient, no persistence and no effect on any stored parameter. This is the technical reason a model cannot "learn from your chat" unless a separate fine-tuning job is run.
8. **Tokens** are integers in the range 0 to V-1. They index rows of the embedding matrix. They are not parameters; the rows they select are.
9. Observing them: `config.json` and `generation_config.json` hold hyperparameters, `nvidia-smi` and `torch.cuda.max_memory_allocated()` show activation and cache growth, and any tokenizer's `encode` returns the tokens.

47.3.6 WORDS - remember these

1. Hyperparameter — a choice made before training — a configuration value not updated by gradient descent, either architectural or optimization-related.
2. Activation — a temporary number inside the model while it runs — an intermediate tensor of the forward pass, freed unless retained for backward.
3. Context — the text being processed right now — the input token sequence supplied per request, including any system prompt and history.
4. Key-value cache — the model's notes on the conversation so far — cached per-layer key and value tensors reused across decoding steps.
5. Inference — using the model — a forward pass with no gradient computation and no parameter update.
6. Forward pass — pushing data through the model once — computing outputs layer by layer from inputs.

7. Gradient checkpointing — recomputing instead of storing — discarding activations during forward and recomputing them during backward to save memory at the cost of extra compute.

47.4 Parameters, memory and precision

47.4.1 PLAIN - in simple words

1. Every parameter has to be stored somewhere, and storage takes space.
2. So the memory a model needs is a simple multiplication.
3. Memory = number of parameters x bytes used per parameter.
4. That is the whole rule. There is no magic in it.
5. How many bytes per parameter? That is your choice, and it is a trade.
6. Four bytes stores a number very accurately. Two bytes stores it less accurately but takes half the space.
7. One byte is less accurate still. Half a byte, which means four bits, is cruder again.
8. A 7 billion parameter model at two bytes each needs about 14 gigabytes.
9. The same model at half a byte each needs about 3.5 gigabytes.
10. Same model. Same parameters. Different amount of detail kept per number.
11. But the weights are not the only thing in memory when the model runs.
12. There is also the model's running notes on your conversation, the temporary numbers, and the software's own overhead.
13. And training is far worse again, because training keeps several extra copies of every parameter. We will work that out too.

47.4.2 PLAIN - a picture in your head

1. Think of writing down the price of something.
2. You could write 12.345678. That is very precise and takes many characters.
3. You could write 12.35. Less precise, fewer characters.
4. You could write 12. Cruder still, fewest characters.
5. If you have to write down seven billion prices, the choice matters enormously for how much paper you need.
6. And for most purposes 12.35 is good enough. Nobody notices the missing digits when you are adding up a shopping basket.
7. That is exactly the precision trade in a model. Fewer digits per number, much less memory, slightly less exact answers.
8. Now the training part. To train, you do not only need the current prices.
9. You also need, for every price, a note of which direction to move it, and two running averages of how it has been moving lately.
10. So training needs about four to five sheets of paper for every one sheet that using the finished model needs.

Where this comparison breaks: numbers in a computer are not stored as decimal digits. They are stored in a floating point format with a sign, an exponent and a fraction, and the loss of accuracy is not a clean chopping of digits. It is a coarser grid of representable values, and the grid spacing changes with magnitude.

47.4.3 PLAIN - a worked example

1. Take the same models at four different precisions. Here is the arithmetic for weights only, using 1 GiB = 1,073,741,824 bytes.
2. 7 billion x 4 bytes = 28,000,000,000 bytes = 26.1 GiB.
3. 7 billion x 2 bytes = 14,000,000,000 bytes = 13.0 GiB.
4. 7 billion x 1 byte = 7,000,000,000 bytes = 6.5 GiB.
5. 7 billion x 0.5 bytes = 3,500,000,000 bytes = 3.3 GiB.

6. Repeat for the other sizes and the full table falls out:

Model	fp32 GiB	fp16 GiB
7B	26.1	13.0
13B	48.4	24.2
70B (70.55B)	262.8	131.4
405B (405.85B)	1511.9	756.0

Model	int8 GiB	int4 GiB
7B	6.5	3.3
13B	12.1	6.1
70B (70.55B)	65.7	32.9
405B (405.85B)	378.0	189.0

7. Sanity check against reality: the published fp16 file for Llama 3.1 70B is about 141 GB in decimal gigabytes, which is 131 GiB. The arithmetic matches the real file.
8. Now add what inference needs beyond the weights.
9. The key-value cache for Llama 3.1 70B is 320 KiB per token. At 8,000 tokens that is 2.5 GiB. At 32,000 tokens it is 10 GiB.
10. Temporary activations for one sequence are roughly 0.5 to 1 GiB.
11. The framework's own overhead, buffers and fragmentation is another 1 to 2 GiB in practice.
12. So a realistic total for 70B at fp16 with an 8,000 token context is 131.4 + 2.5 + 1.0 + 2.0 = about 137 GiB, not 131.

47.4.4 PLAIN - what is really happening inside

1. Training memory is a different story, and much bigger, for a simple reason.
2. Training must keep, for every single parameter, four extra things.
3. First, the parameter itself, usually kept twice: a fast low-precision copy used for the maths and an accurate master copy used for the update.
4. Second, the gradient, which is one number per parameter saying which way to nudge it.
5. Third and fourth, the optimizer's two running averages. The standard optimizer, called Adam, keeps a running average of recent gradients and a running average of recent squared gradients.
6. That is what "Adam's two moments" means: two extra numbers per parameter.
7. Count the bytes for the common mixed-precision recipe:

```
16-bit working weights          2 bytes
32-bit master weights           4 bytes
gradients                       4 bytes
Adam first moment  (m)          4 bytes
Adam second moment (v)          4 bytes
                               ----------
total                          18 bytes per parameter
```

8. For a 7B model: 7,000,000,000 x 18 = 126,000,000,000 bytes = 117 GiB.
9. That is before a single activation is stored.
10. Add training activations, which for a 7B-class model at sequence length 2048 come to roughly 28 GiB per sample without recomputation.
11. So training a 7B model is a 150 GiB-plus problem, not a 14 GiB problem.
12. That is why a model you can happily run on one 24 GiB graphics card cannot be fully fine-tuned on it. Using needs weights. Training needs weights plus four more copies plus history.

13. It is also why techniques that train only a small extra set of parameters, which Chapter 50 covers, exist at all.

47.4.5 TECHNICAL - the engineer's version

1. The precision formats, with their real layouts:

Format	Bits	Layout (sign/exp/frac)
fp32	32	1 / 8 / 23
fp16	16	1 / 5 / 10
bf16	16	1 / 8 / 7
fp8 E4M3	8	1 / 4 / 3

2. bfloat16 was introduced by Google for its Tensor Processing Units and keeps fp32's 8-bit exponent, so it has the same dynamic range as fp32 with far fewer fraction bits. That is why bf16 training rarely needs loss scaling while fp16 training usually does.
3. fp16 has a maximum finite value of 65504 and underflows below about 6e-8, which is why gradients in fp16 are scaled up before the backward pass and scaled down afterwards.
4. fp8 in the E4M3 and E5M2 variants was standardized across NVIDIA, Arm and Intel in a 2022 joint proposal, and is supported in hardware from NVIDIA's Hopper generation, announced March 2022. NVIDIA's Blackwell generation, announced 18 March 2024, added hardware support for 4-bit floating point.
5. Key-value cache size per token is exact and worth memorizing:

```
kv_bytes_per_token = 2 * L * n_kv * head_dim * bytes_per_element
```

6. The leading 2 is for keys and values. Real figures at fp16:

Config	Per token	32k ctx	128k ctx
Llama 3.1 8B (GQA 8)	128 KiB	4.0 GiB	16.0 GiB
Llama 3.1 70B (GQA 8)	320 KiB	10.0 GiB	40.0 GiB
Llama 3.1 405B (GQA 8)	504 KiB	15.8 GiB	63.0 GiB
70B-shape without GQA	2560 KiB	80.0 GiB	320.0 GiB

7. That last row is the point of grouped-query attention. Cutting key-value heads from 64 to 8 cuts the cache by a factor of 8, which is what makes long contexts affordable at all.
8. Full-fine-tuning memory in bytes per parameter, by optimizer:

Optimizer state	Bytes/param	7B total
Adam, mixed precision	18	117 GiB
Adam, pure fp32	16	104 GiB
SGD with momentum	12	78 GiB
Inference, fp16	2	13 GiB

9. Adam was published by Diederik Kingma and Jimmy Ba in 2014 as Adam: A Method for Stochastic Optimization; AdamW, the decoupled weight decay variant used by almost every modern language model, came from Ilya Loshchilov and Frank Hutter in 2017. Both keep two moment estimates per parameter, which is the source of the 8 bytes.

10. ZeRO, from the 2019 Microsoft paper ZeRO: Memory Optimizations Toward Training Trillion Parameter Models, shards optimizer states, gradients and parameters across data-parallel ranks in three stages, so the per-device cost falls roughly by the number of devices. It does not reduce the total.
11. Practical observation commands: `nvidia-smi --query-gpu=memory.used --format=csv`, `torch.cuda.memory_summary()`, and for serving, vLLM's reported KV cache blocks and `--gpu-memory-utilization` setting.

47.4.6 WORDS - remember these

1. Precision — how many digits are kept per number — the numeric format's bit width and its split between exponent and fraction.
2. fp32 — the accurate, expensive format — IEEE 754 single precision, 4 bytes, 24 bits of significand including the implicit bit.
3. bf16 — the training-friendly 16-bit format — brain floating point, fp32's exponent range with 8 bits of significand.
4. Key-value cache — the model's stored notes on tokens seen so far — cached K and V tensors per layer, sized 2 x L x n_kv x head_dim per token.
5. Optimizer state — the extra bookkeeping training needs — Adam's first and second moment estimates, one of each per parameter.
6. Mixed precision — computing in 16-bit, updating in 32-bit — a training recipe with a low-precision working copy and an fp32 master copy.
7. GiB versus GB — two different gigabytes — 2^30 = 1,073,741,824 bytes versus 10^9 bytes; file sizes are usually quoted in the decimal form and memory in the binary form, and confusing them costs you 7 per cent.

47.5 Quantization

47.5.1 PLAIN - in simple words

1. **Quantization** means storing each of the model's numbers using fewer bits.
2. That is the whole idea in one line. Fewer bits per number, smaller model.
3. The trained numbers are typically stored in 16 bits each.
4. Quantization rewrites them into 8 bits, or 4 bits, or sometimes fewer.
5. Nothing about the model's shape changes. It still has 70 billion parameters.
6. Each of those 70 billion is just written down more roughly.
7. To do it you need a trick, because a 4-bit slot holds only 16 different values, and the real weights are spread over a wide range.
8. The trick is to store a small group of weights together with one shared scaling number.
9. The 4-bit value says "which of my 16 steps", and the shared scale says "and here is how big a step is".
10. Multiply them back together at load time or on the fly, and you get an approximation of the original number.
11. The approximation is not exact. The question is whether the error matters.
12. Mostly, at 8 bits and 5 bits, it barely does. At 4 bits it is small. Below that it starts to hurt in ways you can measure.

47.5.2 PLAIN - a picture in your head

1. Imagine a shop that must label 10,000 items with prices.
2. The prices range from 1 rupee to 3,000 rupees.
3. The labels only have room for a single digit, 0 to 9.
4. That looks impossible, until you split the shop into departments.
5. In the pencil department, prices run 1 to 10 rupees, so you write on the department sign: "each step on the label = 1 rupee".

6. In the furniture department, prices run 1,000 to 3,000, so that sign says "each step on the label = 200 rupees".
7. Now a label reading 7 means 7 rupees in one department and 1,400 in the other, and one digit is enough everywhere.
8. The department sign is the **scale**. The department is the **group**.
9. Smaller departments mean more accurate prices but more signs to store.
10. That is the entire design space of quantization: how small the groups are, and how much extra you pay to store the signs.

Where this comparison breaks: real quantization also needs a **zero point** when the values are not centred on zero, so the label's 0 does not have to mean a value of 0. And unlike a shop, a badly chosen department boundary does not just annoy a customer; it puts an outlier weight far outside the range and distorts every weight in the group with it.

47.5.3 PLAIN - a worked example

1. Take eight real-looking weights and quantize them by hand.
2. The values: 0.42, -1.13, 0.05, 2.31, -0.77, 1.02, -2.90, 0.66.
3. Symmetric 8-bit first. Find the largest absolute value: 2.90.
4. Scale = 2.90 / 127 = 0.022835.
5. Divide each value by the scale and round:

```
value    /scale    rounded int8
 0.42     18.39         18
-1.13    -49.49        -49
 0.05      2.19          2
 2.31    101.16        101
-0.77    -33.72        -34
 1.02     44.67         45
-2.90   -127.00       -127
 0.66     28.90         29
```

6. To use them, multiply back by the scale. 18 x 0.022835 = 0.4110.
7. Original 0.42, recovered 0.4110. Error 0.0090.
8. Across all eight, the largest error is 0.0111. Small.
9. Now 4-bit, asymmetric, using minimum and maximum instead of absolute maximum.
10. Minimum -2.90, maximum 2.31. Scale = (2.31 - (-2.90)) / 15 = 0.347333.
11. Zero point = round(2.90 / 0.347333) = 8. So the stored code 8 means 0.
12. Quantized codes: 9, 5, 8, 15, 6, 11, 0, 10.
13. Recovered values: 0.3473, -1.0420, 0.0000, 2.4313, -0.6947, 1.0420, -2.7787, 0.6947.
14. Largest error now 0.1213, about eleven times worse than 8-bit. That is the price of four bits instead of eight.
15. Note the storage cost of the scale itself. If one 16-bit scale and one 16-bit zero point cover a group of 64 weights, the real cost is 4 + 32/64 = 4.5 bits per weight, not 4.

47.5.4 PLAIN - what is really happening inside

1. There are three levels of granularity, and they are the main design choice.
2. **Per-tensor**: one scale for an entire matrix. Cheapest to store, worst accuracy, because one outlier anywhere ruins the scale for everything.
3. **Per-channel**: one scale per row or per column. Much better, because a badly behaved output channel no longer poisons its neighbours.
4. **Per-group**: one scale for every fixed run of weights, typically 32, 64, 128 or 256 of them. Best accuracy, most scales to store.
5. Then there is the question of when you quantize.

6. **Post-training quantization** takes a finished model and compresses it. Fast, needs no training run, sometimes needs a few hundred sample texts to calibrate the scales.
7. **Quantization-aware training** simulates the rounding during training, so the model learns weights that survive rounding well. Much more expensive, better results at very low bit widths.
8. Finally: not everything gets squeezed equally.
9. The embedding table and the output layer are often kept at higher precision, because errors there hit every token directly and there is no later layer to absorb the damage.
10. Normalization scales, which are a tiny fraction of parameters, are almost always left at 16 or 32 bits.
11. Some formats keep specific matrices, such as the attention value projection and the feed-forward down projection, at a higher bit width because measurements showed those layers are the most sensitive.
12. And the key-value cache is a separate decision. You can run 4-bit weights with a 16-bit cache, or quantize the cache too.

47.5.5 TECHNICAL - the engineer's version

1. The affine quantization map, which is the standard, is:

```
q = clamp(round(x / s) + z, qmin, qmax)
x_hat = (q - z) * s
```

2. Symmetric quantization fixes z = 0 and sets s = max|x| / qmax. Asymmetric quantization derives s from the range and z from the offset. Symmetric is cheaper at run time because the zero point drops out of the arithmetic.
3. The named methods, with correct attribution:

Method	Year	One-line idea
LLM.int8()	2022	Split out outlier channels
GPTQ	2022	Second-order error correction
AWQ	2023	Protect salient weights
NF4 (QLoRA)	2023	Normal-distribution 4-bit

4. LLM.int8(), by Tim Dettmers and colleagues in 2022, observed that a small number of feature dimensions carry extreme magnitudes and must stay in 16 bits, while the rest go to int8. This is mixed-precision decomposition.
5. GPTQ, by Elias Frantar, Saleh Ashkboos, Torsten Hoefler and Dan Alistarh, posted October 2022 and published at ICLR 2023, quantizes weights column by column and uses approximate second-order information from a small calibration set to adjust the not-yet-quantized weights to compensate for the error just introduced.
6. AWQ, Activation-aware Weight Quantization, by Ji Lin and colleagues at MIT, posted June 2023 and awarded Best Paper at MLSys 2024, observes that protecting roughly 1 per cent of weights, chosen by activation magnitude rather than weight magnitude, recovers most of the loss, and implements the protection by per-channel scaling rather than by keeping mixed precision.
7. NF4, NormalFloat4, comes from the QLoRA paper by Tim Dettmers, Artidoro Pagnoni, Ari Holtzman and Luke Zettlemoyer, May 2023. Its 16 levels are placed at the quantiles of a normal distribution rather than evenly, because trained weights are approximately normally distributed. QLoRA also applies double quantization, quantizing the scales themselves, saving about 0.37 bits per parameter.
8. GGUF is the file format used by llama.cpp, introduced on 21 August 2023 as the successor to GGML, GGMF and GGJT. Its k-quant types use a super-block of 256 weights subdivided into blocks of 32. Exact sizes, read from the ggml header:

Type	Bytes per 256	Bits per weight
Q2_K	84	2.625
Q3_K	110	3.4375
Q4_K	144	4.5
Q5_K	176	5.5
Q6_K	210	6.5625

9. The suffixes S and M mean small and medium mixes. Q4_K_M keeps the attention value projections and the feed-forward down projections at Q6_K in part of the network, so its measured average is about 4.8 bits per weight rather than 4.5. Checking against real published files: Llama 3.1 8B at Q4_K_M is 4.9 GB, and 8.03e9 x 4.83 / 8 = 4.85e9 bytes. The published 405B Q4_K_M file is 243 GB, and 405.85e9 x 4.79 / 8 = 243e9. The arithmetic closes both ways.
10. Quality cost, measured. From a January 2026 evaluation of llama.cpp quantization on Llama-3.1-8B-Instruct, WikiText-2 perplexity, lower better:

Format	Perplexity	Size cut
fp16 baseline	7.32	0%
Q8_0	7.33	46.9%
Q6_K	7.35	59.0%
Q5_K_M	7.40	64.4%
Q4_K_M	7.56	69.4%
Q3_K_M	7.96	75.0%
Q3_K_S	8.96	77.2%

11. Read that table honestly. Q8_0 costs 0.01 perplexity, which is noise. Q4_K_M costs 0.24, about 3.3 per cent relative, for a 69 per cent size cut. Q3_K_S costs 1.64, about 22 per cent relative, and that is a real degradation you will notice.
12. The same study reported that on aggregate downstream scores across GSM8K, HellaSwag, IFEval, MMLU and TruthfulQA, several quantized variants scored within noise of the fp16 baseline, and Q5_0 scored marginally above it. That last result is a reminder that benchmark noise at these margins is larger than the quantization effect, not evidence that quantizing helps.
13. Now the laptop arithmetic the question asks for. Assume about 75 per cent of installed memory is usable for the model, the rest going to the operating system and everything else.

```
Machine with 64 GiB, about 48 GiB usable:
  70B at fp16   131.4 GiB weights  -> does not fit
  70B at int8    65.7 GiB weights  -> does not fit
  70B at Q4_K_M  39.7 GiB weights
                + 2.5 GiB KV at 8k
                + 2.5 GiB other    -> 44.7 GiB, fits

Machine with 16 GiB, about 12 GiB usable:
  8B at fp16     15.0 GiB weights  -> does not fit
  8B at int8      7.5 GiB weights
                + 1.0 GiB KV at 8k
                + 1.3 GiB other    -> 9.8 GiB, fits
  8B at Q4_K_M    4.5 GiB weights  -> 6.8 GiB total, fits easily
  70B at Q4_K_M  39.7 GiB weights  -> does not fit
```

14. So the honest answer to "why can a 70B model run on a laptop at 4-bit but not at 16-bit" is: 131.4 GiB does not fit in 48 GiB and 39.7 GiB does, and that is the entire reason. Note the 64 GiB case only works

with a modest context. Raise the context to 32,000 tokens and the cache alone adds 10 GiB, taking the total to about 52 GiB, which no longer fits.

15. On Apple silicon the calculation uses unified memory, and the default cap on how much the GPU may claim is around 65 to 75 per cent of installed RAM depending on machine and macOS version, which is where the 75 per cent rule of thumb above comes from.
16. Speed, separately from memory: at batch size 1, generation is memory bandwidth bound, so halving the bytes per weight roughly doubles tokens per second. That is often a bigger practical win than the memory saving.
17. Tools: `llama-quantize` in llama.cpp, `AutoGPTQ` and `GPTQModel`, `autoawq`, `bitsandbytes` for NF4 and int8, `llm-compressor` for vLLM-targeted formats, and `optimum-quanto` in the Hugging Face stack.

47.5.6 WORDS - remember these

1. Quantization — storing each number in fewer bits — mapping high-precision weights onto a small integer grid with a scale and optional zero point.
2. Scale — how big one step of the integer grid is — the multiplier s in x_hat = (q - z) * s.
3. Zero point — where the integer grid's zero sits — the offset z that lets an asymmetric range be represented.
4. Group size — how many weights share one scale — commonly 32, 64 or 128; smaller means more accurate and more metadata.
5. Post-training quantization — squeezing a finished model — PTQ, applied to trained weights with at most a small calibration set.
6. Quantization-aware training — training with the rounding simulated — QAT, with fake quantization in the forward pass and straight-through gradients.
7. Perplexity — how surprised the model is by real text — the exponential of the mean negative log-likelihood per token; lower is better, and only comparable between models sharing a tokenizer.
8. Outlier channel — a feature dimension with extreme values — a dimension whose magnitude forces a bad scale for everything grouped with it.

47.6 Why capability does not scale linearly with parameter count

47.6.1 PLAIN - in simple words

1. Bigger models are, on average, better. That much is true.
2. But parameter count is a weak way to guess how good a model is.
3. A model is the product of at least five things, not one.
4. How many parameters it has. How much text it was trained on. How good that text was. How much computing time was spent. What was done to it afterwards.
5. Get any one of those badly wrong and a huge model performs poorly.
6. Get them all right and a small model can beat a much larger one.
7. This is not a rare exception. It happens constantly.
8. In 2022 a 70 billion parameter model called Chinchilla beat a 280 billion parameter model called Gopher, using the same computing budget.
9. It won by being smaller and reading four times as much text.
10. That result changed how the whole field trains models.
11. So when a company says its model has more parameters than a rival's, that is a fact about the file size, not a claim about usefulness that you should accept without evidence.
12. The honest position: parameter count sets a ceiling. It does not tell you how close to that ceiling the model got.

47.6.2 PLAIN – a picture in your head

1. Think of a student preparing for an exam.
2. Parameter count is how much the student could remember if they studied. Call it brain capacity.
3. Training data is how many books they read.
4. Data quality is whether the books were good ones or nonsense from the internet.
5. Training compute is how many hours they actually spent studying.
6. The stages after training are the coaching: practice papers, feedback on answers, learning how to lay out a solution clearly.
7. Now: a student with enormous capacity who read three books and never practised will fail.
8. A student with ordinary capacity who read widely, studied hard and got good coaching will do well.
9. Everyone knows this about students. The same thing is true about models, and the industry spent years talking as though only capacity mattered.

Where this comparison breaks: a student's capacity is fixed and unknown; a model's is chosen and exactly known. And a model has no motivation, no sleep and no forgetting, so the analogy tells you nothing about the training dynamics. It only tells you that one input out of five does not determine the output.

47.6.3 PLAIN – a worked example

1. Take the Chinchilla experiment, published by Jordan Hoffmann and colleagues at DeepMind in March 2022, in Training Compute-Optimal Large Language Models.
2. Gopher: 280 billion parameters, trained on 300 billion tokens.
3. Chinchilla: 70 billion parameters, trained on 1.4 trillion tokens.
4. Both used about the same total computing budget. That is the key control.
5. Chinchilla is four times smaller and read four and a half times more.
6. On MMLU, a broad multiple-choice benchmark, Chinchilla scored 67.5 per cent, more than 7 percentage points above Gopher.
7. It also beat GPT-3 at 175 billion parameters, Jurassic-1 at 178 billion and Megatron-Turing NLG at 530 billion.
8. A 70B model beat a 530B model. That is a factor of seven and a half in parameter count, going the wrong way for the bigger model.
9. The paper's rule of thumb: for a fixed compute budget, model size and training tokens should grow together, roughly 20 tokens per parameter.
10. By that rule, GPT-3 at 175B should have been trained on about 3.5 trillion tokens. It was trained on 300 billion. It was undertrained by roughly a factor of ten.
11. So the entire generation of models before 2022 was too big for the amount of text it had read.

47.6.4 PLAIN – what is really happening inside

1. Training reduces a single measured quantity: the average surprise the model feels when it sees real text.
2. That quantity falls smoothly and predictably as you add parameters, add data or add compute. Those smooth curves are called scaling laws.
3. A scaling law is a fitted curve, not a law of nature. It describes what has been measured over the ranges measured.
4. The curves are also very shallow. To halve the error you need far more than double the size.
5. And low surprise on text is not the same thing as being useful to a person.
6. That gap is closed by the stages after pre-training: supervised fine-tuning on demonstrations, and preference training on human comparisons.
7. Those stages change very few things about the model's knowledge and a great deal about its behaviour.
8. A raw pre-trained model of 70 billion parameters is often unusable as an assistant. The same model after post-training is helpful. Same parameters, different usefulness.

9. Data quality does something similar. Training on carefully filtered, deduplicated, high-quality text produces much better models per parameter than training on raw web scrape.
10. So the causal chain from parameter count to usefulness runs through at least four other factors, each of which can dominate.

47.6.5 TECHNICAL - the engineer's version

1. Separate the three kinds of claim, as the reader asked.
2. **Established fact.** Loss falls as a power law in parameters, data and compute over several orders of magnitude. Kaplan and colleagues at OpenAI established this in January 2020 in Scaling Laws for Neural Language Models.
3. **Established fact.** Kaplan's recommended allocation was corrected by Hoffmann and colleagues in March 2022. Under a fixed compute budget, the compute-optimal ratio is roughly 20 training tokens per parameter, and pre-2022 models were badly undertrained.
4. **Established fact.** The Chinchilla scaling coefficients were partially disputed. Epoch AI published a replication attempt in 2024 finding that the paper's third estimation approach had fitting problems, though the broad conclusion, that data and parameters should scale together, survived.
5. **Established fact, and important.** Compute-optimal is not deployment-optimal. If a model will serve billions of requests, it is rational to train a smaller model far past its Chinchilla point, because inference cost is paid forever and training cost is paid once. Llama 3 8B was trained on about 15 trillion tokens, which is roughly 1,875 tokens per parameter, nearly 100 times the Chinchilla ratio, and it is a much better 8B model for it.
6. **Established fact.** Data curation matters enormously per parameter. Microsoft's phi line, starting with the June 2023 paper Textbooks Are All You Need, showed that filtered and synthetic textbook-style data lets very small models reach scores that previously needed models many times larger, though critics noted benchmark contamination risk in such setups.
7. **Established fact.** Mistral 7B, released 27 September 2023, outperformed Llama 2 13B on all benchmarks its authors reported. A 7B beat a 13B. Neither architecture nor data was held constant, which is exactly the point: parameter count alone did not decide it.
8. **Active research.** How much of frontier capability comes from post-training as against pre-training scale; whether reasoning-focused post-training and test-time compute change the scaling picture; how to measure data quality before spending the compute.
9. **Marketing claim.** Any headline of the form "our model has N parameters, therefore it is better". Also any comparison that quotes total parameters for a mixture-of-experts model against dense parameters for a rival without saying so.
10. **Where experts disagree.** Some argue continued scaling of pre-training is still the main driver of progress; others argue returns from pre-training scale have flattened and that data quality, post-training and inference-time compute now dominate. Both camps include serious researchers, and the public evidence does not settle it, partly because frontier training details are no longer published.
11. Practical consequence for an engineer: never select a model by parameter count. Select by measured performance on your own task, then by cost and latency. Parameter count is only useful for predicting memory and speed, which it predicts extremely well.

Model	Params	Tokens trained	Ratio
GPT-3 (2020)	175B	300B	1.7
Gopher (2021)	280B	300B	1.1
Chinchilla (2022)	70B	1.4T	20
Llama 3 8B (2024)	8B	~15T	~1875

47.6.6 WORDS - remember these

1. Scaling law — a curve saying how much better a bigger model gets — a fitted power law relating loss to parameters, data or compute.
2. Compute-optimal — the best size for a fixed training budget — the parameter-and-token split minimizing loss at fixed FLOPs, about 20 tokens per parameter per Chinchilla.
3. Undertrained — too big for the text it read — a model whose token count is far below its compute-optimal allocation.
4. Overtrained (deliberately) — trained far past compute-optimal on purpose — a smaller model given many more tokens to reduce lifetime inference cost.
5. Pre-training — the long first phase — self-supervised next-token prediction on a very large corpus.
6. Post-training — everything done afterwards — supervised fine-tuning, preference optimization and safety training, covered in Chapter 50.
7. MMLU — a broad general-knowledge exam for models — Massive Multitask Language Understanding, 57 subjects of multiple-choice questions, released 2020.

47.7 Tokens: why text is not fed in as words or characters

47.7.1 PLAIN - in simple words

1. A model works on numbers, so text must become numbers first.
2. The obvious plan is: one number per word. It fails, for four reasons.
3. First, there is no fixed list of all words. New ones appear every day.
4. Second, typos. "recieve" is not in any word list, so it becomes an unknown.
5. Third, many languages do not put spaces between words at all.
6. Fourth, a word list big enough to be useful is enormous, and the model would need one row of numbers per word, which is very expensive.
7. The opposite plan is: one number per letter. That fails too.
8. A page of text becomes thousands of letters, so sequences get very long, and long sequences are expensive.
9. Worse, the model must then learn spelling from scratch before it can learn anything about meaning.
10. So neither extreme works. The answer is in between: chop text into pieces that are usually bigger than a letter and often smaller than a word.
11. Those pieces are called **tokens** or **subwords**.
12. Common words become one token. Rare words become several pieces.
13. A word nobody has ever seen still gets encoded, because in the worst case it falls back to very small pieces.
14. Nothing is ever unknown. That is the property that makes it work.

47.7.2 PLAIN - a picture in your head

1. Think about how a printer's typecase used to work.
2. One option: cut a separate metal block for every word in the language. Millions of blocks, and a new one every time somebody coins a word.
3. Another option: keep only 26 letter blocks and build everything letter by letter. Very few blocks, but setting one page takes forever.
4. What real printers did was in between. They kept letters, and they also kept **ligatures**: single blocks for common combinations like "fi" and "ffl".
5. Frequent combinations got their own block because it saved work.
6. Rare combinations were built up from letters.
7. That is exactly what subword tokenization does, chosen automatically by counting which combinations are frequent in a big pile of text.

Where this comparison breaks: a printer's ligatures were chosen by human craftsmen for how they looked. A tokenizer's pieces are chosen by a counting algorithm for how often they occur, and many of them look strange to a human eye. There is also no requirement that a token be a meaningful unit. Tokens regularly cut straight through the middle of a morpheme.

47.7.3 PLAIN - a worked example

1. Here is byte-pair encoding worked completely, on a tiny corpus.
2. The corpus is four words with counts: low appears 5 times, lower 2 times, newest 6 times, widest 3 times.
3. Start by splitting every word into characters, with an underscore marking the end of a word.
4. The starting symbol set is 11 symbols: `_ d e i l n o r s t w`.
5. Now count every adjacent pair, weighted by word frequency, and merge the most frequent pair. Repeat.

```
step 1: top pairs  es=9  st=9  t_=9  we=8  lo=7  ow=7
        merge e+s -> es
  l o w _        (5)
  l o w e r _    (2)
  n e w es t _   (6)
  w i d es t _   (3)

step 2: top pairs  est=9  t_=9  lo=7  ow=7  ne=6  ew=6
        merge es+t -> est
  n e w est _    (6)
  w i d est _    (3)

step 3: merge est+_ -> est_       (count 9)
  n e w est_     (6)
  w i d est_     (3)

step 4: merge l+o -> lo           (count 7)
step 5: merge lo+w -> low         (count 7)
  low _          (5)
  low e r _      (2)

step 6: merge n+e -> ne           (count 6)
step 7: merge ne+w -> new         (count 6)
step 8: merge new+est_ -> newest_ (count 6)
step 9: merge low+_ -> low_       (count 5)
step 10: merge w+i -> wi          (count 3)
```

6. After 10 merges the words look like this: `low_` (5), `low e r _` (2), `newest_` (6), `wi d est_` (3).
7. The learned merge list, in order, is: es, est, est_, lo, low, ne, new, newest_, low_, wi.
8. The vocabulary is now 11 starting symbols + 10 merges = 21 tokens.
9. Notice what happened. "newest" became a single token because it was frequent.
10. "lower" is still in pieces, because it appeared only twice.
11. "est_" became a token that means, roughly, the superlative ending. Nobody told the algorithm about grammar. It fell out of counting.
12. Real training does this 30,000 to 200,000 times instead of 10 times, on billions of words instead of 16.

47.7.4 PLAIN - what is really happening inside

1. Training a tokenizer and using one are two different programs.
2. Training: count pairs, merge the most frequent, repeat until the vocabulary reaches the target size. Save the ordered merge list and the vocabulary.
3. Using: split the incoming text into characters, then apply the saved merges in exactly the order they were learned. Same order, every time.

4. That ordering is what makes the process deterministic. The same text always gives the same tokens.
5. Then each final piece is looked up in the vocabulary to get its integer id.
6. A crucial detail: modern tokenizers do not start from characters. They start from bytes.
7. There are exactly 256 possible byte values, so every possible input, in any language, in any encoding, including broken text, can always be represented.
8. That is why you never see an "unknown token" error from a modern model.
9. The cost of the byte fallback is that unusual scripts get split into individual bytes, and one character can become three or four tokens.
10. There is also a pre-tokenization step. Before any merging, the text is split by a fixed pattern, usually keeping leading spaces attached to words and splitting runs of digits into groups.
11. That pattern is why " the" with a leading space is a different token from "the" without one, and both exist in the vocabulary.

47.7.5 TECHNICAL - the engineer's version

1. The four families you need to be able to name and tell apart:

Family	Chooses merges by	First described
BPE	Highest pair frequency	Gage 1994; Sennrich 2016
WordPiece	Highest likelihood gain	Schuster, Nakajima 2012
Unigram LM	Prunes a large set	Kudo 2018
SentencePiece	Framework, not algorithm	Kudo, Richardson 2018

2. **Byte-pair encoding** was invented as a data compression algorithm by Philip Gage, published in The C Users Journal in February 1994. Rico Sennrich, Barry Haddow and Alexandra Birch adapted it to machine translation in Neural Machine Translation of Rare Words with Subword Units, posted 2015 and published at ACL 2016. GPT-2 in 2019 moved BPE to operate on raw bytes, giving the 256-symbol base alphabet and eliminating unknown tokens.
3. **WordPiece** came from Mike Schuster and Kaisuke Nakajima at Google in the 2012 ICASSP paper Japanese and Korean Voice Search. It merges the pair that most increases the likelihood of the training data under a unigram language model, rather than the most frequent pair. Concretely it maximizes count(xy) / (count(x) * count(y)), so a pair of two rare symbols can beat a pair of two common ones. BERT, released by Google in October 2018, uses WordPiece, and marks word continuations with a double hash prefix.
4. **Unigram language model** tokenization was introduced by Taku Kudo in the 2018 ACL paper Subword Regularization. It works in the opposite direction: start with a large candidate vocabulary, score every candidate by how much removing it would hurt the corpus likelihood, and iteratively prune the worst until the target size is reached. Because it keeps a probability per token, it can sample alternative segmentations of the same string, which is used as a data augmentation technique.
5. **SentencePiece** is a library, not an algorithm, from Taku Kudo and John Richardson, EMNLP 2018 system demonstrations. It implements both BPE and Unigram, and its real contribution is treating input as a raw stream with no language-specific pre-tokenization, encoding the space character itself as a visible symbol so that detokenization is exactly reversible. That is what makes it work for Japanese, Chinese and Thai, which do not delimit words with spaces. T5, Llama 1 and Llama 2 use SentencePiece; Llama 3 moved to a tiktoken-style byte-level BPE with a 128,000-entry vocabulary.
6. Real vocabulary sizes, for scale:

Tokenizer	Vocabulary	Used by
GPT-2 (r50k)	50,257	GPT-2, GPT-3
cl100k_base	100,277	GPT-3.5, GPT-4

Tokenizer	Vocabulary	Used by
o200k_base	200,019	GPT-4o and later
Llama 3	128,256	Llama 3.x family

7. Larger vocabularies mean fewer tokens per document, which means cheaper attention and shorter sequences, but a larger embedding and output matrix. Going from 50k to 200k vocabulary at hidden size 4096 costs about 1.2 billion extra parameters across the two tables.
8. Special tokens are added outside the learned merges: sequence start, sequence end, padding, and chat role markers such as a header start marker. These are reserved ids that the merge process can never produce from text, which is a deliberate security property: user text cannot forge a role boundary.
9. Tools: `tiktoken` for the OpenAI encodings, `tokenizers` and `AutoTokenizer` in the Hugging Face stack, `sentencepiece` for the T5 and Llama 2 lineage, and `llama-tokenize` in llama.cpp.

47.7.6 WORDS - remember these

1. Token — a small piece of text the model works in — an integer id indexing one row of the embedding matrix, produced by the tokenizer.
2. Subword — a piece smaller than a word — a token that is part of a word, such as “ization” in “tokenization”.
3. Byte-pair encoding — repeatedly glue the most common pair together — BPE, a greedy merge algorithm producing an ordered merge list.
4. Byte-level — starting from raw bytes, not letters — using the 256 byte values as the base alphabet so no input is ever unrepresentable.
5. Pre-tokenization — the split done before merging — a regular expression pass that fixes where merges may not cross, such as across whitespace.
6. Vocabulary — the full list of possible tokens — the id-to-string table, size V, equal to the number of rows in the embedding matrix.
7. Special token — a marker that is not ordinary text — a reserved id such as sequence start or a chat role header, unreachable by merging user text.
8. Detokenization — turning tokens back into text — concatenating the token strings; exactly reversible in SentencePiece and byte-level BPE.

47.8 What tokenization does in practice

47.8.1 PLAIN - in simple words

1. Now let us look at real text going through a real tokenizer.
2. All the counts below are measured, not estimated.
3. English prose splits neatly. Most common words are one token each.
4. Numbers get chopped into groups of digits, and the groups do not line up with how you would read the number.
5. Web addresses split into many small pieces, because slashes, dots and question marks all break the text apart.
6. Program code splits reasonably well, because tokenizers are trained on a lot of code these days.
7. Hindi and other Indian scripts split badly on older tokenizers and much better on newer ones, but still worse than English.
8. An emoji is one character to you and often two tokens to an older model.
9. Two consequences follow directly, and they explain famous model failures.
10. The model cannot see letters, so it cannot reliably count them or reverse a word.
11. The model sees digits in arbitrary groups, so arithmetic on long numbers goes wrong for reasons that have nothing to do with reasoning.

47.8.2 PLAIN - a picture in your head

1. Imagine reading a book through a narrow slot that shows a few characters at a time, but you do not control where the slot boundaries fall.
2. Somebody else cut the page into strips before handing it to you.
3. For English the strips mostly line up with words, so reading is easy.
4. For a number like 17394205 the strips fell at 173, 942, 05.
5. You can read the strips perfectly. You just never saw the digits as digits in their proper columns.
6. Now somebody asks you how many times the letter r appears in "strawberry".
7. You never saw letters. You saw three strips: "str", "aw", "berry".
8. You can guess, from having read the answer somewhere. You cannot count.

Where this comparison breaks: the model is not blind to letters in principle. It has seen enough spelled-out text during training to have learned a lot about which letters are inside which tokens. So it often gets these right. The point is that it is doing recall and inference rather than direct observation, which is why it fails unpredictably rather than never.

47.8.3 PLAIN - a worked example

1. Real measurements, taken with the actual OpenAI tokenizers. cl100k_base is the GPT-4 era encoding; o200k_base is the later one.

Text	cl100k	o200k
The quick brown fox… (44 ch)	10	10
I live in India.	5	5
strawberry	3	3
1234567890	4	4
A 55-character URL	17	18
A 32-character Python function	12	12

2. The exact splits, which are the interesting part:

```
"strawberry"  cl100k -> "str" "aw" "berry"
"strawberry"  o200k  -> "st" "raw" "berry"
"1234567890"  both   -> "123" "456" "789" "0"
"17394205"    both   -> "173" "942" "05"
"unbelievable" cl100k -> "un" "belie" "vable"
"tokenization" both  -> "token" "ization"
```

3. Look at "17394205". The groups are 173, 942, 05. That is not thousands, hundreds, tens. It is an arbitrary cut.
4. Now the code example, tokenized by cl100k_base into 12 tokens:

```
"def" " add" "(a" "," " b" "):NL" "   " " return" " a" " +" " b"
(and a final newline token), where NL stands for a newline
```

5. Note that the four-space indent became a single token of three spaces plus a space attached to "return". Indentation is compressed, which is why code tokenizes efficiently.
6. Now Devanagari. Take the Hindi sentence meaning "I live in India", which is मैं भारत में रहता हूँ।In roman letters that is "main bharat mein rahta hoon". It is 22 characters and 58 bytes of UTF-8, because each Devanagari character takes three bytes where an English letter takes one.

Encoding	Tokens for the Hindi	English is
GPT-2 (r50k)	34	5
cl100k_base	24	5
o200k_base	6	5

7. On the GPT-4 era tokenizer, the same meaning costs 24 tokens in Hindi and 5 in English. That is 4.8 times more, for identical content.
8. A longer measured pair: a 57-character Hindi sentence about tokenization costs 61 tokens on cl100k_base and 21 on o200k_base. The equivalent English sentence costs 11 on both.
9. So on cl100k_base the Hindi is 5.5 times the cost. On o200k_base it is 1.9 times. The gap narrowed a lot but did not close.
10. Emoji: the grinning face character, code point U+1F600, is one character and four UTF-8 bytes. cl100k_base spends 2 tokens on it. o200k_base spends 1.

47.8.4 PLAIN - what is really happening inside

1. Why does Devanagari cost so much on the older tokenizers?
2. Because the tokenizer's merges were learned from a corpus that was mostly English. Frequent English sequences got merged into single tokens.
3. Devanagari sequences were rare, so few merges were learned for them.
4. What is left is the byte fallback. Each Devanagari character is three bytes in UTF-8, and if no merge covers it, those three bytes become separate tokens.
5. That is the 3x penalty in its rawest form, before you even count the missing word-level merges.
6. Newer tokenizers were trained on far more multilingual text and given much larger vocabularies, so many whole Hindi words now have their own token. That is why o200k_base needed only 6 tokens where cl100k_base needed 24.
7. Now the letter-counting failure, mechanically.
8. The model receives the id for "berry" as a single integer. It receives no information whatsoever about the five letters inside it, except what it learned indirectly during training.
9. Asking it to count the letter r in "strawberry" is asking it to recall a fact about the internal composition of three token ids.
10. It can often do this, because spelling appears in training data. It fails when the tokens are unusual, or when the counting has to be exact.
11. Reversing a word is worse, because it requires the letters in order, which is even less directly available.
12. Arithmetic fails for the same class of reason. When 17394205 arrives as 173, 942, 05, the place values that column-wise addition depends on are not aligned with anything the model can see.
13. This is why some model families deliberately force digits to be split one at a time, and why models are much better at arithmetic when allowed to write the working out step by step rather than answering in one shot.

47.8.5 TECHNICAL - the engineer's version

1. Measured compression ratios, cl100k_base, on real text samples:

Text type	Chars/token	Tokens/word
Plain narrative prose	3.78	1.25
Ordinary email	4.58	1.12
Technical prose	4.87	1.07
Python code	3.76	2.07
Compact JSON	2.76	3.22

2. The common rules of thumb, roughly 4 characters per token and roughly 0.75 words per token in English, are conventions rather than standards, and the table shows they hold for prose and fail badly for structured data. JSON at 2.76 characters per token is 45 per cent worse than the rule of thumb suggests, which matters when you are budgeting a prompt full of API output.
3. The unfairness is documented. Aleksandar Petrov, Emanuele La Malfa, Philip Torr and Adel Bibi published Language Model Tokenizers Introduce Unfairness Between Languages at NeurIPS 2023, measuring the same content across many languages and finding token-count ratios up to 15 times between the best and worst served languages, with over 4 times difference remaining even for character and byte-level encoders.
4. The three consequences they name are exactly the ones that hurt an Indian developer: cost, because APIs bill per token; latency, because more tokens means more forward passes; and effective context, because a fixed window holds far less meaning in a penalized language.
5. Work the third one out. A 128,000-token window at 4.8 characters per token holds about 614,000 characters of English. At the cl100k_base Hindi rate of about 0.93 characters per token, the same window holds about 119,000 characters of Hindi. The window is the same. The content it holds is a fifth.
6. Mitigations that actually work: choose a model whose tokenizer was trained with substantial data in your language, measure with the real tokenizer rather than trusting a rule of thumb, and check whether the provider's multilingual pricing is per token, which it almost always is.
7. On the letter-counting failure: the standard explanation is tokenization, and it is largely right, but be precise. Character-level models also make counting mistakes, and models can spell tokens correctly when asked step by step. So tokenization makes the information indirect rather than absent. That is an important distinction and it is often stated too strongly.
8. On arithmetic: the tokenizer's digit grouping is a genuine and measurable cause. Research on numeric tokenization has shown that forcing single-digit tokenization, or right-to-left digit grouping, measurably improves multi-digit arithmetic. Some model families now tokenize digits individually for this reason.
9. Practical tools: `tiktoken.get_encoding("cl100k_base").encode(text)` returns the ids; `enc.decode_single_token_bytes(i)` shows exactly what each id is; Hugging Face's `tokenizer.tokenize(text)` shows the string pieces.

```
import tiktoken
enc = tiktoken.get_encoding("cl100k_base")
ids = enc.encode("The cat sat on the mat.")
print(len(ids), ids)
for i in ids:
    print(i, enc.decode_single_token_bytes(i))
```

47.8.6 WORDS - remember these

1. Compression ratio — how much text fits in one token — measured characters per token or tokens per word for a given tokenizer and text type.
2. Byte fallback — what happens when no merge covers the text — encoding characters as their individual UTF-8 bytes, one token each.
3. Tokenizer fertility — how many tokens a tokenizer spends per word — a standard fairness measure; higher fertility means a worse-served language.
4. Digit grouping — how numbers get cut up — the pre-tokenizer rule that splits runs of digits, commonly into groups of up to three.
5. UTF-8 — the standard way text becomes bytes — 1 byte for ASCII, 2 for Latin accented and Greek, 3 for Devanagari and most CJK, 4 for emoji.
6. Effective context — how much meaning your window actually holds — the window size divided by your language's tokens-per-meaning cost.

47.9 Token economics

47.9.1 PLAIN - in simple words

1. When you use a model through a company's service, you pay per token.
2. Not per question. Not per minute. Per token in and per token out.
3. This is not arbitrary. Tokens are what the machine actually does work on.
4. One token in costs the model one pass through the multiply-and-add machinery, though those can be done for many tokens at once.
5. One token out costs a full pass through the entire model, and those cannot be done in parallel, because each output token depends on the one before.
6. That asymmetry is why output tokens cost more than input tokens.
7. Typically output is priced at four to six times input.
8. So a long question with a short answer is cheap. A short question with a very long answer is not.
9. There is a second, sneakier cost. Long conversations resend everything.
10. If you have a chat with 50 turns, every new question sends all 50 previous turns as input, because the model has no memory between requests.
11. So the input token count grows with the conversation and you pay for the whole history every single time.
12. That is the single most common reason a bill is bigger than expected.

47.9.2 PLAIN - a picture in your head

1. Think of a translator you hire, who charges by the page.
2. Pages they read cost you a little. Pages they write cost you more, because writing is slower than reading.
3. That is input versus output pricing.
4. Now the important twist. This translator has no memory at all.
5. Every time you ask a follow-up question, you must hand them the entire previous conversation to read again, from the beginning.
6. On question one you hand over one page. On question fifty you hand over sixty pages, and pay to have all sixty read again.
7. Some translators offer a discount: if you hand them the exact same bundle of pages you handed them a few minutes ago, they charge much less for those.
8. That discount is prompt caching, and it is real.

Where this comparison breaks: a human translator reading sixty pages takes an hour. A model processes the input tokens of a long prompt in parallel, so sending a long history costs money and some latency but nothing like proportionally more wall-clock time than a short one.

47.9.3 PLAIN - a worked example

1. Build a realistic support assistant and cost it.
2. Every request contains: a system prompt of 800 tokens, four retrieved document chunks totalling 4,000 tokens, 1,200 tokens of conversation history, and the user's 150-token question.
3. Input total: 800 + 4,000 + 1,200 + 150 = 6,150 tokens.
4. The answer averages 350 tokens of output.
5. Prices per million tokens, checked on 13 August 2026, input then output:

Model	Input $/1M	Output $/1M	Per request
Claude Sonnet 5	2.00	10.00	$0.0158
Claude Opus 5	5.00	25.00	$0.0395
gpt-5.6-sol	2.50	15.00	$0.0206
Gemini 3.6 Flash	1.50	7.50	$0.0119

6. Work one out fully, for Claude Sonnet 5.
7. Input: 6,150 / 1,000,000 x $2.00 = $0.0123.
8. Output: 350 / 1,000,000 x $10.00 = $0.0035.
9. Total per request: $0.0158.
10. At 10,000 requests a day that is $158 a day, or $4,740 a month.
11. On Claude Opus 5 the same traffic is $11,850 a month.
12. On a small cheap model such as gpt-5.6-luna at $0.10 in and $0.60 out, the same traffic is $247.50 a month.
13. Notice that the input tokens dominate: $0.0123 of the $0.0158, or 78 per cent. Cutting retrieved context is worth far more than shortening answers.

47.9.4 PLAIN - what is really happening inside

1. Watch what happens when the context grows and everything else stays fixed.

Input tokens	Cost per request	Per 10,000
6,150	$0.0158	$158
14,150	$0.0318	$318
38,150	$0.0798	$798
106,150	$0.2158	$2,158
406,150	$0.8158	$8,158

2. Those are all the same question, the same answer, and the same model. The only change is how much context was stuffed in.
3. Going from 6,000 to 400,000 tokens of context multiplies the bill by 52.
4. This is why "just put the whole document in the context window" is a financial decision, not only a technical one.
5. Now caching. The system prompt and the retrieved chunks are often identical between requests within a short window.
6. Providers offer a large discount on input tokens that are an exact prefix match of a recent request. One current example prices cached input at one tenth of fresh input.
7. Rework the example on a provider charging $2.50 fresh, $0.25 cached and $15.00 output. Without caching: $0.0206 per request.
8. With the 4,800-token system-plus-documents prefix cached: 4,800 at $0.25 plus 1,350 at $2.50 plus 350 at $15.00 = $0.0098.
9. That is a 52 per cent saving for changing nothing but the ordering and stability of the prompt.
10. The rule that follows: put the stable parts of your prompt first and the variable parts last, because prefix caching only matches from the start.

47.9.5 TECHNICAL - the engineer's version

1. Why output costs more, precisely. Processing input is the **prefill** phase: all input tokens go through the model in one batched forward pass, and the hardware is compute-bound and highly efficient.
2. Producing output is the **decode** phase: one token at a time, each requiring a full pass over every weight, and at small batch sizes the hardware is memory-bandwidth-bound and mostly idle in its arithmetic units.
3. So a decoded token costs far more machine time than a prefilled one, and the pricing reflects that. The typical ratio of 4x to 6x is a convention that has held across providers, not a standard.
4. Reasoning models add a third category. Tokens generated internally before the visible answer are billed as output tokens even though you never see them, and they can be several times the visible answer's length. Budget for them explicitly.

5. Long-context surcharges are now common. As of August 2026, one provider prices prompts over 200,000 tokens at double the input rate and 1.5 times the output rate; another prices a long-context tier at double input and 1.5 times output above a threshold. Check the specific model's page, because this changes often.
6. Batch APIs typically halve the price in exchange for asynchronous delivery within a window, commonly 24 hours. If your workload is not interactive, this is the largest single discount available.
7. Estimating before you build: tokenize a representative sample of your real prompts with the actual tokenizer and take the mean. Do not estimate from character counts if your content is JSON, code or a non-Latin script, because the rule of thumb is wrong by up to a factor of five in those cases.
8. A checklist that reduces token spend, in order of typical impact:

```
1. Retrieve less. Four chunks instead of twenty.
2. Cache the stable prefix; put variable text last.
3. Summarize old conversation turns instead of resending.
4. Use a small model for routing and a large one only when needed.
5. Cap max output tokens per request.
6. Batch anything that is not interactive.
7. Strip whitespace and boilerplate from injected documents.
```

9. Self-hosting changes the shape of the cost, not always its size. You pay for the machine by the hour whether it is busy or not, so self-hosting wins at high steady utilization and loses badly at low or spiky utilization.
10. Observability: every provider returns a usage object with input and output token counts per response. Log it per request with a feature label, or you will not be able to attribute the bill later.

47.9.6 WORDS - remember these

1. Input token — a token you send — a token in the prompt, processed in the prefill phase, billed at the lower rate.
2. Output token — a token the model writes — a token produced in the decode phase, billed at the higher rate, including hidden reasoning tokens.
3. Prefill — reading the prompt — the parallel forward pass over all input tokens, populating the key-value cache.
4. Decode — writing the answer — the sequential per-token generation loop, one full forward pass per token.
5. Prompt caching — paying less for a repeated prefix — reusing a stored key-value cache for an exact matching prompt prefix, at a discounted rate.
6. Batch API — cheaper if you can wait — asynchronous processing at roughly half price with a delivery window, commonly 24 hours.
7. Context stuffing — putting everything in the prompt — a strategy whose cost grows linearly in tokens and whose attention cost grows faster.

47.10 Embeddings: turning a token id into a vector

47.10.1 PLAIN - in simple words

1. After tokenization you have a list of integers. The model cannot use those directly.
2. An integer id like 8415 is just a name. Token 8416 is not "one more than" token 8415 in any meaningful way.
3. So each id has to be turned into something with actual structure.
4. It is turned into a list of numbers. A long one: often 4,096 numbers.
5. That list is called the token's **embedding**.
6. Where does the list come from? A big table.
7. The table has one row per possible token, and one column per number in the list.

8. If there are 128,256 possible tokens and each embedding is 4,096 numbers, the table is 128,256 rows by 4,096 columns.
9. Turning an id into an embedding is just reading row number 8415 of the table.
10. That is a lookup. No multiplication, no cleverness. Fetch the row.
11. The numbers in the table were learned during training, so the table is made of parameters, exactly like every other matrix in the model.
12. And that is the entrance to the model. Text became ids. Ids became rows of numbers. Now the layers can do arithmetic.

47.10.2 PLAIN - a picture in your head

1. Think of a very long index card box, with one card per word.
2. On each card, instead of a definition, there are 4,096 numbers.
3. To look up a word you find its card and copy the numbers off it.
4. The numbers mean nothing to you individually. Number 1,872 on the card for "cat" is 0.031, and there is no sentence explaining why.
5. But the cards are arranged in a very particular way in an imaginary space.
6. Cards for similar things ended up near each other. The card for "cat" is close to the card for "dog", and far from the card for "invoice".
7. Nobody sorted them deliberately. They ended up there because training pushed words used in similar ways toward similar numbers.
8. So the meaning is in the arrangement, not in any single number on any card.

Where this comparison breaks: index cards sit on a table, which is two dimensional. These cards sit in a space with 4,096 directions. Almost every intuition you have about "near" and "far" from two dimensions gets weaker in 4,096, and some of them break completely. That gets its own honest treatment below.

47.10.3 PLAIN - a worked example

1. Take a real sentence and follow it in.
2. The sentence: "The cat sat on the mat."
3. Tokenized with cl100k_base it becomes 7 tokens with these real ids:

Id	Piece
791	The
8415	(space)cat
7731	(space)sat
389	(space)on
279	(space)the
5634	(space)mat
13	.

4. Note that "The" at the start and " the" in the middle are different tokens with different ids, because one has a leading space and a capital letter.
5. Now the lookup. The embedding matrix for an 8B-class model is 128,256 x 4,096.
6. Row 791 is fetched. It is 4,096 numbers. Row 8415 is fetched. Also 4,096.
7. Do that for all seven ids and stack them.
8. The result is a block of numbers 7 rows by 4,096 columns.
9. In the shape notation engineers use, the tensor went from [7] to [7, 4096].
10. That is 28,672 numbers describing a six-word sentence.
11. Size of the table itself: 128,256 x 4,096 = 525,336,576 parameters.
12. At two bytes each, that one table is about 1.05 gigabytes, which is 6.5 per cent of the whole 8B model.

47.10.4 PLAIN - what is really happening inside

1. The lookup is mathematically a matrix multiplication, done the lazy way.
2. The formal version: make a vector of 128,256 numbers that is all zeros except a single 1 in position 8415. That is called a one-hot vector.
3. Multiply that one-hot vector by the embedding matrix. The result is row 8415.
4. Every implementation skips the multiplication and just indexes the row, because multiplying by 128,255 zeros is a waste.
5. That is why the operation is called an embedding lookup rather than an embedding multiply, even though it is a linear layer.
6. One thing is missing at this point: order.
7. The lookup gives the same vector for "cat" whether it is the first word or the fiftieth. Position information has to be added separately.
8. Older models added a learned position vector to the token vector. Modern models mostly rotate the query and key vectors inside attention instead, which Chapter 48 covers.
9. From here the stacked embeddings flow into layer 1, then layer 2, and so on.
10. Critically, the vector at each position keeps being updated as it passes through layers. It starts as "the meaning of this token in general" and becomes "the meaning of this token in this sentence".
11. At the very end, the vector at the last position is multiplied by the output matrix to produce one score per possible token, and the highest scores are the model's prediction of what comes next.

47.10.5 TECHNICAL - the engineer's version

1. The embedding matrix has shape [V, d] and is exactly V x d parameters.

Model	V	d	Embedding params
GPT-2 small	50,257	768	38,597,376
Llama 3.1 8B	128,256	4,096	525,336,576
Llama 3.1 70B	128,256	8,192	1,050,673,152
Llama 3.1 405B	128,256	16,384	2,101,346,304

2. In PyTorch this is `nn.Embedding(V, d)`, whose forward pass is `F.embedding`, which is an index-select, not a matmul.
3. The output projection, `lm_head`, has shape [V, d] as well and produces **logits**: one raw score per vocabulary entry. Softmax over logits gives the next-token distribution.
4. If embeddings are tied, `lm_head.weight` is the same storage as `embed_tokens.weight`. Tying was popularized by Press and Wolf in 2016 in Using the Output Embedding to Improve Language Models, and independently by Inan, Khosravi and Socher the same year.
5. Common hidden sizes, which are also the embedding dimensions: 768 for GPT-2 small, 1024 and 1600 for larger GPT-2 variants, 2048 for many 1B-class models, 4096 for 7B and 8B, 5120 for 13B, 8192 for 70B, 16384 for 405B.
6. Now the honest note about dimensionality that the reader asked for.
7. You cannot visualize 4,096 dimensions. Nobody can. Any picture you have seen of word vectors as arrows on a page is a projection down to two dimensions by a method such as PCA, t-SNE or UMAP, and those projections distort distances substantially. t-SNE in particular preserves local neighbourhoods and does not preserve global distances at all, so cluster sizes and the gaps between clusters in a t-SNE plot are not meaningful.
8. What does carry over from 2D and 3D: dot products, cosine of the angle between vectors, Euclidean distance, addition and subtraction of vectors, and the idea that direction can encode something.
9. What does not carry over, and this is the part people get wrong:
    1. In high dimensions almost every pair of random vectors is nearly orthogonal, so a cosine of 0 means "unrelated", not "opposite".

2. Distances concentrate: the ratio between the nearest and farthest point in a random set shrinks toward 1 as dimension grows, which is one face of the curse of dimensionality.
3. The volume of a high-dimensional ball is almost entirely near its surface, so intuitions about "the middle of the cloud" are wrong.
4. There is far more room than you expect. In d dimensions you can pack exponentially many nearly-orthogonal directions, which is why a 4,096-wide space can hold far more than 4,096 distinguishable concepts.

10. That last point has a name in current research: **superposition**, the hypothesis that models represent many more features than they have dimensions, by using nearly-orthogonal directions and tolerating small interference. Anthropic's 2022 paper Toy Models of Superposition is the standard reference. This is active research, not settled fact.

47.10.6 WORDS - remember these

1. Embedding — a token turned into a list of numbers — a row of the embedding matrix, a vector in R^d.
2. Embedding matrix — the lookup table — a [V, d] parameter tensor mapping token ids to vectors.
3. One-hot vector — all zeros with a single 1 — the formal input to an embedding layer, never materialized in practice.
4. Logits — raw scores before they become probabilities — the [V] output of the final projection, converted by softmax.
5. Dimension — how many numbers are in one vector — d, the model's hidden size, typically 768 to 16,384.
6. Curse of dimensionality — high-dimensional space behaves strangely — the family of effects including distance concentration and near-orthogonality of random vectors.
7. Superposition — packing more ideas than there are directions — the hypothesis that features are stored along nearly-orthogonal directions with tolerated interference. Active research.

47.11 What embeddings capture

47.11.1 PLAIN - in simple words

1. If two words have similar embeddings, the model treats them similarly.
2. So the natural question is: how do you measure "similar" for two lists of numbers?
3. Two standard answers, both simple.
4. **Distance**: treat each list as a point, and measure how far apart the points are. Closer means more similar.
5. **Angle**: treat each list as an arrow from the origin, and measure the angle between the arrows. Smaller angle means more similar.
6. The angle version is the one almost everybody uses, and it is called **cosine similarity**.
7. It gives 1 for arrows pointing the same way, 0 for arrows at right angles, and -1 for arrows pointing opposite ways.
8. The angle version ignores how long the arrows are and looks only at direction, which is usually what you want.
9. There is also a famous claim that these vectors capture relationships, not just similarity.
10. The claim is that king minus man plus woman lands near queen.
11. It is a real effect. It is also considerably weaker than the popular story, and the popular demonstration was rigged in a specific way. Section 47.11.3 and 47.11.5 give the honest account.

47.11.2 PLAIN - a picture in your head

1. Think of a map of a country, where each town is a dot.
2. Two towns close together on the map are similar in location. That is distance.
3. Now draw an arrow from the capital to every town.

4. Two towns roughly in the same direction from the capital, one near and one far, have arrows at a small angle. That is cosine similarity.
5. On a map, "50 km north-east" is a direction you can apply anywhere. Start at any town, go 50 km north-east, arrive somewhere new.
6. The analogy claim is that word space has directions like that. There is a direction that means "make it female", and applying it to king lands on queen.
7. The map picture makes clear why this could work and why it might not.
8. On a real map, "50 km north-east" from a coastal town lands in the sea.

Where this comparison breaks: a map has two dimensions and every direction is meaningful. Word space has thousands, and only a tiny number of directions correspond to anything a human would name. Most directions in the space mean nothing at all.

47.11.3 PLAIN - a worked example

1. Cosine similarity, computed by hand on small vectors.
2. Take four-number toy vectors, made up for illustration:

```
cat = [0.8, 0.1, 0.7, 0.2]
dog = [0.7, 0.2, 0.8, 0.1]
car = [0.1, 0.9, 0.0, 0.8]
```

3. The formula: cosine = (a . b) / (|a| x |b|), where a . b is the sum of pairwise products and |a| is the square root of the sum of squares.
4. Step 1, the dot product of cat and dog: 0.8x0.7 + 0.1x0.2 + 0.7x0.8 + 0.2x0.1 = 0.56 + 0.02 + 0.56 + 0.02 = 1.16.
5. Step 2, the length of cat: sqrt(0.64 + 0.01 + 0.49 + 0.04) = sqrt(1.18) = 1.0863.
6. Step 3, the length of dog: also 1.0863.
7. Step 4: cosine = 1.16 / (1.0863 x 1.0863) = 1.16 / 1.18 = 0.9831.
8. An angle of 10.6 degrees. Very similar.
9. Now cat against car. Dot product = 0.08 + 0.09 + 0.00 + 0.16 = 0.33. Lengths 1.0863 and 1.2083. Cosine = 0.33 / 1.3126 = 0.2514.
10. An angle of 75.4 degrees. Nearly unrelated.
11. Compare with straight-line distance: cat to dog is 0.20, cat to car is 1.41. Both measures agree here, which is common but not guaranteed.
12. Now the analogy, on a three-number toy set:

```
king     = [0.92, 0.81, 0.12]
man      = [0.88, 0.14, 0.09]
woman    = [0.13, 0.16, 0.91]
queen    = [0.15, 0.74, 0.86]
princess = [0.20, 0.60, 0.88]
```

13. king - man + woman = [0.17, 0.83, 0.94].
14. Cosine of that result against every word in the toy set:

Word	Cosine with result
queen	0.9999
princess	0.9911
woman	0.8556
king	0.6041
man	0.3092

15. Queen wins. The demonstration works. But look at the third row: woman, one of the inputs, scores 0.8556, well above king and man.
16. That third row is where the honest correction lives.

47.11.4 PLAIN - what is really happening inside

1. The standard way to answer an analogy from vectors is called 3CosAdd.
2. Compute b - a + c, then find the vocabulary word whose vector has the highest cosine with the result.
3. There is a rule buried in the standard implementation: the three input words are excluded from the search.
4. So when you compute king - man + woman, the code refuses to answer king, man or woman, no matter how well they score.
5. That matters a great deal, because very often the highest-scoring word is one of the inputs.
6. The result you are shown is therefore the best answer after the most likely answers have been forbidden.
7. That does not make the effect fake. Queen really does score highly, and consistently, on many word pairs.
8. It makes the effect smaller and less magical than the demonstrations imply.
9. When researchers removed the exclusion rule and let the inputs compete, accuracy collapsed. The numbers are in the technical block below.
10. There is a second, separate honesty point about these vectors.
11. In word2vec, each word has exactly one vector, forever. "Bank" gets one vector that has to serve both the river bank and the money bank.
12. Modern models do not work that way. The vector for a word changes depending on the sentence it appears in.

47.11.5 TECHNICAL - the engineer's version

1. Word2vec was published by Tomas Mikolov and colleagues at Google in 2013, in two papers: Efficient Estimation of Word Representations in Vector Space in January 2013, and Distributed Representations of Words and Phrases and their Compositionality at NeurIPS 2013. The analogy result appeared in a third, Linguistic Regularities in Continuous Space Word Representations, by Mikolov, Yih and Zweig at NAACL in June 2013.
2. GloVe followed from Jeffrey Pennington, Richard Socher and Christopher Manning at Stanford, EMNLP 2014, fitting vectors to global co-occurrence counts rather than local context windows.
3. The vector offset method, 3CosAdd, ranks candidate d by $\cos(d, b - a + c)$. Omer Levy and Yoav Goldberg proposed 3CosMul in 2014 in Linguistic Regularities in Sparse and Explicit Word Representations, which multiplies similarities instead of adding offsets and performs better.
4. Now the correction, which is well documented and not fringe.
5. Tal Linzen's 2016 RepEval paper Issues in Evaluating Semantic Spaces Using Word Analogies showed that a large part of reported analogy accuracy is explained by simple baselines: for many analogy items, just returning the nearest neighbour of c, ignoring a and b entirely, scores well.
6. Malvina Nissim, Rik van Noord and Rob van der Goot published Fair Is Better than Sensational: Man Is to Doctor as Woman Is to Doctor in Computational Linguistics, volume 46 issue 2, 2020. They showed the standard implementations forbid returning any of the three input words, and that this is not a detail.
7. Their measured result: when they modified the algorithms to allow input words as answers, 3CosAdd accuracy fell from 71 per cent to 21 per cent, and 3CosMul fell from 73 per cent to 45 per cent.
8. Read that carefully. On the standard benchmark, roughly two thirds of 3CosAdd's apparent success disappears when the exclusion is lifted. The frequent correct-if-allowed answer is one of the inputs, usually b.
9. So the accurate statement is: the offset direction carries real relational information, and it is much weaker than the "king - man + woman = queen" story suggests. Both halves of that sentence are true, and popular accounts usually give only the first half.

10. The same paper makes a separate methodological point about bias claims: asking a system "man is to doctor as woman is to what" while forbidding it from answering "doctor" guarantees a different answer, and then reporting that different answer as evidence of bias is unsound. Bias in embeddings is real and separately demonstrated; that particular demonstration is not sound evidence of it.
11. **Static versus contextual embeddings.** Word2vec and GloVe are static: one vector per word type, fixed. Contextual embeddings give a different vector per occurrence.
12. ELMo, from Matthew Peters and colleagues at the Allen Institute, NAACL 2018, Deep Contextualized Word Representations, was the first widely used contextual model. BERT, from Jacob Devlin and colleagues at Google, posted October 2018 and published at NAACL 2019, made it standard.
13. In a transformer, the row fetched from the embedding matrix is static, and it becomes contextual immediately: after the first attention layer the vector at that position has mixed in information from the other tokens.
14. So in the sentence "I sat on the river bank" and "I went to the bank for a loan", the input embedding for "bank" is byte-for-byte identical, and the representation at layer 12 is quite different. That is the whole point of the architecture.

Property	Static (word2vec)	Contextual
Vectors per word	One	One per occurrence
Handles polysemy	No	Yes
Needs a model to compute	No, table lookup	Yes, forward pass
Typical year	2013 to 2017	2018 onward

47.11.6 WORDS - remember these

1. Cosine similarity — how aligned two lists of numbers are — the dot product divided by the product of the norms; 1 identical direction, 0 orthogonal.
2. Euclidean distance — straight-line distance between two points — the square root of the sum of squared differences.
3. Dot product — pairwise multiply and add — the unnormalized similarity, which grows with vector length as well as alignment.
4. Vector offset — subtracting one vector from another — the b - a term in the 3CosAdd analogy method, treated as a relation direction.
5. Static embedding — one fixed vector per word — word2vec or GloVe style, a pure lookup with no sentence context.
6. Contextual embedding — a different vector each time — the hidden state at a position after attention has mixed in surrounding tokens.
7. Polysemy — one word with several meanings — the phenomenon static embeddings cannot represent and contextual embeddings can.

47.12 Embeddings as a tool in their own right

47.12.1 PLAIN - in simple words

1. So far embeddings have been a part inside a language model.
2. They are also a useful product on their own, sold separately.
3. The idea: instead of one vector per token, get one vector for a whole sentence, paragraph or document.
4. Now you can compare pieces of text by comparing numbers.
5. That single capability unlocks five very common jobs.
6. **Semantic search**: find documents that mean the same thing as a query, even if they share no words with it.
7. **Clustering**: group thousands of support tickets into themes automatically.

8. **Recommendation**: suggest articles near the ones a reader liked.
9. **Deduplication**: find near-identical items that keyword matching misses.
10. **Retrieval for a model**: fetch the few relevant paragraphs from a large collection and paste them into the prompt.
11. That last one is the foundation of retrieval-augmented generation, which Chapter 51 covers in full.
12. The thing that makes all five work is the same: meaning became geometry, and geometry is cheap to compute with.

47.12.2 PLAIN - a picture in your head

1. Imagine a library where every book has been assigned a position in a very large warehouse.
2. Books are not shelved by author or title. They are shelved by what they are about.
3. Two books about monsoon farming end up on adjacent shelves even if one is in English and one in Hindi and they share no words.
4. To answer a question you do not read anything. You compute where the question would sit in the warehouse, walk to that spot, and take the nearest ten books.
5. That is semantic search, exactly.
6. Keyword search is the old catalogue: it finds books whose title contains your exact words, and misses everything phrased differently.

Where this comparison breaks: a warehouse has aisles you can walk. A vector space has thousands of dimensions and no aisles, so finding the nearest items needs a specialized index. Doing it by brute force means comparing against every item, which is fine for ten thousand documents and far too slow for a hundred million.

47.12.3 PLAIN - a worked example

1. Suppose four short documents and one query, each already turned into a four-number vector by an embedding model.

```
query = [0.75, 0.15, 0.60, 0.20]

doc A "how to feed a kitten"  = [0.80, 0.10, 0.70, 0.20]
doc B "puppy training basics" = [0.70, 0.20, 0.80, 0.10]
doc C "car insurance renewal" = [0.10, 0.90, 0.00, 0.80]
doc D "truck loading limits"  = [0.20, 0.80, 0.10, 0.90]
```

2. Compute the cosine of the query against each.
3. The query's length is sqrt(0.5625 + 0.0225 + 0.36 + 0.04) = sqrt(0.985) = 0.9925.
4. Query and doc A: dot = 0.60 + 0.015 + 0.42 + 0.04 = 1.075. Doc A's length is 1.0863. Cosine = 1.075 / (0.9925 x 1.0863) = 0.9971.
5. Query and doc B: dot = 0.525 + 0.03 + 0.48 + 0.02 = 1.055. Cosine = 0.9786.
6. Query and doc C: dot = 0.075 + 0.135 + 0.00 + 0.16 = 0.37. Cosine = 0.3085.
7. Query and doc D: dot = 0.15 + 0.12 + 0.06 + 0.18 = 0.51. Cosine = 0.4196.

Document	Cosine	Rank
A, kitten feeding	0.9971	1
B, puppy training	0.9786	2
D, truck loading	0.4196	3
C, car insurance	0.3085	4

8. The two animal documents rank above the two vehicle documents, and no word from the query appears in any of them.

9. In a real system the vectors have 384, 768, 1024 or 3072 numbers instead of four, and there are millions of documents, but the computation per pair is exactly this one.

47.12.4 PLAIN - what is really happening inside

1. Where does a whole-sentence vector come from?
2. Run the text through a model and you get one vector per token. You need one vector for the whole thing.
3. Three common ways to collapse them.
4. **Mean pooling**: average all the token vectors. Simple, works well, still the most common method.
5. **CLS pooling**: use the vector at a special marker token placed at the start, which was trained to summarize the sequence.
6. **Last-token pooling**: in a decoder-only model, use the vector at the final position, which has seen everything before it.
7. But a raw language model's vectors are not well suited to comparison straight out of the box.
8. So embedding models are trained specially, with a method called contrastive training.
9. You show the model many pairs. Some pairs mean the same thing, some do not.
10. You push the matching pairs' vectors together and the non-matching pairs' vectors apart.
11. After enough pairs, cosine similarity in that space genuinely tracks meaning similarity, which it does not automatically do otherwise.
12. Finally, vectors are usually normalized to length 1, which makes cosine similarity and dot product the same computation, and makes the search faster.

47.12.5 TECHNICAL - the engineer's version

1. Embedding models are separate, smaller models from the generative ones, and they are usually encoder-style rather than decoder-style.
2. Typical output dimensions and what they cost to store, per million documents:

Dimensions	Bytes at fp32	Per 1M docs
384	1,536	1.5 GB
768	3,072	3.1 GB
1,024	4,096	4.1 GB
3,072	12,288	12.3 GB

3. Sentence-BERT, by Nils Reimers and Iryna Gurevych, EMNLP 2019, is the paper that made sentence embeddings practical, by fine-tuning BERT with a siamese network so that cosine similarity between sentence vectors is meaningful. Before it, comparing raw BERT sentence vectors performed worse than averaging GloVe vectors.
4. Contrastive objectives in common use: InfoNCE, multiple-negatives ranking loss, and in-batch negatives, where every other item in the batch is treated as a negative example for free.
5. Matryoshka representation learning, from a 2022 paper by Aditya Kusupati and colleagues, trains a model so that the first 256 or 512 numbers of a 3,072- number vector are usable on their own. Several current embedding APIs expose this as a dimension parameter, letting you trade accuracy for storage without re-embedding.
6. Nearest-neighbour search at scale uses an approximate index rather than brute force. HNSW, Hierarchical Navigable Small World graphs, from Yury Malkov and Dmitry Yashunin in 2016, is the most widely deployed. IVF with product quantization, from the FAISS library released by Facebook AI Research in 2017, is the other common family.
7. Approximate means approximate: these indexes return the true nearest neighbours most of the time, not always, and the recall is a tunable parameter traded against speed. That is a real correctness consideration and is often glossed over.

8. Vector databases and vector indexes in wide use as of 2026 include FAISS, pgvector for PostgreSQL, Qdrant, Weaviate, Milvus, Chroma and the vector features of Elasticsearch and OpenSearch.
9. Retrieval-augmented generation, named in a 2020 paper by Patrick Lewis and colleagues at Facebook AI Research, is embedding search plus prompt construction: embed the query, retrieve the top k chunks by cosine, paste them into the context, and let the generative model answer from them. Chapter 51 covers the whole pipeline including chunking and reranking.
10. Honest limitation: pure vector search is worse than keyword search at exact matching, such as product codes, error numbers and names. Production systems almost always combine both, which is called hybrid search, usually merged with reciprocal rank fusion.
11. A second honest limitation: an embedding has no notion of truth or recency. Two documents that contradict each other on a fact will sit close together, because they are about the same thing.

47.12.6 WORDS - remember these

1. Embedding model — a model whose output is a vector, not text — usually an encoder trained with a contrastive objective for similarity.
2. Pooling — turning many token vectors into one — mean, CLS or last-token aggregation over the sequence.
3. Contrastive training — pull matches together, push non-matches apart — an objective such as InfoNCE using in-batch negatives.
4. Semantic search — search by meaning, not words — ranking documents by cosine similarity between query and document embeddings.
5. Approximate nearest neighbour — fast, nearly-correct nearest search — index structures such as HNSW or IVF-PQ that trade recall for speed.
6. Hybrid search — combining keyword and vector search — merging BM25 and vector rankings, commonly with reciprocal rank fusion.
7. Normalization — scaling every vector to length 1 — makes dot product equal cosine similarity and speeds up search.

47.13 The context window

47.13.1 PLAIN - in simple words

1. The **context window** is the maximum number of tokens the model can be given at once, including everything: system instructions, history and your question.
2. It is measured in tokens, not words and not characters.
3. It is a hard limit set by the model, not a preference.
4. Why is there a limit at all? Because of how attention works.
5. Attention lets every token look at every other token.
6. With 100 tokens that is 10,000 pairs. With 1,000 tokens, a million pairs.
7. Double the length and the number of pairs goes up four times, not two.
8. So cost grows with the square of the length. Chapter 48 works through why.
9. When you exceed the window, one of two things happens.
10. Either the service refuses the request with an error, or something silently gets cut off, usually the oldest part of the conversation.
11. Silent cutting is the dangerous one, because the model then answers confidently based on text it never received.
12. And there is a further catch. A model that accepts a long input does not necessarily use all of it well.

47.13.2 PLAIN - a picture in your head

1. Think of a desk with a fixed area.
2. You can spread out papers on it, and you can read anything on the desk instantly.

3. But when the desk is full and you add another page, one falls off the edge.
4. Nobody tells you which one fell off, or that anything fell at all.
5. A bigger desk helps. It also costs more, and it takes longer to scan.
6. Now the awkward part. Even with a huge desk, if you spread out four hundred pages, you will reliably notice the first few and the last few and be much vaguer about the middle.
7. That is not a memory limit. It is an attention limit, and models have a measurable version of the same thing.

Where this comparison breaks: a desk keeps papers between work sessions. A context window does not persist at all. When the request ends, everything on the desk is swept away, and the next request starts empty. Anything that appears to persist was resent by the application.

47.13.3 PLAIN - a worked example

1. Work out what a window actually holds.
2. At about 4 characters per token, a 128,000-token window holds roughly 512,000 characters of English.
3. A typical printed page is about 2,000 characters, so that is about 256 pages.
4. But you never get all of it. Reserve space for the answer, and for the system prompt.
5. Now the attention arithmetic, which is why longer costs so much more:

Tokens	Pairs (n squared)	Relative to 512
512	262,144	1x
2,048	4,194,304	16x
8,192	67,108,864	256x
32,768	1,073,741,824	4,096x
131,072	17,179,869,184	65,536x
1,048,576	1,099,511,627,776	4,194,304x

6. Going from 512 to 1,048,576 tokens is 2,048 times the length and about four million times the attention pair count.
7. Memory grows too, but linearly, through the key-value cache: 40 GiB for a 70B model at 128,000 tokens at 16-bit, as computed in section 47.4.
8. So a million-token context is not a small engineering step from a thousand-token one. It requires different algorithms, not just more memory.

47.13.4 PLAIN - what is really happening inside

1. What happens when the input is too long? Four common strategies.
2. **Reject**: return an error. Honest, and the only one that cannot silently mislead.
3. **Truncate**: cut the input to fit. Usually the oldest turns go first. Cheap, and information is genuinely lost.
4. **Sliding window**: keep a moving window of the most recent tokens and drop what falls out of the back. Used in streaming settings.
5. **Summarize and compact**: replace old turns with a shorter summary generated by the model, keeping recent turns verbatim. This is what most long-running assistants do.
6. Each strategy loses something, and you should know which one your framework is using, because they fail differently.
7. Now the harder problem. Suppose the whole document does fit.
8. The model still does not attend evenly across it.
9. Researchers put a single fact somewhere in a long context and asked a question that requires it, varying only the position.

10. Accuracy was highest when the fact was at the very beginning or the very end, and lowest when it was in the middle.
11. The shape of the curve is a U. The finding is called "lost in the middle".
12. So placing your most important instruction in the middle of a long prompt is a measurably bad idea.

47.13.5 TECHNICAL - the engineer's version

1. Context windows have grown by more than three orders of magnitude in seven years:

Model and year	Context window
GPT-2, 2019	1,024
GPT-3, 2020	2,048
Llama 2, 2023	4,096
Llama 3.1, 2024	131,072

2. As of 13 August 2026, current commercial figures: Claude Opus 5, Claude Sonnet 5 and Claude Fable 5 accept 1,000,000 input tokens with up to 128,000 output tokens; Claude Haiku 4.5 accepts 200,000. The GPT-5.6 family accepts about 1.05 million input tokens, which is 2^20, with 128,000 output. These figures change often; check the provider's model page.
3. The cost of attention is O(n^2) in time for the score matrix and O(n) in memory for the key-value cache. FlashAttention, from Tri Dao and colleagues in 2022, does not change the O(n^2) compute but avoids materializing the n x n matrix in high-bandwidth memory, which removes the O(n^2) memory term and is what made long contexts practical. Chapter 48 covers the mechanism.
4. Position encoding is the second limit. RoPE, rotary position embedding from Jianlin Su and colleagues in 2021, encodes position by rotating query and key vectors. Extending a trained model beyond its training length needs the rotation frequencies adjusted, by methods such as position interpolation or the NTK-aware and YaRN scalings. A model advertised at 128,000 tokens is often a model trained at 8,000 and extended.
5. **Lost in the middle**: Nelson Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni and Percy Liang, arXiv July 2023, published in TACL 2024. Across multi-document question answering and key-value retrieval, performance was highest when the relevant information was at the start or end of the input and degraded significantly in the middle, and extended-context models were not reliably better than their shorter counterparts at using the extra length.
6. **RULER**, from Cheng-Ping Hsieh and colleagues at NVIDIA, April 2024, extended the needle-in-a-haystack test with multi-hop tracing and aggregation tasks and evaluated 17 long-context models. Its headline finding: although all claimed 32,000 tokens or more, only about half maintained satisfactory performance at 32,000, despite near-perfect scores on the simple retrieval test.
7. So state the distinction plainly. **Claimed context** is the number the API will accept without an error. **Effective context** is the length at which the model still performs near its short-context accuracy. The second number is smaller, is model-specific, and is rarely published by vendors.
8. The needle-in-a-haystack test, popularized by Greg Kamradt in November 2023, is a weak measure on its own: passing it shows the model can find one exact verbatim string, which is much easier than reasoning over dispersed information.
9. Practical rules that follow: put critical instructions at the beginning and repeat them at the end; retrieve fewer, better chunks rather than more; measure your own task at your own lengths rather than trusting the advertised number; and log token counts so you notice when truncation starts.
10. Tools: every API returns input token counts in its usage object; Anthropic and OpenAI both expose a token counting endpoint or library so you can check before sending; `tiktoken` and `AutoTokenizer` let you count offline.

47.13.6 WORDS - remember these

1. Context window — the most tokens the model accepts at once — the maximum input sequence length, a fixed architectural and training limit.
2. Truncation — cutting input to make it fit — dropping tokens, usually oldest first, often silently.
3. Sliding window — a moving view of recent tokens — keeping the last n tokens and discarding what falls out.
4. Lost in the middle — the dip in accuracy for content in the centre — the U-shaped position-versus-accuracy curve documented by Liu and colleagues.
5. Effective context — how much of the window is actually usable — the length at which task accuracy still holds up, usually shorter than the claimed window.
6. RoPE — how position is encoded in modern models — rotary position embedding, rotating query and key vectors by an angle proportional to position.
7. Needle in a haystack — the simple long-context test — planting one verbatim fact and asking for it back; necessary but far from sufficient.

47.14 Putting it together: one prompt from text to numbers

47.14.1 PLAIN - in simple words

1. Everything in this chapter meets in one short journey.
2. You type a sentence. It is characters.
3. The tokenizer chops it into pieces and looks each piece up, giving integers.
4. Each integer picks one row out of the embedding table, giving a list of numbers.
5. Those lists are stacked into a block, one row per token.
6. The block goes into layer 1, comes out changed, goes into layer 2, and so on.
7. Each layer transforms the block using its own frozen parameters.
8. At the end, the row belonging to the last token is turned into one score for every possible next token.
9. The scores are turned into probabilities, one is picked, and that becomes the next token.
10. Then the whole thing repeats with that token added on the end.
11. Nothing in the model changed at any point. Only the numbers flowing through.

47.14.2 PLAIN - a picture in your head

1. Picture a factory line with 32 stations.
2. At the entrance, a machine cuts the incoming text into standard-sized parts and stamps each with a part number.
3. A stores clerk takes each part number and fetches the matching tray of 4,096 components.
4. The trays travel down the line. Every station reworks all the trays a little, using its own fixed jigs and tools.
5. At the last station, the final tray is compared against a catalogue of 128,256 possible products, and the closest matches are scored.
6. One is chosen and shipped. Then the whole line runs again with that product added to the input.

Where this comparison breaks: on a factory line each station works on one item at a time. In a transformer every station works on all positions at once, and crucially each position can look at the others, which no factory station does.

47.14.3 PLAIN - a worked example

1. The prompt: "The cat sat on the mat." Model: an 8B-class model with hidden size 4,096, 32 layers, vocabulary 128,256.

```
stage 0  raw text
         "The cat sat on the mat."     23 characters
```

```
stage 1  tokenize (cl100k_base)
         7 tokens
         791, 8415, 7731, 389, 279, 5634, 13
         "The" " cat" " sat" " on" " the" " mat" "."
         shape: [7]

stage 2  embedding lookup
         fetch rows 791, 8415, ... from a
         [128256, 4096] table
         shape: [7, 4096]   = 28,672 numbers

stage 3  through layer 1 ... layer 32
         shape stays [7, 4096] at every layer boundary
         inside a layer it widens to [7, 14336] in the
         feed-forward middle and comes back

stage 4  final normalization
         shape: [7, 4096]

stage 5  take the last row only, position 6
         shape: [4096]

stage 6  multiply by the output matrix [128256, 4096]
         shape: [128256]      one logit per token

stage 7  softmax -> probabilities, then sample
         one token id comes out, say 13 for "."

stage 8  append and repeat from stage 2 with 8 tokens
```

2. Real sizes at each stage, at 16-bit: stage 2 holds 28,672 numbers, which is 56 KiB. Stage 6 produces 128,256 numbers, which is 251 KiB.
3. The key-value cache after this prompt holds 7 tokens x 128 KiB = 896 KiB.
4. Every one of those numbers is an activation. Every number in the tables they passed through is a parameter.

47.14.4 PLAIN - what is really happening inside

1. Notice the two places where the vocabulary size appears.
2. Once at the entrance, as a table you index into. Once at the exit, as a table you multiply against.
3. Those two are the same shape and are sometimes literally the same numbers.
4. Notice that the shape [7, 4096] is stable through the whole stack.
5. That stable shape is called the residual stream. Every layer reads from it and writes back into it by adding.
6. Notice that only the last row is used to predict the next token, during generation. The other rows were computed anyway, and their keys and values are cached so they do not need recomputing next time.
7. Notice what the second pass costs. It processes one new token, not eight. All previous work is in the cache.
8. That is why the first token of a reply is slower than the rest: the first one pays for the whole prompt, and each later one pays for a single token.

47.14.5 TECHNICAL - the engineer's version

1. The shapes with a batch dimension, which is how it really runs:

Stage	Tensor shape	dtype
Token ids	[B, S]	int64
After embedding	[B, S, 4096]	bf16
FFN intermediate	[B, S, 14336]	bf16
Logits	[B, S, 128256]	fp32

2. Logits are usually computed in fp32 even in a bf16 model, because softmax over 128,256 entries is numerically sensitive.
3. During prefill, S is the full prompt length and all positions are computed. During decode, S is 1 and only the new position is computed, with the cache supplying the rest.
4. Per-token generation cost in floating point operations is approximately 2 x N, where N is the parameter count, ignoring attention over the cache. For an 8B model that is about 16 billion operations per token.
5. Prompt processing cost is approximately 2 x N x S, so a 6,150-token prompt on an 8B model is about 98 trillion operations, which is why prefill is compute-bound and decode is bandwidth-bound.
6. Memory traffic per decoded token at batch 1 is roughly the entire weight file, because every weight is read once. At 4-bit that is about 4.5 GiB per token for an 8B model, so a machine with 100 GB/s of memory bandwidth caps out near 21 tokens per second regardless of how fast its arithmetic is.
7. That single fact explains most local inference performance questions, and it is why quantization speeds up generation as well as shrinking it.
8. Inspecting the journey in practice: `tokenizer(text, return_tensors="pt")` gives stage 1; `model.get_input_embeddings()(ids)` gives stage 2; `model(ids, output_hidden_states=True).hidden_states` gives every stage 3 boundary as a tuple of 33 tensors for a 32-layer model.

47.14.6 WORDS - remember these

1. Residual stream — the stable block of numbers every layer reads and writes — the [B, S, d] tensor threaded through the network by residual connections.
2. Prefill — processing the prompt — one parallel forward pass over all input positions, filling the key-value cache.
3. Decode — producing one token — a forward pass with sequence length 1 reusing the cache.
4. Logit — a raw score for one possible next token — one entry of the [V] output vector before softmax.
5. Softmax — turning scores into probabilities — exponentiate and divide by the sum, giving a distribution over the vocabulary.
6. Time to first token — how long before the reply starts — the latency of prefill, growing with prompt length.
7. Memory bandwidth bound — limited by moving numbers, not by computing — the regime single-stream decoding runs in, where tokens per second is set by bytes read per token.

47.98 Common wrong ideas

1. Wrong: more parameters means a smarter model. Right: parameter count sets a ceiling and predicts memory and speed very well, but Chinchilla at 70B beat Megatron-Turing NLG at 530B, and Mistral 7B beat Llama 2 13B. Data quantity, data quality, training compute and post-training all matter as much or more.
2. Wrong: a token is a word. Right: a token is a subword piece. English averages about 1.1 to 1.3 tokens per word, JSON averages over 3, and a Hindi sentence can cost 24 tokens where the English equivalent costs 5.
3. Wrong: the context window is the model's memory, and a model that accepts 1,000,000 tokens uses all of them equally. Right: it is the input size limit for one request, nothing persists between requests, any apparent memory is your application resending the history, and accuracy dips for content in the

middle, with RULER finding only about half of models claiming 32,000 tokens performed well at that length.

4. Wrong: quantization makes a model dumber in proportion to the bits removed. Right: the curve is very flat then steep. Going 16-bit to 8-bit cost 0.01 perplexity on a measured Llama-3.1-8B test; 16-bit to Q4_K_M cost 0.24; going to Q3_K_S cost 1.64. The first two thirds of the saving is nearly free.
5. Wrong: embeddings understand meaning. Right: they encode statistical co-occurrence as position in a space. That is enough for similarity to be useful and is not understanding. Two contradictory documents about the same fact sit close together, because they are about the same thing.
6. Wrong: hyperparameters are just small parameters. Right: hyperparameters are chosen by a human before training and are never learned. Parameters are learned by gradient descent and are never chosen.
7. Wrong: a 7B model needs 14 GB to train. Right: 14 GB is inference at 16-bit. Full training with Adam needs about 18 bytes per parameter for states alone, which is 117 GiB for 7B, before activations.
8. Wrong: king minus man plus woman equals queen proves embeddings capture relationships cleanly. Right: it is a real but weak effect, and the standard evaluation forbids returning the input words. With that exclusion removed, 3CosAdd accuracy fell from 71 per cent to 21 per cent.
9. Wrong: the model cannot count letters because it is stupid. Right: it receives “strawberry” as three integer ids and never sees individual letters. It answers from indirect knowledge, which is why it succeeds sometimes and fails unpredictably.
10. Wrong: input and output tokens cost the same. Right: output is typically priced four to six times higher, because generating is sequential and bandwidth-bound while reading the prompt is parallel and compute-bound.

47.99 Chapter summary in 20 lines

1. A parameter is one learned number sitting in one position of one matrix, set during training and frozen thereafter; “weight” and “parameter” are used interchangeably, with biases being additive parameters rather than weights.
2. A model’s parameter count is the sum of the sizes of all its matrices, and it is computable by hand from five published shape numbers.
3. Worked exactly, Llama 3.1 8B holds 8,030,261,248 parameters: 32 layers of 218,112,000 plus two 525,336,576 vocabulary tables plus a final norm.
4. The billions live in the feed-forward blocks: 70 per cent for 8B, 80 per cent for 70B, 81 per cent for 405B, with attention near 17 per cent throughout.
5. Hyperparameters are pre-training human choices, activations are temporary numbers created while running, context is the input text, tokens are the pieces it is chopped into. Only parameters are in the model file.
6. Memory equals parameter count times bytes per parameter: 7B is 26.1 GiB at fp32, 13.0 at fp16, 6.5 at int8 and 3.3 at int4.
7. Inference needs more than the weights: a key-value cache of 320 KiB per token for a 70B model, plus activations and framework overhead.
8. Training needs about 18 bytes per parameter for weights, master copy, gradients and Adam’s two moments, so a 7B model needs about 117 GiB of state before activations are counted.
9. Quantization stores each weight in fewer bits using a shared scale and optional zero point, per tensor, per channel or per group of 32 to 256.
10. GPTQ, AWQ, NF4 and the GGUF k-quants are the common formats; Q4_K is exactly 4.5 bits per weight, and Q4_K_M averages about 4.8 because sensitive tensors are kept at Q6_K.
11. A 70B model at Q4_K_M needs about 39.7 GiB of weights and fits a 64 GiB machine at a modest context; at fp16 it needs 131.4 GiB and does not.
12. Capability does not scale linearly with parameters: Chinchilla in 2022 showed models were undertrained, recommended about 20 tokens per parameter, and a 70B model beat a 530B one at equal compute.

13. Text is tokenized into subwords because whole words give an unbounded vocabulary and characters give sequences that are far too long.
14. Byte-pair encoding repeatedly merges the most frequent adjacent pair; on a four-word corpus, ten merges produce a 21-token vocabulary in which “newest” is a single token and “est_” emerges as a suffix without anyone naming it.
15. WordPiece merges by likelihood gain, Unigram prunes down from a large candidate set, and SentencePiece is the framework that makes both work without language-specific pre-tokenization.
16. English runs about 4 characters or 1.1 to 1.3 tokens per word; JSON is far worse; and a Hindi sentence costing 5 tokens in English cost 24 on the GPT-4-era tokenizer and 6 on the later one, which is real, measurable cost and context inequality.
17. Models cannot reliably count letters, reverse words or do long arithmetic because they never see letters or aligned digits, only token ids.
18. An embedding turns a token id into a row of a [V, d] table; the numbers mean nothing individually and everything collectively, and 4,096 dimensions cannot be visualized, with distance concentration and near-orthogonality breaking most 2D intuitions.
19. Similarity is measured by cosine of the angle; the king minus man plus woman result is real but much weaker than told, since removing the input-word exclusion dropped 3CosAdd accuracy from 71 to 21 per cent.
20. The context window is limited because attention cost grows with the square of length; exceeding it truncates or errors; and accepting a long context is not the same as using it well, as the lost-in-the-middle and RULER results show.

48

CHAPTER

KB

The Transformer

48.0 What this chapter gives you

1. You will be able to say what came before the transformer, why it worked, and the exact reason the field abandoned it in about eighteen months.
2. You will be able to explain attention in ordinary words to somebody who has never seen a formula, using a sentence whose meaning flips on one adjective.
3. You will be able to write the scaled dot-product attention formula from memory and say what every symbol in it does.
4. You will be able to compute a full attention pass by hand on a four-token sentence and get numbers that match a real implementation.
5. You will be able to explain the causal mask, draw it as a matrix, and say what changes when a model does not use one.
6. You will be able to explain multi-head attention, work out the tensor shapes for a real model configuration, and say honestly what heads do and do not do.
7. You will be able to explain why attention has no sense of order at all, and describe sinusoidal, learned, ALiBi and rotary position encoding.
8. You will be able to draw a whole transformer block from memory, with residual connections and normalization in the right places, and say why.
9. You will be able to trace one prompt through a real model as a series of tensor shapes, from token ids to a probability over the vocabulary.
10. You will be able to explain the quadratic cost of attention, compute a real key-value cache size, and name the fixes that are in production today.

48.1 What came before, and why it was not enough

48.1.1 PLAIN – in simple words

1. Before 2017, the standard way to handle a sentence was to read it one word at a time, left to right, the way a person reads aloud.
2. The machine that did this is called a **recurrent neural network**, usually shortened to RNN. Recurrent just means it feeds its own output back into itself.
3. It kept a single running summary of everything it had read so far. That summary is called the **hidden state**. It is a list of numbers.
4. Read word one, update the summary. Read word two, update the summary again. Keep going to the end.
5. That is the whole design. One summary, updated once per word, in order.
6. It worked. It was the best thing anybody had for sequences for about twenty-five years.
7. It had two problems, and the second one killed it.

8. Problem one: information from early words faded. By word two hundred, word three had almost no influence left on the summary.
9. Problem two, and this is the one that mattered: you cannot do the work for word ten until you have finished the work for word nine.
10. That is a rule you cannot break, because word ten literally needs the summary that word nine produced.
11. A modern graphics chip has tens of thousands of small calculators that all want work at the same time.
12. A design that insists on doing one word, then the next, then the next, cannot feed those calculators. They sit idle.
13. So training was slow, and it stayed slow no matter how much hardware you bought.
14. The transformer's central idea is to throw away the left-to-right rule and let every word look at every other word at once.
15. Once you do that, all the words can be processed in parallel, and the machine finally has enough work to do.

48.1.2 PLAIN - a picture in your head

1. Imagine you are told a long story down a telephone line, one word at a time, and you are not allowed to write anything down.
2. You are allowed to keep exactly one small notepad in your head with, say, twenty facts on it. That is the hidden state.
3. Every new word arrives and you must decide what to erase to make room.
4. At word ten you still remember the beginning clearly.
5. At word two hundred the beginning has been overwritten many times. It is not gone in principle, but almost nothing of it survives.
6. Now imagine the person telling the story is speaking as fast as you can process. You cannot skip ahead. You cannot ask a friend to handle the second half while you do the first.
7. The story is strictly serial. That is the cost.
8. The transformer is the opposite arrangement. The whole story is printed on a sheet of paper in front of you, all at once, and you may look at any word whenever you want.

Where this comparison breaks: a real hidden state is not a list of twenty readable facts. It is a vector of a few hundred to a few thousand numbers with no individually meaningful entries. And nothing "decides what to erase" in a deliberate way. A learned matrix multiplication and a gate value between 0 and 1 produce that effect, and the effect looks like forgetting from outside.

48.1.3 PLAIN - a worked example

1. Here is why the fading happens, with real arithmetic.
2. In training, the correction signal for word 3 has to travel backwards from word 200 through 197 update steps.
3. At each step it gets multiplied by a number. Call that number the shrink factor. If it is less than 1, the signal shrinks.
4. Here is what repeated multiplication does:

Shrink factor	After 10 steps	After 100 steps
0.5	0.00098	7.9e-31
0.8	0.107	2.0e-10
0.9	0.349	2.7e-05
0.99	0.904	0.366

5. At a shrink factor of 0.9, after 100 steps the signal is 0.0000266 of its original size. It has effectively vanished.

6. That is the **vanishing gradient problem**, and over a sequence it is vanishing over time steps rather than over layers.
7. Now the opposite. If the factor is 1.1 instead of 0.9, after 100 steps the signal is 13,780 times bigger. Training explodes and produces nonsense.
8. So you need the factor to sit very near 1.0 for hundreds of steps in a row, which is a very hard thing to arrange by accident.
9. Now the parallel problem, also with real arithmetic.
10. Take a sequence of 1,000 tokens.
11. An RNN needs 1,000 steps that must happen one after another. Nothing about the sequence can be done out of order.
12. A transformer needs 1 step for the whole sequence, because the operation is one big matrix multiplication over all 1,000 positions at once.
13. That is not a small speed-up. It is the difference between a chip running at a few per cent of its capacity and a chip running near its limit.

48.1.4 PLAIN - what is really happening inside

1. The RNN update is one line: new summary equals a function of the old summary and the new word.
2. Written plainly: h_t = tanh(W_h h_{t-1} + W_x x_t + b).
3. h_t is the summary after word t. x_t is the word. W_h and W_x are learned matrices. tanh squashes everything into the range -1 to +1.
4. Notice that h_{t-1} appears on the right-hand side. That is the whole dependency chain, and it is unbreakable.
5. To learn, you unroll this into a chain 200 links long and push the error backwards along it, which is the repeated multiplication just shown.
6. The **LSTM**, invented by Sepp Hochreiter and Jürgen Schmidhuber in 1997, was the fix for the fading.
7. It added a second track, called the cell state, that flows forward with addition rather than repeated multiplication.
8. Addition does not shrink a signal, so the correction can travel a long way without dying.
9. It also added **gates**: small learned circuits that output a number between 0 and 1 for each position, deciding how much to keep, how much to let in and how much to output.
10. The **GRU**, from Kyunghyun Cho and colleagues in 2014, is a simpler version with two gates instead of three, and usually performs about the same.
11. These fixes worked well enough for real products. Google Translate switched to an eight-layer LSTM system in September 2016.
12. But gates fix the fading. They do not fix the serial rule. Step t still needs step t-1. That part is structural.
13. Meanwhile a second problem appeared in translation. The standard design was an **encoder-decoder**: one RNN reads the whole source sentence into a single vector, and a second RNN writes the translation out of it.
14. Everything in a fifty-word sentence had to fit through one fixed-size vector. That vector is a bottleneck, and it is the same size whether the sentence has five words or fifty.
15. In September 2014, Dzmitry Bahdanau, Kyunghyun Cho and Yoshua Bengio published a fix. Instead of one vector, keep all the encoder's per-word states, and let the decoder look back and pick out the relevant ones.
16. That "look back and weight what matters" step is attention. It was invented as an accessory bolted onto an RNN.
17. The 2017 transformer paper's move was to delete the RNN and keep only the accessory.

48.1.5 TECHNICAL - the engineer's version

1. The vanishing and exploding gradient problem in recurrent networks was identified by Sepp Hochreiter in his 1991 diploma thesis at the Technical University of Munich, and analysed formally by Yoshua Bengio, Patrice Simard and Paolo Frasconi in 1994.

2. The mechanism is the chain rule applied through time. The gradient of the loss at step T with respect to the state at step t contains the product of Jacobians from t to T.
3. If the largest singular value of the recurrent Jacobian is consistently below 1, that product decays geometrically. Above 1, it grows geometrically.
4. With tanh activation the derivative is at most 1.0 and typically well below it, so decay is the common case. With sigmoid gates the derivative peaks at 0.25.
5. Exploding gradients are cheaply fixed by gradient clipping, which rescales the gradient vector when its norm passes a threshold. Vanishing gradients are not fixable that way, because you cannot amplify information that is gone.
6. LSTM, from Hochreiter and Schmidhuber, Neural Computation volume 9 issue 8, 1997, introduced the constant error carousel: an additive cell state path whose local derivative is close to 1.
7. The forget gate was not in the original LSTM. Felix Gers, Jürgen Schmidhuber and Fred Cummins added it in 1999 and 2000, and the version with the forget gate is what everybody now means by "LSTM".
8. Parameter counts per unit differ: a vanilla RNN cell has one weight matrix pair, an LSTM has four, and a GRU has three.

Model	Year	Gates	Sequential steps
Elman RNN	1990	0	n
LSTM	1997	3	n
GRU	2014	2	n
Transformer	2017	none	1

9. Sequence-to-sequence learning with a fixed context vector was published by Ilya Sutskever, Oriol Vinyals and Quoc Le in September 2014, using a 4-layer LSTM with 1,000 units per layer and a 1,000-dimensional context vector.
10. Additive attention, also called Bahdanau attention, was published the same month by Bahdanau, Cho and Bengio in the paper "Neural Machine Translation by Jointly Learning to Align and Translate". It scores each encoder state with a small feed-forward network and softmaxes the scores.
11. Multiplicative attention, from Minh-Thang Luong, Hieu Pham and Christopher Manning in 2015, replaced that small network with a dot product. That is the direct ancestor of the 2017 formula.
12. Google Neural Machine Translation, deployed in September 2016, used 8 encoder and 8 decoder LSTM layers with attention, residual connections and model parallelism across 8 GPUs. It cut translation errors substantially and was still fundamentally serial.
13. The training-time argument is about arithmetic intensity. An RNN step at batch size B computes a matrix times a matrix of shape [B, d] by [d, d]. With B = 32 and d = 1024 that is 67 MFLOP, then a synchronization, then the next step.
14. A transformer layer at sequence length 1,000 computes [1000, d] by [d, d], which at d = 4096 is 33.6 GFLOP in a single call, with no dependency inside the call.
15. Modern accelerators need large matrix-matrix operations to reach peak. This is covered in Chapter 22, which explains why a GPU is a wide parallel machine rather than a fast serial one. Long thin operations leave most of the chip idle.
16. This is established fact and was stated openly in the abstract of the 2017 paper: the models are "more parallelizable and requiring significantly less time to train".

48.1.6 WORDS - remember these

1. Recurrent neural network — a network that reads one item at a time and keeps a running summary — a network with a cyclic dependency h_t = f(h_{t-1}, x_t), unrolled over time for training.
2. Hidden state — the running summary — the vector carried from one time step to the next, of size equal to the layer width.

3. Vanishing gradient — the correction signal dying out over distance — the geometric decay of a product of Jacobians whose singular values are below 1.
4. LSTM — a recurrent cell with gates that remembers longer — long short-term memory, with input, forget and output gates and an additive cell state.
5. GRU — a simpler gated cell — gated recurrent unit, with update and reset gates and no separate cell state.
6. Encoder-decoder — one network reads, another writes — a sequence-to-sequence architecture with a context representation passed between them.
7. Bottleneck vector — squeezing a whole sentence into one fixed list of numbers — the fixed-dimension context vector in pre-attention seq2seq.
8. Attention — looking back and weighting what matters — a learned weighted average over a set of value vectors, with weights from a scoring function.

48.2 Attention in plain language, before any mathematics

48.2.1 PLAIN - in simple words

1. Read this sentence carefully: “the trophy did not fit in the suitcase because it was too big”.
2. What does “it” mean here? The trophy.
3. You did not need a rule book. You worked it out.
4. Now read the same sentence with one word changed: “the trophy did not fit in the suitcase because it was too small”.
5. What does “it” mean now? The suitcase.
6. The sentence is identical except for the last word, and the meaning of “it” flipped completely.
7. Stop and notice what you just did. To understand the word “it”, you looked back at other words in the sentence.
8. You did not look back at all of them equally. “the”, “did” and “in” contributed almost nothing.
9. You looked hard at “trophy”, “suitcase”, “fit” and the final adjective.
10. Then you combined what you found and produced a meaning for “it”.
11. That is attention. Look back, weight some things more than others, combine.
12. The model does exactly this, for every word, at every layer, with the weights worked out by learned matrices rather than by a person.
13. There is one important extra bit. The model does this for every word at once, not just for the confusing ones.
14. So while it is deciding what “it” points to, it is simultaneously deciding what “big” applies to and what “fit” takes as its subject.
15. And it does not know in advance which words are the confusing ones. Every position gets the same treatment.

48.2.2 PLAIN - a picture in your head

1. Picture a meeting where a decision has to be made, and eight people are in the room.
2. You are the chairperson. You ask a question.
3. Everybody in the room has something to say. Some of it is highly relevant. Most of it is not.
4. You do not take a plain average of all eight opinions, because that would drown the useful answer in noise.
5. Instead you listen to each person, decide how much their answer bears on your question, and give each one a share of your final decision.
6. Person 3 gets 60 per cent of your attention because they clearly know. Person 7 gets 25 per cent. The other six split the remaining 15 per cent.
7. Your conclusion is a blend, weighted by relevance.

8. That is one attention operation. Your question is the query. Each person's relevance is a weight. Each person's opinion is a value.
9. Now the part that makes it a transformer: every person in the room is doing this at the same time, each with their own question, including you.

Where this comparison breaks: in a real meeting, relevance is judged by understanding. In attention, relevance is a dot product between two learned vectors, which is a geometric measure of alignment. It correlates with what we would call relevance because training pushed it that way, and it has no independent notion of what the question is about. Also, the weights always add up to exactly 1, so a position cannot decide that nothing in the sentence is relevant. It must spread its 1.0 somewhere, and it often dumps it on a token that carries no meaning, which is a real and measured effect.

48.2.3 PLAIN – a worked example

1. Take the sentence with "big", and consider the position holding "it".
2. Here is roughly how a trained model distributes attention from "it", based on what published attention maps for coreference look like. These are illustrative figures to show the shape, not a measurement of one named model.

```
Sentence: the trophy did not fit in the suitcase
          because it was too big

Attention from the position holding "it":

  trophy    ############################  0.41
  suitcase  ############                  0.18
  big       ##########                    0.15
  fit       ######                        0.09
  it        #####                         0.08
  the       ##                            0.03
  other     ####                          0.06
                                          -----
                                          1.00
```

3. Now swap "big" for "small". The same position, in the same model, shifts:

```
Sentence: the trophy did not fit in the suitcase
          because it was too small

  suitcase  ##########################    0.39
  trophy    ############                  0.17
  small     ###########                   0.16
  fit       ######                        0.10
  it        #####                         0.08
  the       ##                            0.03
  other     ####                          0.07
                                          -----
                                          1.00
```

4. One word changed at the end of the sentence, and the top weight moved from "trophy" to "suitcase".
5. This is the point. The weights are not fixed by the identity of the word "it". They are computed fresh from the actual contents of this sentence.
6. That is why the same word gets a different representation in different sentences, which is the thing older methods could not do.
7. Chapter 47 covered embeddings, which are the fixed starting vectors for tokens. The embedding for "it" is one single vector, the same every time.

8. Attention is what turns that one fixed vector into a different vector in every different context. That is the entire job.
9. This sentence pair is not invented for this book. It is a **Winograd schema**, a test format proposed by Hector Levesque, Ernest Davis and Leora Morgenstern in 2011, named after Terry Winograd, who used an example of this shape in 1972.
10. The point of the format is that no amount of word-frequency statistics solves it. You have to model the situation being described.

48.2.4 PLAIN - what is really happening inside

1. Every token position holds a vector of numbers. Call it the position's current understanding of itself.
2. Attention runs in three moves, and every position does all three.
3. Move one: each position emits a **query**, meaning "here is what I am looking for". It is a vector, computed from the position's own current vector.
4. Move two: each position also emits a **key**, meaning "here is what I am, if anybody is looking". Also a vector, also computed from the same input.
5. Move three: each position emits a **value**, meaning "here is what I will contribute if you pick me".
6. Now the matching. Position 10's query is compared against every key in the sentence, including its own.
7. The comparison is a dot product: multiply the two vectors element by element and add up. A big result means the two vectors point in a similar direction.
8. That gives one raw score per position. For a 12-token sentence, position 10 gets 12 scores.
9. Those 12 scores go through softmax, which turns any list of numbers into positive numbers that add to exactly 1.
10. Now they are weights. 0.41 on "trophy", 0.18 on "suitcase", and so on.
11. Finally, take every position's value vector, multiply it by that position's weight, and add them all up.
12. The result is position 10's new vector. It now contains a blend of the whole sentence, weighted by what position 10 was looking for.
13. That new vector replaces the old one, and the next layer does the whole thing again with different learned matrices.
14. Repeat 32 times for a typical 8-billion-parameter model, and information from anywhere can reach anywhere, by many different routes.

48.2.5 TECHNICAL - the engineer's version

1. The operation just described is **self-attention**: queries, keys and values are all derived from the same input sequence.
2. **Cross-attention** is the same operation where queries come from one sequence and keys and values from another. It is what connects the decoder to the encoder in a translation model.
3. Self-attention is a set operation, not a sequence operation. It is permutation-equivariant, which is proved numerically in section 48.7.
4. The pairwise score matrix has shape [n, n] for a sequence of length n. Every position produces one row.
5. The distance between two tokens does not appear anywhere in the computation. Position 1 and position 4,000 are exactly as easy to connect as position 1 and position 2. That is the structural difference from an RNN, where the path length between positions i and j is |i - j| update steps.
6. Attention weight maps are widely published and easy to produce with the `output_attentions=True` flag in Hugging Face transformers, or with the BertViz tool from Jesse Vig, 2019.
7. Two honest warnings about reading those maps.
8. **The honest version:** an attention weight is not an explanation. Sarthak Jain and Byron Wallace published "Attention is not Explanation" in 2019, showing that alternative attention distributions can produce identical predictions. Sarah Wiegreffe and Yuval Pinter replied with "Attention is not not Expla-

nation" the same year, arguing the test was too strict. The dispute is not settled. Attention weights show where information flowed, not why.

9. Second warning: a large fraction of attention mass in trained models lands on the first token or on punctuation. This is called an **attention sink**, and was analysed by Guangxuan Xiao and colleagues in 2023 in the StreamingLLM work. The softmax must sum to 1, so a head with nothing to say has to put its weight somewhere, and it learns to park it on a fixed harmless position.
10. Established fact: attention computes content-dependent weighted averages, and this is measurable. Active research: what individual heads compute and whether their weights can be read as explanations. Marketing claim: any statement that the model "focuses on" or "pays attention to" something in the human sense.

48.2.6 WORDS - remember these

1. Self-attention — every word looking at every word in the same sentence — attention where Q, K and V all derive from one input sequence.
2. Cross-attention — one sequence looking at a different sequence — attention where Q comes from the target and K, V from the source.
3. Query — what a position is looking for — a learned linear projection of the position's hidden state, of dimension d_k.
4. Key — what a position advertises about itself — a learned projection compared against queries by dot product, of dimension d_k.
5. Value — what a position contributes if selected — a learned projection that is averaged with the attention weights, of dimension d_v.
6. Softmax — turning any scores into shares that add to 1 — exp(x_i) divided by the sum of exp over all j, applied along the key axis.
7. Attention weight — how much one word listened to another — one entry of the row-stochastic [n, n] matrix produced by softmax over scaled scores.
8. Attention sink — a position that soaks up leftover weight — a token, typically the first, that receives large attention mass with no semantic role, arising from the sum-to-one constraint.
9. Winograd schema — a sentence pair whose meaning flips on one word — a coreference test designed to be unsolvable by surface statistics.

48.3 Query, key and value, taught slowly

48.3.1 PLAIN - in simple words

1. Three words do all the work in attention: query, key and value.
2. They sound abstract. They are not. They are three different views of the same token, made by three different learned recipes.
3. The **query** is the question this position is asking. "I am a pronoun. I need to know what noun I stand for."
4. The **key** is the label this position wears. "I am a singular concrete noun, and I am the subject of the sentence."
5. The **value** is the content this position offers up if selected. "Here is my actual meaning, ready to be copied into whoever picks me."
6. Every single position produces all three, every time.
7. So a position is simultaneously asking a question, advertising an answer, and holding content to hand over.
8. The matching happens between queries and keys only. Values take no part in deciding who wins.
9. Once the weights are decided, the values are what actually get blended and passed on.
10. That separation is the useful part of the design. What makes something findable is allowed to be different from what it contains.

11. And the crucial fact people miss: all three come from the same starting vector. There is no separate store of keys and values somewhere.
12. Take one token's vector, multiply it by matrix number 1 to get the query, matrix number 2 to get the key, matrix number 3 to get the value.
13. Those three matrices are learned during training and frozen afterwards. They are ordinary parameters, in the sense of Chapter 47.

48.3.2 PLAIN – a picture in your head

1. Picture a library, but a strange one where you may only ask one question and must get one answer back immediately.
2. You walk in with a question written on a card. That is your **query**.
3. Every book in the library has a small index card clipped to its spine. The card says what the book is about, in a compressed form. That is the **key**.
4. The book's actual pages are the **value**.
5. You do not read any books. You hold your question card next to every index card in turn and score how well they line up.
6. Book 3's card scores highly. Book 7's card scores moderately. Most cards score near zero.
7. Then you do something no real library allows. You do not pick one book. You take a weighted blend of every book's contents: 60 per cent of book 3, 25 per cent of book 7, and small slivers of the rest.
8. That blend is your answer.
9. The index card is deliberately not the book. A card can say "this is a cookbook about bread" while the book contains actual recipes.
10. That is the whole reason keys and values are separate. Findability and content are different jobs.

Where this comparison breaks: in a real library the index card was written by a librarian who read the book. Here both the card and the book come from the same raw vector through two different learned matrices, so they are not independent. Also, a library has a fixed catalogue. Here the keys are recomputed from scratch for every input, so the same word gets a different key in a different sentence. And a real search returns one book. Attention always returns a blend and can never return nothing, because the weights are forced to sum to 1.

48.3.3 PLAIN – a worked example

1. Now the projection, with numbers small enough to check on paper.
2. We will use a four-token sentence: "the cat was hungry".
3. Each token arrives as a vector of 4 numbers. In a real model this would be 4,096 numbers, from the embedding table of Chapter 47. Here it is 4.
4. Stack the four token vectors as rows. That is the input matrix X, shape [4 tokens, 4 dimensions]:

```
X (4 tokens x 4 dimensions)

           d0  d1  d2  d3
the         1   0   1   0
cat         0   1   0   1
was         1   1   0   0
hungry      0   1   1   0
```

5. **The honest version:** real embedding values are floating-point numbers like -0.0113 or 0.0392, never neat integers. Integers are used here only so you can verify every multiplication by hand.
6. Now the three learned matrices. Each is shape [4, 3], because we are choosing to project from 4 dimensions down to 3.

```
W_Q (4x3)        W_K (4x3)        W_V (4x3)
 1  0  0          1  0  0          2  0  1
```

```
1  1  2          1  0  2          0  2  0
0  1  0          1  1  1          1  1  2
1  1  0          0  1  2          0  1  1
```

7. To get the query for "hungry", take the row for "hungry", which is [0, 1, 1, 0], and multiply it by W_Q.
8. Doing it column by column:
9. Column 0: (0 x 1) + (1 x 1) + (1 x 0) + (0 x 1) = 1.
10. Column 1: (0 x 0) + (1 x 1) + (1 x 1) + (0 x 1) = 2.
11. Column 2: (0 x 0) + (1 x 2) + (1 x 0) + (0 x 0) = 2.
12. So the query for "hungry" is [1, 2, 2].
13. Do the same for every token and every matrix. The three results:

```
Q = X W_Q          K = X W_K          V = X W_V
the     1  1  0    the     2  1  1    the     3  1  3
cat     2  2  2    cat     1  1  4    cat     0  3  1
was     2  1  2    was     2  0  2    was     2  2  1
hungry  1  2  2    hungry  2  1  3    hungry  1  3  2
```

14. Check one more by hand. The key for "cat" is row [0, 1, 0, 1] times W_K.
15. Column 0: 0 + 1 + 0 + 0 = 1. Column 1: 0 + 0 + 0 + 1 = 1. Column 2: 0 + 2 + 0 + 2 = 4. So [1, 1, 4]. That matches the table.
16. Notice the shapes. X is [4, 4]. Each W is [4, 3]. Each of Q, K, V is [4, 3].
17. The number of tokens never changes. Only the width changes.

48.3.4 PLAIN - what is really happening inside

1. There is no lookup, no dictionary, no search index. There are three matrix multiplications.
2. A matrix multiplication of a [4, 4] by a [4, 3] is just 12 dot products, one per output cell, which is exactly what was done above by hand.
3. The words "query", "key" and "value" are borrowed from database retrieval. They are a naming choice, not a mechanism.
4. Nothing enforces that W_Q learns "questions" and W_K learns "labels". Those are the roles they end up playing because of where they sit in the formula.
5. W_Q and W_K only ever appear together, inside a dot product. Their product Q K-transpose is what the model actually uses.
6. That means the pair behaves as a single learned bilinear form. You could fold them into one matrix and get identical scores.
7. They are kept separate because it costs far fewer parameters. Two matrices of [d, d_k] cost 2 d d_k parameters, while one full [d, d] matrix costs d squared. With d = 4096 and d_k = 128 that is 1,048,576 against 16,777,216.
8. W_V is genuinely separate, because its output does not go into any score. It goes into the answer.
9. In code, all three projections for a whole batch are usually done as one matrix multiplication with the three weight matrices concatenated, then split afterwards. That is an implementation detail for speed, not part of the idea.
10. On the hardware side, these are exactly the dense matrix multiplications Chapter 22 described as the operation a GPU is built to do. The whole sequence goes through in one call.

48.3.5 TECHNICAL - the engineer's version

1. The projections are three linear maps with no bias in most modern models: Q = X W_Q, K = X W_K, V = X W_V.
2. Shapes for one attention layer, with input X of shape [B, n, d_model]:

Tensor	Shape	Notes
X	[B, n, d_model]	layer input
W_Q	[d_model, h x d_k]	learned
W_K	[d_model, h_kv x d_k]	learned
W_V	[d_model, h_kv x d_v]	learned

3. In the 2017 base model, d_model = 512, h = 8, d_k = d_v = 64, so each of the four projection matrices is 512 by 512 and holds 262,144 parameters.
4. In Llama 3.1 8B, d_model = 4096, 32 query heads, 8 key-value heads, head dimension 128. So q_proj is [4096, 4096], while k_proj and v_proj are [1024, 4096]. That asymmetry is grouped-query attention, covered in 48.13.
5. Bias terms: GPT-2 uses biases on these projections. The Llama family, Mistral and Qwen do not. This is an implementation detail with negligible effect on quality and a small effect on parameter count.
6. Real parameter names in a checkpoint, which you can list without loading the model using the safetensors header:

```
model.layers.0.self_attn.q_proj.weight   [4096, 4096]
model.layers.0.self_attn.k_proj.weight   [1024, 4096]
model.layers.0.self_attn.v_proj.weight   [1024, 4096]
model.layers.0.self_attn.o_proj.weight   [4096, 4096]
```

7. Note the convention: PyTorch stores a linear layer's weight as [out_features, in_features] and computes x times W-transpose. So the printed shape is the transpose of the mathematical W in the formula. This trips up almost everybody once.
8. The low-rank factorization argument is exact. Scores depend only on the product W_Q W_K-transpose, which has rank at most d_k. Choosing d_k = 128 with d_model = 4096 imposes a rank-128 constraint on the score matrix, and that constraint is the parameter saving.
9. Whether that rank limit hurts has been studied directly. Srinadh Bhojanapalli and colleagues at Google Research published "Low-Rank Bottleneck in Multi-head Attention Models" in 2020, showing that tying d_k to d_model divided by the number of heads does constrain what a head can express, and that decoupling the two can help.

48.3.6 WORDS - remember these

1. Projection — turning a vector into a different-shaped vector — multiplication by a learned matrix, usually without a bias term.
2. Linear layer — a matrix multiply, optionally plus a bias — `nn.Linear` in PyTorch, storing weight as [out_features, in_features].
3. d_model — the width of the main pipe through the model — the residual stream dimension, 512 in the 2017 base model, 4096 in Llama 3.1 8B.
4. d_k — the width of one head's queries and keys — the per-head key dimension, 64 in the 2017 paper, 128 in most current models.
5. Bilinear form — one matrix sandwiched between two vectors — the effective scoring matrix W_Q W_K-transpose, of rank at most d_k.
6. Low-rank factorization — writing a big matrix as two thin ones — replacing a [d, d] matrix with [d, r] times [r, d], costing 2dr instead of d squared.

48.4 Scaled dot-product attention, with the actual mathematics

48.4.1 PLAIN - in simple words

1. Here is the whole formula. It is one line, and it is the most quoted line in modern machine learning.

```
Attention(Q, K, V) = softmax( Q K^T / sqrt(d_k) ) V
```

2. Read it right to left in four steps.
3. Step one: Q K^T. Compare every query with every key by dot product. Output is a square grid of scores, one row per token, one column per token.
4. Step two: divide by the square root of d_k, the width of a key. This is a fixed number, not learned. It stops the scores from getting too large.
5. Step three: softmax each row. Each row of scores becomes a row of weights that are all positive and add up to exactly 1.
6. Step four: multiply by V. Each row of the output is a weighted average of all the value vectors, using that row's weights.
7. That is it. Four steps, three of which are matrix multiplications and one of which is a division.
8. The result has the same number of rows as the input. One output vector per token, exactly as many as went in.
9. Everything else in this chapter is either a variation on this line, a way to make it faster, or a way to fix something it cannot do on its own.

48.4.2 PLAIN - a picture in your head

1. Picture a school report card, but sideways.
2. Down the left are the students asking questions. Across the top are the students offering answers. Same list of students on both sides.
3. In each cell you write a number: how useful is that answer for that question. This grid is Q K-transpose.
4. Now the trick. Nobody trusts raw scores, so each row is converted into percentages that add to 100. That is softmax.
5. Row 4 might read: 3 per cent, 60 per cent, 3 per cent, 34 per cent. That is student 4's opinion of who to listen to.
6. Finally each student writes down their answer on a card. Student 4 makes a mixture: 3 per cent of card 1, 60 per cent of card 2, and so on.
7. What student 4 walks away with is a blend of the whole class, weighted by their own judgement of relevance.
8. And every student does this at the same time, so the entire grid is filled in one operation.

Where this comparison breaks: the "percentages" from softmax are never exactly zero. Every student always gets a nonzero share, even an irrelevant one, because the exponential function never reaches zero. And "how useful is that answer" is not a judgement. It is the cosine-like alignment of two vectors, scaled by their lengths. Two vectors can align strongly for reasons that have nothing to do with meaning.

48.4.3 PLAIN - a worked example

1. This is the full computation, using the Q, K and V produced in section 48.3 for the sentence "the cat was hungry". Every number below was computed and checked with numpy.
2. Recall the matrices:

```
Q                  K                  V
the     1  1  0    the     2  1  1    the     3  1  3
cat     2  2  2    cat     1  1  4    cat     0  3  1
was     2  1  2    was     2  0  2    was     2  2  1
hungry  1  2  2    hungry  2  1  3    hungry  1  3  2
```

3. Step one: Q K-transpose. Every entry is one dot product.
4. Row "hungry", column "cat": q_hungry is [1, 2, 2], k_cat is [1, 1, 4].

5. (1 x 1) + (2 x 1) + (2 x 4) = 1 + 2 + 8 = 11.
6. Row "hungry", column "the": [1, 2, 2] against [2, 1, 1] gives 2 + 2 + 2 = 6.
7. Row "hungry", column "was": [1, 2, 2] against [2, 0, 2] gives 2 + 0 + 4 = 6.
8. Row "hungry", column "hungry": [1, 2, 2] against [2, 1, 3] gives 2 + 2 + 6 = 10.
9. Doing all sixteen gives the raw score matrix:

```
S = Q K^T          the   cat   was  hungry
     the             3     2     2     3
     cat             8    12     8    12
     was             7    11     8    11
     hungry          6    11     6    10
```

10. Step two: divide by sqrt(d_k). Here d_k = 3, and sqrt(3) = 1.7320508.

```
S / sqrt(3)        the     cat     was  hungry
     the        1.7321  1.1547  1.1547  1.7321
     cat        4.6188  6.9282  4.6188  6.9282
     was        4.0415  6.3509  4.6188  6.3509
     hungry     3.4641  6.3509  3.4641  5.7735
```

11. Step three: softmax each row. Take the row for "hungry": [3.4641, 6.3509, 3.4641, 5.7735].
12. Exponentiate each: e to the 3.4641 = 31.9477, e to the 6.3509 = 572.9812, e to the 3.4641 = 31.9477, e to the 5.7735 = 321.6624.
13. Add them: 31.9477 + 572.9812 + 31.9477 + 321.6624 = 958.5392.
14. Divide each by the total: 31.9477 / 958.5392 = 0.0333, 572.9812 / 958.5392 = 0.5978, 31.9477 / 958.5392 = 0.0333, 321.6624 / 958.5392 = 0.3356.
15. Check: 0.0333 + 0.5978 + 0.0333 + 0.3356 = 1.0000. Good.
16. The full attention weight matrix, every row summing to 1:

```
A                  the     cat     was  hungry
     the        0.3202  0.1798  0.1798  0.3202
     cat        0.0452  0.4548  0.0452  0.4548
     was        0.0436  0.4393  0.0777  0.4393
     hungry     0.0333  0.5978  0.0333  0.3356
```

17. Read that matrix as language. "hungry" puts 0.5978 of its weight on "cat". Whatever "hungry" describes, it is mostly getting it from "cat".
18. "was" splits its weight between "cat" at 0.4393 and "hungry" at 0.4393, which is what a linking verb joining a subject to a predicate would do.
19. "the" is nearly flat, which is what a determiner with no strong question would do.
20. This is a toy with hand-chosen matrices. Do not conclude that this pattern is what a trained model produces. It shows the shape of the computation.
21. Step four: multiply A by V. Take the "hungry" row again:

```
0.0333 x [3, 1, 3]  =  [0.1000, 0.0333, 0.1000]   from "the"
0.5978 x [0, 3, 1]  =  [0.0000, 1.7933, 0.5978]   from "cat"
0.0333 x [2, 2, 1]  =  [0.0667, 0.0667, 0.0333]   from "was"
0.3356 x [1, 3, 2]  =  [0.3356, 1.0067, 0.6712]   from "hungry"
                       ------------------------
               sum =  [0.5022, 2.9000, 1.4022]
```

22. The complete output matrix, one row per token:

```
Output = A V          dim0    dim1    dim2
     the            1.6405  2.1798  1.9607
     cat            0.6807  2.8645  1.5452
```

```
was             0.7257  2.8350  1.5266
hungry          0.5022  2.9000  1.4022
```

23. Shape check: A is [4, 4], V is [4, 3], so the output is [4, 3]. Four tokens in, four tokens out, width 3.
24. Notice that "hungry" came in as [0, 1, 1, 0] and leaves as [0.5022, 2.9000, 1.4022]. It is no longer a context-free word. It now carries mostly "cat".

48.4.4 PLAIN - what is really happening inside

1. Now the piece everybody skips: why divide by the square root of d_k.
2. A dot product adds up d_k separate products. The more terms you add, the larger the total tends to be, in both directions.
3. If the entries of q and k are independent, have mean 0 and variance 1, then each product q_i k_i has mean 0 and variance 1.
4. Adding d_k independent things with variance 1 gives something with variance d_k, so the typical size grows like the square root of d_k.
5. Measured with 20,000 random pairs per dimension:

d_k	Variance of q.k	Std dev	sqrt(d_k)
4	4.00	2.00	2.00
64	63.54	7.97	8.00
512	505.98	22.49	22.63
4096	4120.58	64.19	64.00

6. The measured standard deviation tracks sqrt(d_k) exactly. That is the whole argument.
7. Now why large scores are bad. Softmax exponentiates. A gap of 10 between two scores becomes a ratio of about 22,026 to 1 after exponentiating.
8. Here is a real case with d_k = 64 and four random keys:

```
raw scores       2.153  -7.355  -9.911  -4.582
softmax(raw)     0.9987  0.0001  0.0000  0.0012

after dividing by sqrt(64) = 8
scaled scores    0.2691  -0.9194  -1.2389  -0.5728
softmax(scaled)  0.5110  0.1557  0.1131  0.2202
```

9. Without scaling, one key gets 99.87 per cent of the weight and the rest get essentially nothing. The attention has become a hard selection.
10. That is bad for two reasons. It throws away information, and worse, it kills learning.
11. Softmax's gradient for an output p is proportional to p times (1 - p).
12. At p = 0.5 that is 0.25. At p = 0.999 it is 0.000999. At p = 0.99999 it is 0.00001.
13. So once softmax saturates, the correction signal flowing back through it is almost zero, and the model stops being able to adjust those scores.
14. Dividing by sqrt(d_k) rescales the scores to variance about 1 regardless of head width, keeping softmax in its responsive range at the start of training.
15. The 2017 paper states this reason directly in a footnote, and it is the only place in the paper where the word "suspect" appears about a design choice.
16. One more subtlety. The scaling matters most at initialization, when weights really are near-random and the variance argument holds exactly. Later in training the weights are not random and the argument is approximate. It still works, and nobody has found a reason to change it.

48.4.5 TECHNICAL - the engineer's version

1. The canonical formula, from equation 1 of "Attention Is All You Need": Attention(Q, K, V) = softmax(Q K^T / sqrt(d_k)) V.
2. Shapes, single head, batch B, sequence n:

Step	Operation	Output shape
scores	Q times K^T	[B, n, n]
scaled	divide by sqrt(d_k)	[B, n, n]
weights	softmax over last dim	[B, n, n]
output	weights times V	[B, n, d_v]

3. Floating-point operations, forward pass, single head: 2 n^2 d_k for the score matrix and 2 n^2 d_v for the weighted sum, so 4 n^2 d in total when d_k = d_v = d.
4. Softmax is implemented in the numerically stable form: subtract the row maximum before exponentiating. Without that, exp of a score above about 88 overflows in float32 and above about 11 in float16.
5. That max-subtraction is mathematically a no-op, since softmax is invariant to adding a constant to all inputs. It is purely a numerical safety measure and it is universal.
6. Attention scores are almost always computed in float32 even when the model runs in bfloat16, because the softmax denominator accumulates n terms and bfloat16 has only 8 bits of mantissa.
7. In PyTorch 2.0 and later, the whole thing is one call: `torch.nn.functional.scaled_dot_product_attention(q, k, v, is_causal=True)`, which dispatches to a fused kernel such as FlashAttention when the shapes and dtypes allow it.
8. The reference implementation, which is worth being able to write from memory:

```
import torch, math

def attention(q, k, v, mask=None):
    # q: [B, h, n, dk]  k,v: [B, h, n, dk]
    dk = q.size(-1)
    scores = q @ k.transpose(-2, -1) / math.sqrt(dk)
    if mask is not None:
        scores = scores.masked_fill(mask, float("-inf"))
    w = torch.softmax(scores.float(), dim=-1).to(v.dtype)
    return w @ v
```

9. Alternatives to the dot product exist and lost. Additive attention (Bahdanau) uses a one-hidden-layer network and is theoretically stronger at large d_k but far slower, because it cannot be expressed as a single matrix multiply. The 2017 paper reports it performs comparably at small d_k.
10. Scaling variants exist. Some papers use 1/d_k, some use a learned temperature. The 1/sqrt(d_k) convention is universal in practice, and it is a convention, not a standard. Nothing enforces it.
11. Query-key normalization, which normalizes Q and K to unit length before the dot product, appeared around 2020 and is used in some large-scale training runs to prevent logit growth. It changes the scaling argument but not the structure.

48.4.6 WORDS - remember these

1. Dot product — multiply matching entries and add them up — the inner product of two vectors, equal to |a||b|cos(theta).
2. Score matrix — the grid of how well every query matched every key — the [n, n] tensor Q K^T before scaling and softmax.
3. Scaling factor — the fixed divisor that keeps scores small — 1/sqrt(d_k), chosen so the score variance is about 1 at initialization.

4. Saturation — when softmax has already picked a winner and stops responding — the regime where one probability approaches 1 and the Jacobian term p(1-p) approaches 0.
5. Numerically stable softmax — subtracting the biggest score first — the standard implementation exp(x - max) / sum exp(x - max), avoiding overflow.
6. Attention output — the blended result for each position — the [n, d_v] matrix of weighted sums of value vectors.
7. Additive attention — the older, slower scoring method — Bahdanau's v^T tanh(W_1 q + W_2 k), a small network instead of a dot product.

48.5 Masking: stopping the model reading ahead

48.5.1 PLAIN - in simple words

1. A language model is trained by hiding the next word and asking it to guess.
2. That only works if the model genuinely cannot see the next word.
3. But attention, as built so far, lets every position see every position, including the ones after it.
4. So during training on the sentence "the cat was hungry", the position for "was" can see "hungry", and predicting "hungry" becomes trivial copying.
5. The model would score perfectly during training and be useless at generation time, when there is nothing after the current word to copy.
6. The fix is called a **mask**. Before the softmax, you go into the score grid and destroy every score that looks forward.
7. Destroy means set to negative infinity, because e to the power of negative infinity is 0, so softmax gives that position a weight of exactly zero.
8. You cannot just set the weights to zero after the softmax, because then the remaining weights would not add to 1. Doing it before the softmax means the surviving weights renormalize automatically.
9. The shape of the destroyed region is the triangle above the diagonal, so this is called a **causal mask**, or a look-ahead mask.
10. Causal here means "the past can affect the future, and the future cannot affect the past", the ordinary meaning of cause.
11. There is a second, different kind of mask. When you run many sentences of different lengths together, you pad the short ones with filler tokens.
12. A **padding mask** blocks attention to that filler, so real words never average in meaningless positions.
13. And there are models that use no causal mask at all. They read the whole text both ways at once. Those are excellent at understanding a fixed piece of text and cannot generate.

48.5.2 PLAIN - a picture in your head

1. Imagine an exam where the questions and answers are printed on the same long scroll, in order.
2. Question 1, answer 1, question 2, answer 2, all the way down.
3. You are given the scroll rolled up, and a rule: you may unroll it only as far as the question you are currently answering, never further.
4. So when you are on question 3, you can see questions 1, 2 and 3 and answers 1 and 2, and nothing beyond.
5. That is exactly the causal mask. Position t sees positions 1 through t.
6. Now here is the part that surprises people. The examiner does not hand you one question at a time. The examiner hands the whole class the whole scroll at once, with each student allowed a different unrolling limit.
7. Student 1 may see 1 word. Student 2 may see 2 words. Student 500 may see 500.
8. All 500 students work at the same time, in one operation, and each is graded on a genuinely fair prediction.

9. That is why masked training is fast. One pass over a 4,000-token document produces 4,000 separate prediction problems, all solved in parallel.

Where this comparison breaks: the students are not separate. They are 500 rows of one matrix, and the "unrolling limit" is not enforced by discipline but by arithmetic that makes forbidden information contribute exactly zero. Also, a real exam has one right answer per question. Here the target is the actual next token in the training text, which is one sample from a distribution of acceptable continuations, and the model is graded against that one sample.

48.5.3 PLAIN – a worked example

1. Take the scaled score matrix computed in section 48.4 for "the cat was hungry", and apply a causal mask.
2. Before masking, the scaled scores were:

```
                the     cat     was  hungry
     the       1.7321  1.1547  1.1547  1.7321
     cat       4.6188  6.9282  4.6188  6.9282
     was       4.0415  6.3509  4.6188  6.3509
     hungry    3.4641  6.3509  3.4641  5.7735
```

3. Now set everything strictly above the diagonal to negative infinity:

```
                the     cat     was  hungry
     the       1.7321    -inf    -inf    -inf
     cat       4.6188  6.9282    -inf    -inf
     was       4.0415  6.3509  4.6188    -inf
     hungry    3.4641  6.3509  3.4641  5.7735
```

4. The pattern is a lower triangle of real numbers and an upper triangle of negative infinity. This is the causal mask, and it is always this shape.
5. Now softmax each row. Row "the" has exactly one surviving score, so it gets weight 1.0 and everything else 0.
6. Row "cat" has two surviving scores, 4.6188 and 6.9282. Their difference is 2.3094, so e to the 2.3094 = 10.07, and the weights are 1 / 11.07 = 0.0903 and 10.07 / 11.07 = 0.9097.
7. The full masked attention matrix:

```
A causal           the     cat     was  hungry
     the       1.0000  0.0000  0.0000  0.0000
     cat       0.0903  0.9097  0.0000  0.0000
     was       0.0778  0.7836  0.1386  0.0000
     hungry    0.0333  0.5978  0.0333  0.3356
```

8. Every row still sums to exactly 1. The zeros are true zeros, not small numbers.
9. Compare the "was" row with the unmasked version. Unmasked it was [0.0436, 0.4393, 0.0777, 0.4393]. Masked it is [0.0778, 0.7836, 0.1386, 0].
10. The 0.4393 that was going to "hungry" did not disappear. It was redistributed proportionally across the three legal positions. That is what the renormalization in softmax does for free.
11. Multiplying by V gives the masked output:

```
Output (causal)        dim0    dim1    dim2
     the             3.0000  1.0000  3.0000
     cat             0.2710  2.8193  1.1807
     was             0.5107  2.7057  1.1556
     hungry          0.5022  2.9000  1.4022
```

12. Note the first row. Position 1 can only see itself, so its output is exactly its own value vector [3, 1, 3]. That is always true of the first token in a causal model, in every layer.

13. Note the last row. It is identical to the unmasked last row, because the last position had nothing ahead of it to mask anyway.

48.5.4 PLAIN - what is really happening inside

1. In code, the mask is not really negative infinity. It is a large negative number, or a genuine negative infinity that the softmax handles.
2. In float32, a common choice is -1e9 or the smallest representable value. In float16 it must be about -65504 or smaller in magnitude than that, because float16 cannot hold -1e9.
3. Using a finite value works because e to the power of -1e9 underflows to exactly 0.0 in floating point. The result is identical to true zero.
4. **The honest version:** there is a real failure mode here. If an entire row is masked, every score becomes negative infinity, and softmax computes 0 divided by 0, giving NaN, which then poisons the whole model. Implementations guard against this. It happens in practice with badly constructed padding masks.
5. The mask is not a parameter. Nothing about it is learned. It is a fixed boolean pattern computed from the sequence length.
6. It costs no memory to store, in good implementations, because the triangular pattern is generated on the fly inside the kernel rather than materialized as an [n, n] tensor.
7. Now the padding mask. Suppose you batch three prompts of lengths 5, 9 and 3.
8. To make one rectangular tensor, you pad all three to length 9 with a filler token id.
9. Without a mask, position 2 of the third prompt would average in six filler positions, and its output would be contaminated.
10. The padding mask sets the columns of the padded positions to negative infinity for every row.
11. Padding masks and causal masks combine by taking whichever is more restrictive at each cell, which in boolean terms is a logical OR of the two blocked patterns.
12. A neat trick used by almost every training framework avoids padding entirely: concatenate many documents into one long stream, cut it into fixed-length chunks, and never pad at all. This is called sequence packing.

48.5.5 TECHNICAL - the engineer's version

1. The causal mask M is defined by $M[i][j] = 0$ if j is less than or equal to i, and negative infinity otherwise. It is added to the scaled scores before softmax.
2. The masked formula: softmax(Q K^T / sqrt(d_k) + M) V.
3. In PyTorch, `torch.triu(torch.ones(n, n), diagonal=1).bool()` produces the blocked pattern, and `scores.masked_fill(mask, float("-inf"))` applies it.
4. Since PyTorch 2.0, `scaled_dot_product_attention(..., is_causal=True)` is preferred, because fused kernels skip the masked blocks entirely rather than computing and discarding them, roughly halving the work.
5. That saving is real and worth naming: a causal attention over n tokens does about n^2/2 useful score computations, not n^2.
6. Encoder models use no causal mask. BERT, from Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova at Google in October 2018, attends both directions at every layer.
7. That is why BERT cannot be trained on next-token prediction. If every position sees every position, the answer is already visible.
8. BERT is trained instead on **masked language modelling**: replace 15 per cent of tokens with a special [MASK] token and predict the originals. The masking there is of the input tokens, not of the attention matrix. Two different meanings of the same word, and this confuses people constantly.
9. The consequences of the choice:

Property	Causal (decoder)	Bidirectional (encoder)
Sees future tokens	No	Yes
Can generate text	Yes	No
Training signal per token	Every token	15 per cent of tokens
Best at	Generation, chat	Classification, embedding

10. The training-efficiency line in that table is important and often missed. A causal model gets a prediction target at every one of n positions. A masked language model gets one at roughly 0.15n positions, so it extracts less signal per pass over the data.
11. Prefix language modelling is the hybrid: bidirectional attention over a prompt prefix, causal attention over the generated part. Used in UniLM in 2019 and in some of the T5 variants.
12. Sliding window attention adds a second constraint: block anything further back than w positions. Mistral 7B, released October 2023, used w = 4096. That is covered in 48.13.
13. Observation tools: printing the mask is one line, but for real debugging the useful check is that every attention row sums to 1 and contains no NaN. A single NaN in one row propagates to the entire output within one layer.

48.5.6 WORDS - remember these

1. Mask — blocking some positions from being looked at — an additive [n, n] tensor of 0 and negative infinity applied before softmax.
2. Causal mask — you may not read ahead — the strictly upper-triangular block pattern, also called a look-ahead or autoregressive mask.
3. Padding mask — ignore the filler used to square up a batch — a per-sequence column mask blocking positions beyond the true length.
4. Autoregressive — predicting the next thing from all the previous things — a factorization of the joint probability as a product of conditionals.
5. Masked language modelling — hiding some input words and guessing them — the BERT objective, corrupting about 15 per cent of input tokens.
6. Sequence packing — gluing documents together to avoid padding — concatenating a corpus into a token stream and slicing fixed-length windows.
7. Bidirectional — able to see both left and right — an unmasked encoder attention pattern.

48.6 Multi-head attention

48.6.1 PLAIN - in simple words

1. One attention operation produces one set of weights per position. One opinion about what matters.
2. But a word usually needs several different things at once.
3. Take the word “hungry” in “the cat was hungry”. It needs to know what it describes, which is “cat”. It also needs to know its grammatical role. It also needs to know where it sits in the sentence.
4. Those are different questions, and one set of weights cannot answer them all, because the weights must add up to 1 and get spent once.
5. So the model runs several attention operations side by side, each with its own W_Q, W_K and W_V. Each one is called a **head**.
6. Head 1 might mostly look at the previous word. Head 2 might look at the subject of the sentence. Head 3 might look at matching brackets.
7. They all run at the same time on the same input, and they do not talk to each other while running.
8. When they finish, their outputs are laid side by side in one long row, which is called concatenation.
9. Then one final learned matrix mixes that long row back down to the original width. That matrix is W_O, the output projection.

10. The clever part is that heads cost nothing extra. If the model is 512 wide and you want 8 heads, each head is 64 wide, and 8 times 64 is 512.
11. So the total work is the same as one 512-wide head. You have bought variety for free by splitting rather than adding.
12. That is the trade: eight narrow views instead of one wide view, at identical cost.

48.6.2 PLAIN - a picture in your head

1. Picture a panel of eight specialist doctors examining the same patient at the same time.
2. The cardiologist looks at the heart. The neurologist looks at the nerves. The radiologist looks at the scans.
3. Each one asks different questions, of the same patient, and gets a different answer.
4. None of them sees the whole picture. Each is deliberately narrow.
5. When they are done, all eight write their findings on one long sheet.
6. A general physician reads the whole sheet and produces one combined assessment. That is W_O, the output projection.
7. The panel is faster than eight sequential consultations, because they work simultaneously.
8. And the panel is better than one generalist, because a generalist would have to spend the same total effort spread across all the specialities.

Where this comparison breaks: real specialists chose their speciality and know what it is. Heads are not assigned roles. They are eight identical copies with different random starting values, and whatever specialization appears is an accident of training that nobody designed. Many heads end up doing nothing useful at all, which no hospital would tolerate. And a doctor examines the actual patient. A head sees only a 64-dimensional slice of a vector that has already passed through many previous layers.

48.6.3 PLAIN - a worked example

1. Work out every shape for the 2017 base model, with a 10-token sentence.
2. The configuration: d_model = 512, 8 heads, d_k = d_v = 64, since 512 / 8 = 64.

```
input X                      [1, 10, 512]

project with W_Q [512, 512]  [1, 10, 512]
reshape into heads           [1, 10, 8, 64]
transpose head axis forward  [1, 8, 10, 64]

same for K and V             [1, 8, 10, 64]

scores per head Q K^T        [1, 8, 10, 10]
softmax over last axis       [1, 8, 10, 10]
weights times V              [1, 8, 10, 64]

transpose back               [1, 10, 8, 64]
concatenate heads            [1, 10, 512]
project with W_O [512, 512]  [1, 10, 512]
```

3. Look at the first and last lines. The input was [1, 10, 512] and the output is [1, 10, 512]. The shape is unchanged, which is why blocks can be stacked.
4. Look at the score tensor: [1, 8, 10, 10]. That is 8 separate 10-by-10 grids, one per head. 800 numbers.
5. There is no reshaping magic here. “Reshape into heads” just means reading the same 512 numbers as 8 groups of 64, without moving any data.
6. Head 0 gets dimensions 0 to 63. Head 1 gets 64 to 127. And so on to head 7, which gets 448 to 511.
7. Parameters: W_Q, W_K, W_V and W_O are each 512 by 512, so 262,144 each, and 1,048,576 for the whole multi-head attention block.

8. That is exactly 4 times d_model squared, which is a formula worth memorizing: attention costs 4 d^2 parameters per layer.
9. Now do the same for a modern model, Llama 3.1 8B: d_model = 4096, 32 query heads, 8 key-value heads, head dimension 128.
10. Note that 32 times 128 = 4096, so the query side still adds up to d_model.
11. But the key and value side uses only 8 heads, so 8 times 128 = 1024.

```
q_proj  [4096, 4096]  =  16,777,216 parameters
k_proj  [1024, 4096]  =   4,194,304 parameters
v_proj  [1024, 4096]  =   4,194,304 parameters
o_proj  [4096, 4096]  =  16,777,216 parameters
                         ----------
total attention        =  41,943,040 per layer
```

12. Each of the 8 key-value heads is shared by 4 query heads. That is grouped-query attention, and its purpose is to shrink the memory cost of generation, explained fully in 48.13.
13. Across 32 layers that is 1,342,177,280 attention parameters, which is 16.7 per cent of the model's 8,030,261,248 total.

48.6.4 PLAIN - what is really happening inside

1. Heads are not separate networks. They are one matrix multiplication whose output is interpreted as several groups.
2. W_Q is a single [512, 512] matrix. Nothing in it says "this column belongs to head 3". The head boundaries exist only in how the result is reshaped.
3. That is why the cost is identical to single-head attention with the full width. The reshape is free.
4. The heads become genuinely independent at the score step, because a query from head 3 is only ever dotted with keys from head 3.
5. This makes the attention computation **block diagonal** across heads. Eight small independent problems rather than one large one.
6. The only place heads mix is W_O at the end. Every output dimension of W_O is a weighted combination of all 512 concatenated numbers, so every head can influence every output.
7. That means W_O is not just a resize. It is the thing that decides how much each head's finding is worth, and it is learned.
8. A useful way to see it: W_O can be split into 8 blocks of [64, 512], one per head. Then the block's output is added to the residual stream. Each head writes its own contribution independently, and they sum.
9. This decomposition is exactly how the interpretability literature analyses attention, and it is mathematically exact, not an approximation.
10. Heads within one layer cannot see each other's results. Head 3 cannot use what head 5 found. That only becomes possible in the next layer.

48.6.5 TECHNICAL - the engineer's version

1. Multi-head attention, from the 2017 paper: concat(head_1, ..., head_h) W_O, where head_i = Attention(Q W_Q_i, K W_K_i, V W_V_i).
2. Standard configurations:

Model	d_model	Heads	Head dim
Transformer base 2017	512	8	64
BERT-base 2018	768	12	64
GPT-2 small 2019	768	12	64
Llama 3.1 8B 2024	4096	32	128

3. The convention d_k = d_model / h keeps total cost constant. It is a convention, not a requirement, and models that break it exist.
4. The 2017 paper ran an ablation: with d_model fixed at 512, 1 head scored 0.9 BLEU worse than 8 heads, and 32 heads was also slightly worse than 8. So more heads is not monotonically better once each head becomes too narrow.
5. What heads have been observed to learn, from published analysis:
    1. Positional heads that attend almost entirely to the previous token or the next token. These are the most reliable finding and appear in nearly every model.
    2. Syntactic heads that attend from a verb to its direct object, or from a determiner to its noun. Kevin Clark, Urvashi Khandelwal, Omer Levy and Christopher Manning documented these in BERT in 2019.
    3. Coreference heads that attend from a pronoun to its antecedent, also in the same 2019 analysis, at above-chance but well below perfect accuracy.
    4. Induction heads, which complete a repeated pattern by finding an earlier occurrence of the current token and copying what followed it. Described by Catherine Olsson and colleagues at Anthropic in 2022 and connected to in-context learning.
6. **The honest version:** this is not settled science. Head interpretability is active research, not established fact. The named roles above are real published observations on specific models, and they do not mean that heads in general have clean human-nameable jobs.
7. Redundancy is well established and cuts the other way. Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich and Ivan Titov showed in 2019 that pruning 38 of 48 encoder heads in a translation model cost only 0.15 BLEU on English-Russian WMT data.
8. Paul Michel, Omer Levy and Graham Neubig published the same finding independently in 2019, in a paper titled "Are Sixteen Heads Really Better Than One?". Many heads can be deleted at test time with little loss.
9. So the correct summary is: some heads do identifiable jobs, most heads are partly redundant, and the population as a whole is not a tidy division of labour.
10. Marketing claim to reject: any product page saying the model "has heads dedicated to reasoning" or similar. No such assignment is designed or verified.
11. Reproducing head analysis is straightforward. Load a model with `output_attentions=True`, take the [B, h, n, n] tensor, and for a previous-token head you will see nearly all mass on the first subdiagonal.

48.6.6 WORDS - remember these

1. Head — one independent attention view — one of h parallel attention computations, each of width d_model / h by convention.
2. Multi-head attention — several attention views run side by side — h parallel scaled dot-product attentions, concatenated and projected by W_O.
3. Concatenation — laying results side by side into one long row — joining the h per-head outputs along the feature axis to restore width d_model.
4. Output projection — the matrix that mixes the heads back together — W_O, of shape [d_model, d_model], applied after concatenation.
5. Head dimension — how wide one head is — d_k, typically 64 in 2017-era models and 128 in current ones.
6. Induction head — a head that continues a repeated pattern — a two-head circuit that matches the current token to a previous occurrence and copies the following token.
7. Head pruning — deleting heads that do not earn their keep — zeroing whole heads at inference, shown to cost little accuracy in several 2019 studies.

48.7 Position: attention has no idea what order things are in

48.7.1 PLAIN - in simple words

1. Here is a fact about attention that catches everybody by surprise.
2. Attention has no concept of order. None. It does not know which word came first.
3. To attention, a sentence is a bag of items, not a sequence.
4. Prove it to yourself. Look back at the formula: softmax(Q K^T / sqrt(d_k)) V.
5. Find the position number in it. There is none. There is no i, no t, no index of any kind.
6. Every operation is a dot product between two token vectors, and a dot product does not care where either token sits.
7. So "dog bites man" and "man bites dog" would produce exactly the same set of output vectors, just shuffled the same way as the input.
8. That would be a catastrophe for language, where order is most of the meaning.
9. So order has to be added by hand. The model is given, for each position, some extra numbers that say where that position is.
10. The original method was to add a fixed pattern of sine and cosine waves to each token's vector before the first layer.
11. A second method is to learn a separate vector for position 1, position 2, and so on, exactly like an embedding table for positions.
12. The modern method, used by almost every current model, is different and cleverer. It rotates the query and key vectors by an angle proportional to their position.
13. It is called rotary position embedding, or RoPE, and the reason it won is that it makes the score between two tokens depend only on how far apart they are, not on where they are in absolute terms.

48.7.2 PLAIN - a picture in your head

1. Imagine a set of magnetic word tiles scattered on a table, one word per tile.
2. Attention reads all the tiles at once and works out which ones relate to which others.
3. Now shake the table. The tiles land in a different arrangement. Attention gives exactly the same answer, because it never looked at the arrangement.
4. To fix this, you write a number on the back of each tile: 1, 2, 3, 4.
5. That is learned absolute position embedding. Simple, and it fails as soon as you meet tile 5,000 when you only ever wrote numbers up to 4,096.
6. Now a better fix. Instead of writing numbers, you tilt each tile by an angle. Tile 1 is tilted 10 degrees, tile 2 is tilted 20, tile 3 is tilted 30.
7. When two tiles are compared, what matters is the difference in their tilts. Tiles 1 and 2 differ by 10 degrees. Tiles 500 and 501 also differ by 10.
8. So the comparison naturally depends on the gap between positions, not the absolute positions.
9. That is rotary position embedding. Tilt by angle, compare by difference.

Where this comparison breaks: a tile has one tilt angle. A real RoPE vector is split into 64 pairs of numbers, and each pair is rotated at its own speed, from one full turn every 6.3 positions down to one full turn every two and a half million positions. So it is 64 clocks running at 64 different speeds, not one tilt. And the tilt is applied to the query and key vectors only, never to the values.

48.7.3 PLAIN - a worked example

1. First, the proof that attention ignores order. Take the four-token example from 48.4, which produced this output:

```
Output          dim0    dim1    dim2
  the         1.6405  2.1798  1.9607
  cat         0.6807  2.8645  1.5452
```

```
  was         0.7257  2.8350  1.5266
  hungry      0.5022  2.9000  1.4022
```

2. Now shuffle the input rows into the order hungry, cat, the, was, and run the identical computation with the identical matrices:

```
Output of shuffled input     dim0    dim1    dim2
  hungry                   0.5022  2.9000  1.4022
  cat                      0.6807  2.8645  1.5452
  the                      1.6405  2.1798  1.9607
  was                      0.7257  2.8350  1.5266
```

3. Every row is identical to before. Only their order changed, in exactly the same way the input changed.
4. That property has a name: **permutation equivariance**. Shuffle the input, and the output is shuffled the same way, with no other change.
5. This was verified numerically, not asserted. The two matrices agree to floating-point precision.
6. Now sinusoidal position encoding. The formula from the 2017 paper, for position pos and dimension index i:

```
PE(pos, 2i)   = sin( pos / 10000^(2i/d_model) )
PE(pos, 2i+1) = cos( pos / 10000^(2i/d_model) )
```

7. With d_model = 8 there are 4 sine-cosine pairs. Their frequencies:

Pair	Frequency	Wavelength
0	1.0	6.28 positions
1	0.1	62.8 positions
2	0.01	628 positions
3	0.001	6283 positions

8. The actual values for the first five positions:

```
pos 0:  0.0000  1.0000  0.0000  1.0000  0.0000  1.0000  0.0000  1.0000
pos 1:  0.8415  0.5403  0.0998  0.9950  0.0100  1.0000  0.0010  1.0000
pos 2:  0.9093 -0.4161  0.1987  0.9801  0.0200  0.9998  0.0020  1.0000
pos 3:  0.1411 -0.9900  0.2955  0.9553  0.0300  0.9996  0.0030  1.0000
pos 4: -0.7568 -0.6536  0.3894  0.9211  0.0400  0.9992  0.0040  1.0000
```

9. Read the columns. The first pair changes fast and wraps around quickly. The last pair barely moves at all across five positions.
10. That is the design. Fast waves distinguish neighbours precisely, slow waves distinguish far-apart positions coarsely. It is a positional binary counter made of continuous waves.
11. Now RoPE, with real numbers. Take a query pair q = [1.0, 0.0] and a key pair k = [0.6, 0.8]. Their plain dot product is 0.6.
12. RoPE rotates q by m times theta and k by n times theta, where m and n are the two positions. With theta = 1.0 radian:

m	n	m - n	Dot after rotation
1	0	1	0.997358
5	4	1	0.997358
100	99	1	0.997358
3	0	3	-0.481099
7	4	3	-0.481099

13. Look at the last column. Every pair with the same gap gives the same result, to six decimal places, whether the positions are 1 and 0 or 100 and
    99.
14. That is the whole selling point of RoPE, demonstrated numerically. The score depends on relative distance only.

48.7.4 PLAIN - what is really happening inside

1. Sinusoidal encoding is added to the token embedding, once, before layer 1. Token vector plus position vector, element by element.
2. That means position information and word information are mixed together in the same numbers from the start, and every layer sees the sum.
3. Learned absolute embeddings work identically, except the position vectors are parameters found by training rather than computed from a formula.
4. GPT-2 uses learned absolute embeddings: a table of 1,024 rows by 768 columns, which is 786,432 parameters spent purely on saying where things are.
5. Both of these have the same weakness. They are absolute. Position 3,000 has its own vector, unrelated to position 2,999.
6. And if the table has 1,024 rows, position 1,024 does not exist. The model cannot be run on longer text at all.
7. Sinusoids can at least be computed for any position, but a model trained only on positions 0 to 511 has never seen what position 800 looks like, and in practice quality collapses beyond the trained length.
8. RoPE takes a completely different route. It never adds anything. It rotates.
9. Split the query vector into consecutive pairs: dimensions (0,1), (2,3), (4,5), and so on. Treat each pair as a point on a flat plane.
10. Rotate pair j by an angle of position times theta_j, where theta_j gets smaller for later pairs, exactly like the sinusoid frequencies.
11. Do the same to the key vector, using the key's own position.
12. Now take the dot product. Rotating both vectors by the same amount leaves the dot product unchanged, so only the difference in rotation survives.
13. That is basic geometry: a dot product is invariant under rotation. Rotating q by m and k by n is the same as rotating q by (m - n) and leaving k alone.
14. So the score naturally becomes a function of m - n. No extra terms, no lookup table, no learned parameters at all.
15. RoPE is applied inside every attention layer, to Q and K only, every time. It is not applied to V, and it is not added to the residual stream.

48.7.5 TECHNICAL - the engineer's version

1. Permutation equivariance is exact: for any permutation matrix P, Attention(PX) = P Attention(X). This is why position injection is mandatory rather than helpful.
2. **Sinusoidal**, 2017. Fixed, no parameters, defined for all positions. The base 10,000 sets the slowest wavelength. The paper's stated reason for choosing sinusoids is that PE(pos + k) is a linear function of PE(pos) for any fixed offset k, so relative offsets are in principle learnable as a linear map. This was verified numerically for this chapter: for the fastest pair with theta = 1, rotating PE(5) by 3 radians reproduces PE(8) exactly.
3. The paper also reports that learned and sinusoidal encodings performed almost identically, and chose sinusoidal for its extrapolation potential.
4. Sinusoid dot products fall with distance, which is a useful inductive bias. Measured at d_model = 64: the dot product of PE(0) with itself is 32.0, with PE(1) is 30.92, with PE(10) is 21.05, and with PE(200) is 11.44.

5. **Learned absolute**, used by BERT (2018), GPT-1, GPT-2 and GPT-3. A table of shape [max_positions, d_model]. Hard cap on length, no extrapolation at all.
6. **Relative position representations**, from Peter Shaw, Jakob Uszkoreit and Ashish Vaswani in 2018, add a learned vector for each relative offset into the key and value computation.
7. **T5 relative position bias**, from Colin Raffel and colleagues in 2019, adds a learned scalar to the attention logit per bucket of relative distance, shared across layers, with logarithmically spaced buckets. Cheap and effective.
8. **ALiBi**, from Ofir Press, Noah Smith and Mike Lewis in 2021, adds a linear penalty to the attention logit: minus m times the distance, where m is a fixed per-head slope, typically a geometric sequence such as 1/2, 1/4, 1/8. No parameters at all, and it extrapolates well beyond the training length. The paper's title is "Train Short, Test Long".
9. **RoPE**, from Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen and Yunfeng Liu, in the RoFormer paper first posted in April 2021. It is the dominant choice as of 2026. Llama, Mistral, Qwen, Gemma, DeepSeek and Phi all use it.
10. RoPE frequencies use theta_j = base^(-2j/d_head). With base 10,000 and head dimension 128, the fastest pair has wavelength 6.3 positions and the slowest has wavelength 54,410 positions.
11. Raising the base stretches everything. With base 500,000, as used in Llama 3.1, the slowest pair has wavelength 2,559,196 positions. That is how a 128k-token context is supported without the slowest frequency wrapping.

Method	Year	Parameters	Extrapolates
Sinusoidal	2017	0	Poorly
Learned absolute	2018	max_pos x d	Not at all
T5 bias	2019	~ 32 per head	Moderately
ALiBi	2021	0	Well
RoPE	2021	0	With extension

12. Context extension. A model trained at 4,096 tokens fails at 32,768 because the rotation angles at position 30,000 are outside anything it saw.
13. **Position interpolation**, from Shouyuan Chen, Sherman Wong, Liangjian Chen and Yuandong Tian at Meta in June 2023, divides every position index by a scale factor s. To go from 4,096 to 32,768, use s = 8, so position 32,767 is presented to the model as 4,095.9.
14. That keeps every angle inside the trained range, at the cost of squashing fine positional detail. It requires a short fine-tune, typically 1,000 steps, and works remarkably well.
15. **NTK-aware scaling**, which appeared as a community proposal in mid-2023, changes the RoPE base instead of the positions, stretching low frequencies more than high ones so nearby tokens keep their resolution.
16. **YaRN**, from Bowen Peng, Jeffrey Quesnelle, Honglu Fan and Enrico Shippole in 2023, combines the two: interpolate the low-frequency dimensions, leave the high-frequency ones alone, apply a ramp in between, and add a temperature adjustment to the attention logits. It reaches long contexts with about ten times less fine-tuning data than plain interpolation.
17. Llama 3.1, released July 2024, used a variant of this family to reach a 131,072-token context from a shorter pretraining length, together with base 500,000.
18. Established fact: RoPE gives exact relative-distance dependence, and interpolation methods extend context in practice. Active research: how much real long-range capability extension gives, as opposed to merely not crashing. Benchmarks such as RULER, from NVIDIA in 2024, found many models claiming 32,000-token contexts performed well at much shorter lengths.

48.7.6 WORDS - remember these

1. Permutation equivariance — shuffle the input, get the same output shuffled — Attention(PX) = P Attention(X) for any permutation matrix P.
2. Positional encoding — extra numbers telling the model where each token sits — any scheme injecting order information into an order-blind operation.
3. Sinusoidal encoding — fixed sine and cosine waves at many speeds — the 2017 scheme with frequencies base^(-2i/d), base 10,000.
4. Absolute position — where a token is in the whole sequence — an index from 0 to n-1, encoded directly.
5. Relative position — how far apart two tokens are — the offset m - n, which is what a score should depend on.
6. RoPE — rotating queries and keys by an angle set by position — rotary position embedding, applying a block-diagonal rotation to Q and K.
7. RoPE base (theta) — the number controlling how slow the slowest wave is — 10,000 originally, raised to 500,000 or higher for long contexts.
8. Position interpolation — squeezing long positions into the trained range — dividing position indices by a scale factor s, then fine-tuning.
9. YaRN — a smarter context extension that treats fast and slow waves differently — wavelength-dependent interpolation with a logit temperature correction.
10. ALiBi — penalizing distant tokens with a straight-line penalty — adding minus m times |i - j| to the attention logit, with fixed per-head slopes.

48.8 The rest of the block: feed-forward, residuals and normalization

48.8.1 PLAIN - in simple words

1. Attention is only half of a transformer block. The other half is a small ordinary network applied to each position separately.
2. It is called the **feed-forward network**, or FFN, or the MLP. All three names mean the same thing.
3. It has exactly two steps: widen, then narrow. Multiply by a matrix that makes the vector about four times wider, apply a non-linear function, multiply by a matrix that brings it back to the original width.
4. The important word is "separately". Every position goes through the same FFN, on its own, with no reference to any other position.
5. So attention is where positions talk to each other, and the FFN is where each position thinks about what it just heard.
6. Now the surprising part: this simple two-matrix block holds most of the model's parameters. About 70 per cent for an 8-billion model, and about 80 per cent for larger ones.
7. Chapter 47 counted this exactly. For Llama 3.1 8B, the feed-forward matrices hold 176,160,768 parameters per layer against 41,943,040 for attention.
8. So attention gets all the attention, and the FFN holds the weights.
9. Two more pieces complete the block, and both are about making deep stacks trainable rather than about capability.
10. A **residual connection** means: whatever a sub-layer computes, add the original input back on. Output equals input plus change, not just change.
11. Without residuals, a 32-layer stack cannot be trained. The correction signal dies on the way back, exactly as it did in recurrent networks.
12. **Layer normalization** rescales each vector so its numbers have a predictable size, which stops values drifting to extremes as they pass through dozens of layers.
13. That is the whole block: attention, add the input back, normalize, feed-forward, add the input back, normalize. Then repeat.

48.8.2 PLAIN - a picture in your head

1. Picture a long assembly line with 32 stations, each one modifying a product.
2. In a bad design, each station takes the previous station's output and rebuilds it from scratch. If station 7 does something wrong, everything downstream is corrupted, and the factory manager cannot tell which station caused it.
3. In the residual design, each station is given the product and told: do not rebuild it. Just add your modification on top and pass it along.
4. Now the product carries a continuous history, and every station's change is a small addition rather than a replacement.
5. If the manager finds a fault at the end, the complaint can travel back down the line unchanged, because the additions form a straight unbroken path.
6. That straight path is exactly what a residual connection creates for gradients. It is often called the residual stream, and it runs the whole depth of the model.
7. Normalization is the quality inspector between stations, resetting the product to a standard size and shape so the next station receives something in the range it expects.

Where this comparison breaks: nothing is physically added to a product, and the "modification" is a vector added element by element, which can cancel earlier contributions rather than accumulate. Also, real assembly stations have fixed jobs. These are all trained together, so a station's job is defined only by what the others ended up doing.

48.8.3 PLAIN - a worked example

1. First, why residual connections matter, with arithmetic.
2. During backpropagation, the correction signal is multiplied by something at each layer. Call it the per-layer factor.
3. Without residuals, that factor depends entirely on the layer's weights and there is no reason for it to be near 1.

Per-layer factor	After 32 layers	After 96 layers
0.90	0.034	0.00004
0.95	0.194	0.0073
1.00	1.000	1.000
1.05	4.765	108.2

4. At 0.9 per layer, a 96-layer model receives 0.004 per cent of the signal at the bottom. Training the early layers becomes impossible.
5. With a residual connection, the layer computes $x + f(x)$, so the derivative is $1 + f'(x)$. The 1 is there whatever f does.
6. So there is always a path with factor exactly 1, and the product along that path stays exactly 1 no matter how deep the stack is.
7. That is the entire trick, and it came from image models: Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun published ResNet in December 2015, trained 152 layers, and won the ImageNet competition that year.
8. Now layer normalization, worked on a real vector of 8 numbers:

```
x        =  2.0  -1.0   0.5   4.0  -2.5   1.0   0.0   3.0
mean     =  0.875
variance =  3.921875
std dev  =  1.980372
```

9. Subtract the mean and divide by the standard deviation:

```
normed   =  0.5681 -0.9468 -0.1894  1.5780
           -1.7042  0.0631 -0.4418  1.0730
```

10. Check: the mean of the normalized vector is 0.0 and its standard deviation is 1.0. That is guaranteed by construction.
11. Then multiply by a learned scale and add a learned shift, one per dimension, so the layer can undo the normalization if that helps.
12. Crucially, the mean and variance were computed across the 8 features of this one token. Not across the batch. Not across the sequence.
13. That is why it works with any batch size, including batch size 1, which is the normal case when you chat with a model.
14. RMSNorm is the cheaper modern variant. It skips the mean subtraction and divides by the root mean square:

```
root mean square of x = 2.165064
normed = 0.9238 -0.4619  0.2309  1.8475
        -1.1547  0.4619  0.0000  1.3856
```

15. It has one learned parameter per dimension instead of two, and in practice performs the same. Llama, Mistral, Qwen and Gemma all use it.

48.8.4 PLAIN - what is really happening inside

1. The feed-forward network, written out: FFN(x) = W_2 (activation(W_1 x + b_1))
   - b_2.
2. W_1 has shape [d_model, d_ff] and W_2 has shape [d_ff, d_model], with d_ff usually 4 times d_model.
3. The 4x ratio is a convention from the 2017 paper, where d_model was 512 and d_ff was 2048. It is not a law and modern models vary it.
4. The activation function was ReLU in 2017, then GELU in BERT and GPT-2, and now SwiGLU in most current models. Chapter 46 covered what these do.
5. SwiGLU uses three matrices rather than two: a gate, an up-projection and a down-projection, with the gate multiplied elementwise into the up path.
6. Because it has three matrices, models using SwiGLU shrink d_ff to about 8/3 of d_model to keep the parameter count similar. Llama 3.1 8B uses 14,336 with d_model 4096, which is 3.5x rather than the naive 4x.
7. Now pre-norm versus post-norm, which is a real and consequential difference.
8. **Post-norm**, the 2017 original: x -> attention -> add x -> normalize. The normalization sits after the addition, on the main path.
9. **Pre-norm**, used by almost everything since about 2019: x -> normalize -> attention -> add x. The normalization sits inside the branch, and the main path from input to output has nothing on it but additions.
10. That difference is everything. In pre-norm, the residual stream is a clean sum with no normalization interrupting it, so the gradient path really is unobstructed all the way down.
11. In post-norm, every residual addition is immediately rescaled, which shrinks the accumulated signal and makes deep stacks unstable.
12. The practical consequence: post-norm transformers need a learning-rate warmup or they diverge. Pre-norm ones train stably without it.
13. The 2017 paper used post-norm with 4,000 warmup steps, and that warmup was not optional.

48.8.5 TECHNICAL - the engineer's version

1. The complete pre-norm decoder block, drawn:

```
        x  (residual stream, width d_model)
        |
        +--------------------------+
        |                          |
     [RMSNorm]                     |
        |                          |
   [Multi-head attention]          |
   [ +RoPE on Q and K   ]          |
   [ +causal mask       ]          |
        |                          |
        +----------> (add) <--------+
                      |
                      x1
                      |
        +-------------+------------+
        |                          |
     [RMSNorm]                     |
        |                          |
   [FFN: gate/up -> SwiGLU -> down] |
        |                          |
        +----------> (add) <--------+
                      |
                  output  (width d_model)
```

2. Parameter accounting per layer, d = d_model:

Component	Parameters	GPT-2 small
Attention (4 matrices)	4 d^2	2,359,296
FFN (2 matrices, 4x)	8 d^2	4,718,592
Two LayerNorms	4 d	3,072
Block total	12 d^2 + 4d	7,080,960

3. So the FFN is exactly two thirds of a classic block's parameters, from the ratio 8 d^2 to 12 d^2.
4. For Llama 3.1 8B with SwiGLU: attention is 41,943,040 and the FFN is 3 x 4096 x 14336 = 176,160,768, so the FFN is 80.8 per cent of the layer.
5. Across the whole model, including the two 525,336,576-parameter vocabulary tables, the FFN share is 70.2 per cent and attention is 16.7 per cent. These figures match Chapter 47 exactly.
6. Layer normalization is from Jimmy Lei Ba, Jamie Ryan Kiros and Geoffrey Hinton, July 2016. It normalizes over the feature axis of a single example, unlike batch normalization from Sergey Ioffe and Christian Szegedy in 2015, which normalizes over the batch axis.
7. That distinction is why layer norm won in language models: batch statistics are useless when sequences have different lengths and batch size can be 1.
8. RMSNorm is from Biao Zhang and Rico Sennrich, 2019. It drops the mean subtraction and the shift term, saving one reduction and d parameters per norm, for roughly the same quality.
9. Epsilon inside the square root prevents division by zero. Typical values are 1e-5 for LayerNorm and 1e-5 or 1e-6 for RMSNorm. This is an implementation detail that occasionally matters for numerical reproducibility.
10. Pre-norm was analysed formally by Ruibin Xiong and colleagues in the 2020 paper "On Layer Normalization in the Transformer Architecture", which showed that post-norm gradients at initialization scale with the square root of depth while pre-norm gradients do not, explaining the warmup requirement.

11. Baevski and Auli in 2018 and Nguyen and Salazar in 2019 had already reported the practical benefit. GPT-2, in February 2019, used pre-norm.
12. A third variant exists: sandwich norm, or normalization both before and after the sub-layer, used in some very large training runs including Gemma 2 in 2024, to control activation growth.
13. The residual stream view: because every block writes $x + f(x)$, the model's activations at layer L are the embedding plus the sum of every sub-layer's output so far. This decomposition is exact and is the foundation of mechanistic interpretability work, notably "A Mathematical Framework for Transformer Circuits" from Anthropic in December 2021.
14. Observation: activation norms typically grow with depth in trained transformers, often by an order of magnitude from layer 1 to the last layer. This is measurable with a forward hook and is a routine sanity check when training goes wrong.

48.8.6 WORDS - remember these

1. Feed-forward network — the widen-then-narrow part applied to each position — FFN or MLP, two or three linear layers with a non-linearity, d_ff typically about 4 x d_model.
2. Position-wise — applied to each token independently — the same weights applied to every position with no mixing across the sequence axis.
3. Residual connection — adding the input back onto the output — the identity shortcut $x + f(x)$, from ResNet in 2015.
4. Residual stream — the main pipe running the depth of the model — the d_model-wide activation that every block reads from and adds to.
5. Layer normalization — rescaling one token's numbers to a standard size — normalizing over the feature axis per example, with learned scale and shift.
6. RMSNorm — cheaper layer normalization — dividing by the root mean square with no mean subtraction and no shift term.
7. Pre-norm — normalize before the sub-layer, add after — the arrangement that keeps the residual path free of normalization, standard since about 2019.
8. Post-norm — normalize after adding — the 2017 arrangement, which needs learning-rate warmup to train stably.
9. SwiGLU — a gated activation using three matrices — swish-gated linear unit, standard in the Llama, Mistral and Qwen families.

48.9 The whole model, assembled

48.9.1 PLAIN - in simple words

1. Everything so far has been parts. Here is the whole machine, in order.
2. Step 1: your text is chopped into tokens and each token becomes an integer id. Chapter 47 covered this.
3. Step 2: each id is used to look up a row in the embedding table. Now every token is a vector.
4. Step 3: position information is injected. In modern models this happens inside attention, as a rotation, rather than as an addition here.
5. Step 4: the stack. N identical blocks, one after another. Each block is attention then feed-forward, both with residual connections and normalization.
6. Identical means identical in shape, not in values. Every block has its own separate parameters.
7. Step 5: one final normalization at the top of the stack.
8. Step 6: multiply by one last matrix that turns each d_model-wide vector into one number per token in the vocabulary. Those numbers are called **logits**.
9. Step 7: softmax over the vocabulary turns the logits into probabilities that add up to 1.
10. The output is one probability distribution per input position. For a 10-token prompt, you get 10 distributions.
11. During generation you only use the last one, because that is the prediction for what comes next.

12. During training you use all of them, because each position is a separate prediction problem, which is why masked parallel training is so efficient.

48.9.2 PLAIN - a picture in your head

1. Picture a tall narrow building with 32 identical floors.
2. On the ground floor, a crowd of people walks in. One person per token. Each is handed a card with numbers on it.
3. On every floor, two things happen. First everybody talks to everybody and updates their card based on what they heard. Then each person sits alone and thinks, updating their card again.
4. Nobody joins and nobody leaves. The same number of people walk out of floor 32 as walked into floor 1.
5. The cards get richer on every floor. A person who arrived holding only the word "it" leaves floor 32 holding something that encodes what "it" refers to, what grammatical role it plays, and what is likely to come next.
6. At the top, only the last person in the queue is asked a question: given everything you now know, what word comes next?
7. That person's card is compared against a list of 128,256 possible words, and a score is produced for each.

Where this comparison breaks: the people do not know anything and are not thinking. Each is a vector being multiplied by matrices. And the last person's answer is not derived from a conversation, but from a single dot product of their card against every row of a very large table. Also, at training time every person is asked a question, not just the last one.

48.9.3 PLAIN - a worked example

1. Trace one prompt through Llama 3.1 8B, showing the tensor shape at every stage. Configuration: vocabulary 128,256, d_model 4,096, 32 layers, 32 query heads, 8 key-value heads, head dimension 128, d_ff 14,336.
2. The prompt is six tokens long, batch size 1.

stage	shape	numbers
text	"the cat was hungry"	-
token ids	[1, 6]	6
embedding lookup	[1, 6, 4096]	24,576
block 1 input	[1, 6, 4096]	24,576
norm	[1, 6, 4096]	24,576
q after projection	[1, 32, 6, 128]	24,576
k after projection	[1, 8, 6, 128]	6,144
v after projection	[1, 8, 6, 128]	6,144
k,v repeated to 32 heads	[1, 32, 6, 128]	24,576
scores	[1, 32, 6, 6]	1,152
weights after softmax	[1, 32, 6, 6]	1,152
attention output	[1, 32, 6, 128]	24,576
after o_proj	[1, 6, 4096]	24,576
after residual add	[1, 6, 4096]	24,576
norm	[1, 6, 4096]	24,576
ffn hidden (gate and up)	[1, 6, 14336]	86,016
after down_proj	[1, 6, 4096]	24,576
after residual add	[1, 6, 4096]	24,576
block 2 ... block 32	[1, 6, 4096]	24,576
final RMSNorm	[1, 6, 4096]	24,576
lm_head projection	[1, 6, 128256]	769,536
softmax over vocabulary	[1, 6, 128256]	769,536
take last position	[1, 128256]	128,256

3. Read down the shape column. From the embedding to the final norm, the shape never changes: [1, 6, 4096]. That is the residual stream, and it is constant width for the entire depth.
4. The only places the shape changes are inside a block, temporarily, and at the very end.
5. The last projection is the expensive one in terms of output size. It turns 24,576 numbers into 769,536 numbers, because the vocabulary is large.
6. That final matrix, `lm_head.weight`, is [128256, 4096], holding 525,336,576 parameters. It is 6.5 per cent of the model on its own.
7. Some models tie this matrix to the input embedding table, using the same numbers for both. GPT-2 does this. Llama 3.1 8B does not.
8. Tying halves the vocabulary cost. Untying gives slightly better quality at larger scale. This is a real trade-off and both choices are in current use.

48.9.4 PLAIN - what is really happening inside

1. A **logit** is a raw score before it becomes a probability. That is its only definition. It can be any real number, positive or negative.
2. The name comes from statistics, where the logit function is the logarithm of the odds. In this context it just means "the thing softmax eats".
3. There is exactly one logit per vocabulary entry. For Llama 3.1, 128,256 of them, every single time a token is produced.
4. Each logit is one dot product: the final hidden vector of length 4,096 dotted with one row of the output matrix, also length 4,096.
5. So the model is asking, for every possible next token: how well does my current state line up with the direction that means this token?
6. Softmax then converts. Exponentiate every logit, add them all up, divide.
7. A logit of 4.0 against a logit of 3.2 becomes probabilities 0.57 and 0.26, because what matters is the difference, not the absolute values.
8. Adding 100 to every logit changes nothing at all. Softmax is invariant to a constant shift. That is why logits are not comparable across models or even across steps.
9. During training, the loss is the negative log of the probability the model gave to the token that actually came next. That is cross-entropy loss, from Chapter 46.
10. If the model gave the right token probability 0.5, the loss is 0.693. If it gave 0.01, the loss is 4.605. The worse the guess, the larger the penalty, growing without bound.
11. Nothing else is optimized. There is no separate objective for grammar, for facts, or for helpfulness during pretraining. One number, minimized.

48.9.5 TECHNICAL - the engineer's version

1. Full forward pass, in the order the code runs:

```
h = embed_tokens(ids)                    # [B, n, d]
for layer in layers:                     # N identical blocks
    r = h
    h = rmsnorm(h, layer.attn_norm)
    h = attention(h, rope, causal_mask)  # +RoPE inside
    h = r + h
    r = h
    h = rmsnorm(h, layer.ffn_norm)
    h = swiglu_ffn(h)
    h = r + h
h = rmsnorm(h, final_norm)               # [B, n, d]
logits = h @ lm_head.T                   # [B, n, vocab]
```

2. Reference configurations, all published:

Model	Layers	d_model	Vocab
Transformer base 2017	6 + 6	512	37,000
GPT-2 small 2019	12	768	50,257
Llama 3.1 8B 2024	32	4096	128,256
Llama 3.1 70B 2024	80	8192	128,256

3. Total parameter count for Llama 3.1 8B, worked exactly: 32 layers of 218,112,000, plus 525,336,576 for the input embedding, plus 525,336,576 for the output projection, plus 4,096 for the final norm, giving 8,030,261,248.
4. Activation memory for a forward pass is separate from weights and is dominated by the FFN hidden state. At n = 4096 and bfloat16, one layer's [1, 4096, 14336] gate tensor is 117 MiB, and there are two of them.
5. Weight tying is controlled by a flag, `tie_word_embeddings`, in the Hugging Face config. It is true for GPT-2 and for Gemma, false for Llama 3.1 8B.
6. The final logit tensor is genuinely large. At n = 4,096 and vocabulary 128,256 in float32, the logits alone are 2.1 GiB. Training frameworks compute the loss in chunks along the sequence axis to avoid materializing it.
7. At inference only the last position's logits are needed, so implementations slice the hidden state to [B, 1, d] before the lm_head multiply. Skipping this is a common and expensive beginner mistake.
8. Some models scale the logits. Gemma multiplies the embedding output by the square root of d_model; some models divide final logits by a constant, called logit softcapping, which Gemma 2 uses with a cap of 30.0 on the final layer.
9. Numerical note: logits are computed in float32 even in bfloat16 models, because cross-entropy over 128,256 classes is sensitive to precision in the log-sum-exp reduction.

48.9.6 WORDS - remember these

1. Logit — a raw score before it is turned into a probability — the pre-softmax output, one per vocabulary entry, unbounded in both directions.
2. lm_head — the final matrix that scores every possible next token — the [vocab, d_model] output projection, sometimes tied to the input embedding.
3. Weight tying — reusing the input embedding table as the output matrix — sharing one [vocab, d] tensor for both, halving vocabulary parameters.
4. Residual stream — the constant-width pipe running the depth of the model — the [B, n, d_model] activation that all blocks read and write.
5. Cross-entropy loss — how wrong the guess was, in one number — the negative log probability assigned to the correct next token, averaged over positions.
6. Forward pass — running the model once from input to output — one evaluation of the network with no gradient computation.
7. Hidden state — the working vector for one token at one depth — a d_model-wide slice of the residual stream at a given layer and position.

48.10 The 2017 paper

48.10.1 PLAIN - in simple words

1. The transformer was introduced in one paper: "Attention Is All You Need".
2. It was posted publicly on 12 June 2017, and presented at the NeurIPS conference in December 2017 in Long Beach, California.
3. There were eight authors: Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan Gomez, Lukasz Kaiser and Illia Polosukhin.

4. Most were at Google Brain or Google Research. Aidan Gomez was at the University of Toronto, working with Google Brain.
5. The paper has an unusual footnote saying the authors contributed equally and the listing order is random. That was deliberate and unusual for the time.
6. It was a machine translation paper. Nothing in it is about chatbots, reasoning or general-purpose assistants.
7. The claim was narrow: you can throw away recurrence entirely and get better translation quality in a fraction of the training time.
8. The headline results were 28.4 BLEU on English-to-German and 41.8 BLEU on English-to-French, both on the WMT 2014 test sets.
9. BLEU is a machine translation score from 0 to 100 that measures overlap with human reference translations. Higher is better.
10. The 41.8 was a new single-model best, reached after 3.5 days on eight GPUs.
11. That last number is the point of the whole paper. Competing systems needed far more compute for worse results.
12. What was new was not attention. Attention had existed since 2014.
13. What was new was deleting the recurrent network and keeping only attention, which made the whole sequence processable in parallel.

48.10.2 PLAIN - a picture in your head

1. Imagine a road crew resurfacing a long road, working one metre at a time, because each metre must set before the next can be laid.
2. Adding more workers does not help. The job is serial by construction.
3. Now imagine somebody invents a material that sets independently everywhere, so a thousand workers can lay a thousand metres simultaneously.
4. The material is not better. One metre of road is one metre of road. What changed is that the work is now divisible.
5. That is what removing recurrence did. It converted a serial job into a parallel one, and hardware that was already sitting idle could finally be used.

Where this comparison breaks: attention is not merely a parallel version of the same computation. It also gives every position direct access to every other position, which recurrence never did. So the change was both a speed change and a capability change, and the paper's own results reflect both.

48.10.3 PLAIN - a worked example

1. The two model sizes reported:

Setting	Base	Big
Parameters	65 million	213 million
d_model	512	1024
Heads	8	16
Training time	12 hours	3.5 days

2. Both used 6 encoder layers and 6 decoder layers, and were trained on 8 NVIDIA P100 GPUs in one machine.
3. The base model ran 100,000 steps at about 0.4 seconds each, which is 12 hours.
4. The big model ran 300,000 steps at about 1.0 second each, which is 3.5 days.
5. Now the parallelism argument in concrete terms. Take a 1,000-token training sequence.
6. A recurrent model must execute 1,000 dependent steps. Each step is a small matrix multiplication that cannot fill a GPU, and between each pair of steps the chip waits.

7. A transformer executes one matrix multiplication over all 1,000 positions. With d_model = 512, that single call is 2 x 1000 x 512 x 512, which is 524 MFLOP of dense work handed to the chip in one piece.
8. The chip's utilization goes from a few per cent to near its limit, and the speed-up is far larger than the reduction in arithmetic.
9. That is why the paper could report state-of-the-art translation from a single 8-GPU machine in under four days.

48.10.4 PLAIN - what is really happening inside

1. The architecture in the paper is an encoder-decoder, not the decoder-only design used by chat models today.
2. The encoder has 6 layers, each with bidirectional self-attention and a feed-forward network. It reads the source sentence.
3. The decoder has 6 layers, each with three sub-layers: causal self-attention over what has been produced so far, cross-attention into the encoder's output, and a feed-forward network.
4. Post-norm was used throughout, with a warmup schedule that raised the learning rate for 4,000 steps then decayed it as the inverse square root of the step number.
5. Dropout of 0.1 was applied to every sub-layer output and to the sum of the embedding and positional encoding.
6. Label smoothing of 0.1 was used, which hurts perplexity and helps BLEU, and the paper says so plainly.
7. Decoding used beam search with beam size 4 and length penalty 0.6.
8. Every one of those settings is an ordinary engineering choice. None of them is the idea. The idea is one sentence: remove recurrence.

48.10.5 TECHNICAL - the engineer's version

1. Reported results, WMT 2014 test sets, from Table 2 of the paper:

System	EN-DE BLEU	EN-FR BLEU
Previous best single	25.16	41.29
Previous best ensemble	26.36	41.16
Transformer base	27.3	38.1
Transformer big	28.4	41.8

2. Training cost in floating-point operations: 3.3e18 for the base model and 2.3e19 for the big model, both reported in the paper, and both one to two orders of magnitude below the competing systems in the same table.
3. The paper also reported English constituency parsing results, to argue the architecture generalizes beyond translation. Those results were good but not record-setting, and they are rarely cited.
4. The complexity table in the paper is worth knowing. Self-attention costs O(n^2 d) per layer with O(1) sequential operations and a maximum path length of O(1) between any two positions.
5. A recurrent layer costs O(n d^2) with O(n) sequential operations and a maximum path length of O(n).
6. So self-attention is cheaper than recurrence whenever n is less than d, which was true for translation sentences in 2017, where n was around 30 and d was
   512. The quadratic cost only became the dominant problem years later.
7. Established fact: the architecture, the results and the training times are all in the published paper and have been reproduced many times.
8. **The honest version:** the paper did not predict what the transformer became. It contains no mention of language modelling at scale, no scaling laws, no in-context learning and no suggestion that this design would generalize to images, audio, protein structure or code.

9. The final sentence of the paper's conclusion is about applying attention to images, audio and video, and about making generation less sequential. Those were reasonable next steps, not a prediction of GPT-3.
10. The name "transformer" was chosen by Jakob Uszkoreit, and by several published accounts the team considered other names first. That is anecdote, not documented in the paper.
11. Reproduction is easy today. The `tensor2tensor` library released alongside the paper contained the original code, and Alexander Rush's annotated walkthrough from Harvard in 2018 reimplements the paper in a few hundred lines.

48.10.6 WORDS - remember these

1. BLEU — a translation score based on word overlap — bilingual evaluation understudy, from Papineni and colleagues at IBM in 2002, scored 0 to 100.
2. WMT — the standard machine translation benchmark — the Workshop on Machine Translation shared task, with the 2014 English-German and English-French sets still used as reference points.
3. NeurIPS — the main machine learning conference — the Conference on Neural Information Processing Systems, called NIPS until 2018.
4. Encoder-decoder — one stack reads, another writes — the original transformer layout, with cross-attention linking the two.
5. Warmup — raising the learning rate slowly at the start — a schedule required by post-norm transformers, 4,000 steps in the 2017 paper.
6. Label smoothing — never demanding 100 per cent confidence — replacing the one-hot target with 0.9 on the correct class and the rest spread out.

48.11 The three architectures

48.11.1 PLAIN - in simple words

1. The 2017 paper had two stacks: an encoder that reads and a decoder that writes. The field then split them apart and used them separately.
2. **Encoder-only**: keep the reading stack, throw away the writing stack. No causal mask, so every word sees every other word in both directions.
3. That is BERT, from Google in October 2018. It is superb at understanding a fixed piece of text and cannot generate text at all.
4. Use it for classification, for search, for named-entity extraction, and for producing embedding vectors of documents.
5. **Decoder-only**: keep the writing stack, throw away the reading stack. Causal mask, so every word sees only what came before.
6. That is the GPT line, and essentially every chat model you have ever used.
7. **Encoder-decoder**: keep both, as in the original. The encoder reads the input, the decoder writes the output, and cross-attention connects them.
8. That is T5, and most dedicated translation and speech-recognition models.
9. Decoder-only won for general-purpose models, and the reason is worth stating plainly.
10. A decoder-only model can do everything by writing. Ask it to classify, and it writes the class name. Ask it to translate, and it writes the translation.
11. Every task becomes text in, text out, so one model and one training objective covers all of them.
12. It also trains on more signal per token, needs no separate cross-attention machinery, and gets a prediction target at every position.

48.11.2 PLAIN - a picture in your head

1. Think of three kinds of worker.
2. The encoder-only worker is a proofreader. Given a finished page, they read it forwards and backwards as many times as they like and tell you what it is about. They never write anything.
3. The decoder-only worker is a novelist writing live, one word at a time, never allowed to see the words they have not written yet.
4. The encoder-decoder worker is a translator. They read the whole source document freely, then write the translation one word at a time, glancing back at the source whenever they need to.
5. The novelist turned out to be the most useful, because you can ask a novelist to write a summary, a classification or a translation, and they will.

Where this comparison breaks: a proofreader could write if asked. BERT genuinely cannot, because it was never trained to predict a next token and its attention has no causal structure to build on. The limitation is architectural and training-based, not a matter of instruction.

48.11.3 PLAIN - a worked example

1. Real models, sorted by architecture:

Model	Year	Architecture
BERT	2018	Encoder-only
RoBERTa	2019	Encoder-only
DeBERTa	2020	Encoder-only
T5	2019	Encoder-decoder
BART	2019	Encoder-decoder
Whisper	2022	Encoder-decoder
GPT-2, GPT-3	2019, 2020	Decoder-only
Llama, Mistral	2023 on	Decoder-only
Qwen, Gemma	2023 on	Decoder-only
DeepSeek-V3	2024	Decoder-only

2. Sizes for the encoder models: BERT-base has 110 million parameters, 12 layers and d_model 768. BERT-large has 340 million, 24 layers and d_model 1024.
3. Notice that encoder-only models stopped getting bigger. Nobody built a 100-billion-parameter BERT, because the returns went to generative models.
4. Encoder-only models are still heavily used, and not as a legacy. Almost every semantic search system in production embeds text with a small encoder, because it is cheap, bidirectional and fast.
5. A concrete size comparison for a search system: an encoder producing a 384- dimensional embedding might have 22 million parameters and run in under a millisecond per short document on a CPU.

48.11.4 PLAIN - what is really happening inside

1. The architectural differences reduce to two switches: is attention masked, and is there a second stack with cross-attention.
2. Encoder-only: no mask, one stack, no cross-attention.
3. Decoder-only: causal mask, one stack, no cross-attention.
4. Encoder-decoder: no mask in the encoder, causal mask in the decoder, plus cross-attention layers in the decoder.
5. Everything else, the blocks, the residuals, the normalization, the feed-forward networks, is identical in all three.
6. So they are not three architectures in any deep sense. They are one architecture with different masking and different wiring.

7. The training objective differs too, and that matters more than the wiring.
8. Encoder-only models are trained with masked language modelling, which corrupts about 15 per cent of tokens, giving a learning signal at 15 per cent of positions.
9. Decoder-only models are trained with next-token prediction, which gives a signal at 100 per cent of positions.
10. Encoder-decoder models like T5 use span corruption, dropping contiguous runs of tokens and asking the decoder to reproduce them.
11. That efficiency difference compounds over trillions of tokens, and it is a large part of why the decoder-only line scaled further.

48.11.5 TECHNICAL - the engineer's version

1. T5, from Colin Raffel and colleagues at Google in October 2019, cast every task as text-to-text with a task prefix, and scaled from 60 million to 11 billion parameters.
2. T5 established the text-to-text framing that decoder-only models later adopted without needing the encoder.
3. The empirical comparison exists. The 2022 paper "What Language Model Architecture and Pretraining Objective Work Best for Zero-Shot Generalization?" by Thomas Wang and colleagues found that decoder-only with causal language modelling gave the best zero-shot performance after pretraining alone, while encoder-decoder with masked objectives was stronger after multitask fine-tuning.
4. So the decoder-only win is not absolute. It is a win under the specific regime that turned out to matter: pretrain once, then prompt.
5. Practical reasons decoder-only dominates in deployment:
    1. One stack means one set of key-value caches, so serving is simpler.
    2. The prompt and the generation share the same cache, so a long prompt costs one prefill pass and nothing more.
    3. Any input format can be expressed as text, so no schema changes are needed per task.
6. Encoder-decoder still wins where the input is genuinely a different modality or language from the output. Whisper, released by OpenAI in September 2022, is an audio encoder with a text decoder, and that split is natural.
7. Encoder-only remains standard for retrieval. The sentence-transformers library, from Nils Reimers and Iryna Gurevych in 2019, is the usual tool, and the MTEB leaderboard is the usual benchmark.
8. A hybrid worth naming: many modern retrieval systems now use decoder-only models with the causal mask removed for embedding, which works but requires adaptation training.

48.11.6 WORDS - remember these

1. Encoder-only — reads text both ways, cannot write — a bidirectional transformer stack trained with masked language modelling.
2. Decoder-only — writes text one token at a time — a causally masked stack trained with next-token prediction.
3. Encoder-decoder — reads one sequence, writes another — two stacks joined by cross-attention, the original 2017 layout.
4. Bidirectional — able to use both left and right context — attention with no causal mask.
5. Span corruption — deleting runs of words and asking for them back — the T5 pretraining objective, replacing spans with sentinel tokens.
6. Text-to-text — every task expressed as text in and text out — the T5 framing, inherited by all current chat models.

48.12 Generation: producing text one token at a time

48.12.1 PLAIN - in simple words

1. The model does not write a sentence. It writes one token, then starts again with that token appended.
2. The loop is simple. Feed in the prompt. Get a probability for every possible next token. Choose one. Append it. Feed in the whole thing again.
3. Stop when a special end-of-text token is chosen, or when a length limit is reached.
4. Every word you have ever seen a language model produce came out of this loop, one token at a time.
5. The interesting decision is step three: choose one. There are several ways.
6. **Greedy**: always take the highest-probability token. Fully deterministic and often dull and repetitive.
7. **Temperature**: divide all the logits by a number before softmax. Below 1 sharpens the distribution towards the top choice. Above 1 flattens it.
8. **Top-k**: keep only the k most likely tokens, throw the rest away, and sample from what remains after renormalizing.
9. **Top-p**, also called nucleus sampling: keep the smallest set of tokens whose probabilities add up to p, typically 0.9 or 0.95.
10. **Min-p**: keep tokens whose probability is at least some fraction of the top token's probability. It adapts to how confident the model is.
11. **Repetition and frequency penalties**: reduce the score of tokens that have already appeared, to stop loops.
12. **Beam search**: keep several candidate continuations alive at once and pick the best complete sequence at the end.
13. This is why the same prompt gives different answers. Unless you use greedy decoding, a random draw happens at every single token.

48.12.2 PLAIN - a picture in your head

1. Picture a weather forecaster who can only forecast one hour ahead.
2. To forecast a whole day, they forecast hour one, write it down, then forecast hour two assuming hour one happened exactly as written.
3. Errors compound. If hour three is wrong, hours four to twenty-four are built on a mistake and cannot recover, because the forecaster treats their own written output as fact.
4. That is autoregressive generation, and it explains why long outputs drift.
5. Temperature is the forecaster's willingness to name an unlikely outcome. At temperature 0 they always say the single most likely thing. At high temperature they will occasionally forecast snow in July.

Where this comparison breaks: the forecaster knows their earlier forecast was a guess. The model has no such flag. Once a token is in the context, it is indistinguishable from text you wrote yourself, which is exactly why a model can confidently build on something it invented three sentences ago.

48.12.3 PLAIN - a worked example

1. Take a real five-way choice. The model has produced "the cat sat on the" and these are the top five candidate logits:

```
token    logit
mat       4.0
floor     3.2
chair     2.5
roof      1.1
moon      0.4
```

2. Softmax at temperature 1.0 gives:

```
mat    0.5699    floor  0.2561    chair  0.1272
roof   0.0314    moon   0.0156
```

3. Temperature divides every logit by T before the softmax. Here is the same distribution at several temperatures, computed exactly:

Token	T=0.5	T=1.0	T=2.0
mat	0.7965	0.5699	0.3933
floor	0.1608	0.2561	0.2636
chair	0.0397	0.1272	0.1858
roof	0.0024	0.0314	0.0923
moon	0.0006	0.0156	0.0650

4. At T = 0.5 the top token has 79.65 per cent, and "moon" has 0.06 per cent. At T = 2.0 the top token is down to 39.33 per cent and "moon" is up to 6.5 per cent, a hundredfold increase.
5. At T = 0.1 the top token reaches 0.9997, which is greedy decoding in all but name. Temperature 0 is usually implemented as exactly greedy, since dividing by zero is undefined.
6. **Top-k with k = 2**: keep "mat" and "floor" only. Renormalize 0.5699 and 0.2561 by their sum 0.8260, giving 0.6900 and 0.3100. "chair" is now impossible.
7. **Top-p with p = 0.9**: accumulate from the top. 0.5699, then 0.8260, then 0.9531 which passes 0.9. So three tokens are kept, renormalized to 0.5979, 0.2687 and 0.1334.
8. Note that top-p with 0.9 and 0.95 give the same set here, because the cumulative sum jumps from 0.826 to 0.953 in one step.
9. **Min-p with 0.1**: the threshold is 0.1 times the top probability, so 0.1 x 0.5699 = 0.057. Tokens above that are "mat", "floor" and "chair". Same set as top-p here.
10. The methods differ on a peaked distribution. If the top token has 0.9947 and the rest share 0.0053, min-p keeps only "mat", because the threshold is 0.0995 and nothing else clears it. Top-p at 0.9 also keeps only "mat".
11. On a flat distribution where the top is 0.2419, min-p's threshold is 0.0242 and all five survive, which is the point: the cut adapts to confidence.
12. **Repetition penalty of 1.2** on a token already generated divides its logit by 1.2 when positive. "mat" goes from 4.0 to 3.3333, and its probability falls from 0.5699 to 0.4048.
13. **Frequency penalty of 0.5** subtracts 0.5 for each previous occurrence. If "mat" appeared twice, its logit drops from 4.0 to 3.0 and "floor" becomes the most likely token at 0.4002.

48.12.4 PLAIN - what is really happening inside

1. Generation has two phases, and they behave completely differently.
2. **Prefill**: the whole prompt goes through the model in one pass. All prompt positions are computed in parallel. This is compute-bound and fast per token.
3. **Decode**: each new token requires a full forward pass over one position. The model reads every weight in the file to produce one token.
4. That is why decode is memory-bandwidth-bound. For an 8-billion model at 16-bit, each token requires reading 16 GB of weights from memory.
5. On a card with 2,000 GB/s of memory bandwidth, that sets a hard ceiling of about 125 tokens per second for a single sequence, before any other cost.
6. Beam search keeps b partial sequences alive, extends each by every candidate token, keeps the b best by total log probability, and repeats.
7. It optimizes for the highest-probability complete sequence, which is exactly what you want in translation, where there is one correct output.
8. It is wrong for chat for two reasons. The highest-probability continuation of open-ended text is bland and repetitive, a result documented by Ari Holtzman and colleagues in 2019. And beam search costs b times the compute and b times the key-value cache.

9. So translation systems use beam 4 or 5. Chat systems use sampling with temperature and top-p, and never beam search.
10. Why does the same prompt give different answers? Three separate reasons.
11. First, sampling. Every token is a random draw, so different random seeds give different text.
12. Second, even at temperature 0, floating-point addition is not associative. Changing the batch size changes the order of reductions inside the matrix multiplications, which changes the last bits of the logits, which can flip a near-tie between two tokens.
13. Third, providers change models, system prompts and serving configurations without notice. Determinism across dates is not offered by any major API.

48.12.5 TECHNICAL - the engineer's version

1. The autoregressive factorization: P(x_1 ... x_n) = product over t of P(x_t given x_1 ... x_{t-1}). Every generation method is a search or sampling strategy over this factorization.
2. Temperature applies as softmax(z / T). It is a monotonic transform, so it never changes the ranking, only the gaps.
3. Method origins:

Method	Year	Source
Beam search	pre-1980s	speech recognition
Top-k	2018	Fan, Lewis, Dauphin
Top-p (nucleus)	2019	Holtzman and colleagues
Repetition penalty	2019	Keskar and colleagues, CTRL
Min-p	2024	Nguyen and colleagues

4. Typical production defaults as of 2026: temperature 0.7 to 1.0, top_p 0.9 to 1.0, top_k either 40 or disabled, repetition penalty 1.0 to 1.1. These are conventions, and every provider picks slightly different ones.
5. OpenAI's API exposes `frequency_penalty` and `presence_penalty`, both in the range -2.0 to 2.0, applied additively to logits. Presence penalty applies once if a token has appeared at all; frequency penalty scales with the count.
6. The Hugging Face `generate` method applies processors in a fixed order: penalties first, then temperature, then top-k, then top-p, then min-p, then the sample. Order matters and is worth checking when results surprise you.
7. Greedy decoding is `do_sample=False`. Setting temperature to 0 in most APIs is mapped internally to greedy.
8. Speculative decoding, from Yaniv Leviathan, Matan Kalman and Yossi Matias at Google in 2022 and independently from DeepMind, uses a small draft model to propose several tokens and the large model to verify them in one pass. It is exact, meaning the output distribution is unchanged, and gives two to three times speedup in practice.
9. Constrained decoding masks logits to force valid JSON or a regular expression. Libraries such as Outlines and llama.cpp's GBNF grammars do this by zeroing the probability of any token that would break the grammar.
10. Established fact: sampling parameters change output distribution in exactly the ways computed above. Active research: which defaults are best for which task. Marketing claim: any statement that a specific temperature makes a model "more creative" in a measurable, general sense.

48.12.6 WORDS - remember these

1. Autoregressive generation — writing one token at a time, feeding each back in — sampling from P(x_t given all previous tokens) in a loop.

2. Greedy decoding — always take the most likely token — argmax over logits, fully deterministic given identical arithmetic.
3. Temperature — a dial that flattens or sharpens the choice — the divisor T in softmax(z / T), with T below 1 sharpening.
4. Top-k sampling — only consider the k best candidates — truncate to the k highest logits, renormalize, sample.
5. Top-p sampling — consider just enough candidates to cover p of the mass — nucleus sampling, keeping the smallest prefix of the sorted distribution summing to at least p.
6. Min-p sampling — keep anything within a fraction of the top choice — a threshold of p times max probability, adapting to model confidence.
7. Beam search — keep several drafts alive and pick the best — breadth-limited search over sequences, standard for translation, unsuitable for chat.
8. Prefill — processing the whole prompt at once — the parallel phase of inference, compute-bound.
9. Decode — producing one token per forward pass — the sequential phase, memory-bandwidth-bound.
10. Speculative decoding — a small model guesses, a big model checks — a draft-and-verify scheme that preserves the output distribution exactly.

48.13 The cost of attention: the quadratic problem

48.13.1 PLAIN - in simple words

1. Attention compares every token with every token. For n tokens that is n times n comparisons.
2. Double the length and the work quadruples. Ten times the length and the work goes up a hundredfold.
3. That is what "quadratic" means, and it is the single biggest engineering problem in the architecture.
4. At 1,000 tokens it is a million comparisons, which is nothing.
5. At 1,000,000 tokens it is a trillion comparisons, which is an enormous amount for a single layer of a single head.
6. There is a second cost, separate and often larger in practice: the **key-value cache**.
7. When generating, the model has already computed the keys and values for every earlier token. Recomputing them every step would be wasteful, so they are stored.
8. That store grows with every token generated, and it must live in fast memory next to the chip.
9. For a 70-billion-parameter model, every token of context costs 320 kilobytes of cache. A 128,000-token conversation costs 40 gigabytes.
10. That is more than the model itself in some configurations, and it is why long-context serving is expensive.
11. There are four families of fix, and three of them are in production today.
12. Make the exact computation memory-smarter (FlashAttention). Share keys and values across heads (grouped-query attention). Restrict which tokens can be looked at (sliding window, sparse). Or replace attention with something cheaper (linear attention, state-space models).

48.13.2 PLAIN - a picture in your head

1. Picture a party where every guest must shake hands with every other guest.
2. With 10 guests that is 45 handshakes. Manageable.
3. With 1,000 guests it is 499,500. With 100,000 guests it is about 5 billion.
4. The number of guests grew by a factor of 100 and the handshakes grew by a factor of 10,000.
5. Nobody solves this by shaking hands faster. You solve it by changing the rules: only shake hands with the nearest twenty people, or shake hands in small groups, or appoint representatives.
6. Every one of those is a real technique in the list of fixes.

Where this comparison breaks: a handshake is symmetric and attention is not. Token 5 attending to token 2 is a different computation from token 2 attending to token 5, and in a causal model only one of the two

happens at all. So the true count is n(n+1)/2, not n(n-1)/2, and the arithmetic differs by a factor of about two rather than the shape being different.

48.13.3 PLAIN - a worked example

1. The score matrix has n squared entries. Here is what that means, with the memory it would take at 16 bits per number, for a single head in a single layer:

Tokens	Entries	Memory, one head
1,024	1,048,576	2 MB
32,768	1,073,741,824	2,048 MB
131,072	17,179,869,184	32,768 MB
1,048,576	1,099,511,627,776	2,097,152 MB

2. Read the last row again. Two terabytes, for one head, in one layer, for one sequence.
3. A model like Llama 3.1 8B has 32 heads in each of 32 layers. If you tried to store the score matrices for a 131,072-token sequence in one layer, that is 32 times 32,768 MB, which is 1,024 GiB.
4. No accelerator has that memory. This is why the naive implementation simply cannot run at long context, and why FlashAttention exists.
5. Now the cost of attention relative to the rest of the model. The score computation costs about 4 n squared d operations per layer, and the feed-forward network costs about 16 n d squared.
6. Setting those equal gives a crossover at n = 4d. For Llama 3.1 8B with d = 4096, that is 16,384 tokens.
7. Below 16,384 tokens the feed-forward network dominates the compute. Above it, attention does. Measured for that configuration:

Sequence length	Attention share
512	2 per cent of FFN
4,096	19 per cent of FFN
24,576	114 per cent of FFN
131,072	609 per cent of FFN

8. Now the key-value cache, computed exactly. The formula is 2 x layers x kv_heads x head_dim x bytes, per token.
9. For Llama 3.1 8B: 2 x 32 x 8 x 128 x 2 = 131,072 bytes per token, which is exactly 128 KiB.

Context	Llama 3.1 8B	Llama 3.1 70B
2,048 tokens	0.25 GiB	0.62 GiB
32,768 tokens	4.00 GiB	10.00 GiB
131,072 tokens	16.00 GiB	40.00 GiB

10. Without grouped-query attention, using 32 key-value heads instead of 8, the 8B model's cache would be 512 KiB per token, so 64 GiB at full context instead of 16 GiB. That single design choice saves four times the memory.
11. Batching multiplies this. Serving 32 concurrent conversations of 8,192 tokens each on Llama 3.1 8B costs 32 GiB of cache, on top of 16 GB of weights.
12. That is the actual reason long context costs more per token in commercial APIs. It is memory, not arithmetic.

48.13.4 PLAIN - what is really happening inside

1. **FlashAttention** is the most important fix, and it is important to understand what it is not. It is not an approximation. Its output is exactly the same as the naive implementation, bit-for-bit equivalent up to floating-point reassociation.
2. The insight is that the n-by-n score matrix never needs to exist all at once.
3. The naive implementation writes the whole score matrix to the chip's main memory, reads it back for the softmax, writes it again, and reads it again for the multiply by V.
4. Those trips to main memory are the bottleneck, not the arithmetic. Chapter 22 covered why: a GPU's arithmetic units are far faster than its memory.
5. FlashAttention splits Q, K and V into tiles that fit in the chip's small fast on-chip memory, and computes the output tile by tile, keeping a running softmax normalizer as it goes.
6. The running softmax is possible because of an algebraic trick published by Maxim Milakov and Natalia Gimelshein in 2018, called online softmax, which lets you update a softmax result as new terms arrive.
7. The score matrix is therefore never stored. Memory use drops from order n squared to order n, and the speed rises because far fewer bytes move.
8. **Grouped-query attention** attacks the cache instead. Instead of 32 separate key and value heads, use 8, and let four query heads share each one.
9. The cache shrinks by exactly that ratio, and the quality loss is small.
10. **Sliding window attention** simply forbids looking back more than w tokens. Cost becomes linear in n, and information travels further only by hopping layer by layer, at w tokens per layer.
11. **Linear attention** rewrites the formula so the value sum can be accumulated without ever forming the score matrix, giving linear cost at the price of a weaker, approximate similarity function.

48.13.5 TECHNICAL - the engineer's version

1. FlashAttention, from Tri Dao, Daniel Fu, Stefano Ermon, Atri Rudra and Christopher Re, May 2022. FlashAttention-2 from Tri Dao, July 2023, improved work partitioning. FlashAttention-3, July 2024, targets Hopper-class hardware with asynchrony and low-precision paths.
2. Complexity: standard attention uses O(n^2) memory. FlashAttention uses O(n) and reduces high-bandwidth-memory accesses from O(n^2 d) to about O(n^2 d^2 / M), where M is on-chip SRAM size. Reported speedups on training are roughly 2x to 4x end to end.
3. It recomputes the attention matrix during the backward pass rather than storing it, trading a little arithmetic for a large memory saving. This is gradient checkpointing applied at kernel level.
4. Multi-query attention, from Noam Shazeer in November 2019, uses one key-value head for all query heads. Grouped-query attention, from Joshua Ainslie and colleagues at Google in May 2023, interpolates between the two.

Scheme	KV heads (32 q heads)	Cache size
Multi-head (MHA)	32	1.0x
Grouped-query (GQA)	8	0.25x
Multi-query (MQA)	1	0.031x

5. Multi-head latent attention, introduced in DeepSeek-V2 in 2024 and used in DeepSeek-V3, compresses keys and values into a low-rank latent vector and reconstructs them, cutting cache further than GQA at comparable quality.
6. Sliding window attention: Longformer, from Iz Beltagy, Matthew Peters and Arman Cohan in April 2020, combined a local window with a few global tokens. Mistral 7B, October 2023, used a 4,096-token window with 32 layers, giving a theoretical receptive field of 131,072 tokens.
7. Sparse attention: the Sparse Transformer, from Rewon Child, Scott Gray, Alec Radford and Ilya Sutskever in April 2019, used strided and fixed patterns to reach O(n sqrt(n)). BigBird, from Manzil

Zaheer and colleagues in July 2020, combined window, global and random attention and proved it is a universal approximator.

8. Linear attention: Angelos Katharopoulos and colleagues in June 2020 showed that replacing softmax with a kernel feature map lets you reorder the multiplications to get O(n) cost, and that the resulting model is a recurrent network at inference. The paper is titled "Transformers are RNNs".
9. Performer, from Krzysztof Choromanski and colleagues in September 2020, approximates the softmax kernel with random features. Reformer, from Nikita Kitaev, Lukasz Kaiser and Anselm Levskaya in January 2020, used locality-sensitive hashing.
10. **The honest version:** the approximate methods have largely not displaced exact attention in frontier models. Exact attention plus FlashAttention plus GQA is what production systems actually run as of 2026. The approximations remain valuable in specific niches.
11. State-space models are the serious current alternative. Mamba, from Albert Gu and Tri Dao in December 2023, uses a selective state-space layer with linear cost in sequence length and a constant-size state, so there is no growing key-value cache at all.
12. Mamba-2 followed in 2024 and showed a formal duality between state-space models and a restricted form of attention. Jamba, from AI21 Labs in March 2024, is a production hybrid interleaving Mamba and attention layers.
13. Active research, stated plainly: hybrid models are being trained at scale and show competitive quality with much better long-context economics. Whether they displace attention entirely is not known. Pure state-space models have measured weaknesses on tasks requiring precise recall of arbitrary earlier tokens, which is exactly what attention is good at.
14. Observation tooling: `nvidia-smi` shows memory use, but for cache accounting the useful figures come from the serving framework. vLLM, which implements PagedAttention from Woosuk Kwon and colleagues in 2023, reports cache block usage directly and manages the cache in fixed-size pages like an operating system manages virtual memory.

48.13.6 WORDS - remember these

1. Quadratic complexity — work grows with the square of the length — O(n^2) in sequence length for the score matrix and the weighted sum.
2. KV cache — stored keys and values so earlier tokens are not recomputed — 2 x layers x kv_heads x head_dim x bytes per token of context.
3. FlashAttention — computing attention without ever storing the score matrix — an exact, IO-aware tiled kernel with online softmax and backward recomputation.
4. Online softmax — updating a softmax as new terms arrive — a rescaling recurrence that avoids a second pass over the scores.
5. Multi-query attention — one key-value head shared by all query heads — MQA, cutting cache by the number of heads at some quality cost.
6. Grouped-query attention — a few key-value heads shared in groups — GQA, the standard compromise, 8 KV heads for 32 query heads in Llama 3.
7. Sliding window attention — only look back a fixed distance — local attention with window w, giving linear cost and a layer-multiplied receptive field.
8. State-space model — a linear recurrence with a fixed-size state — the Mamba family, linear in sequence length with no growing cache.
9. PagedAttention — managing the cache in fixed-size blocks — the vLLM technique that removes cache fragmentation, modelled on virtual memory paging.

48.14 Mixture of experts: paying for a big model, running a small one

48.14.1 PLAIN - in simple words

1. Every transformer block has two halves. Attention mixes information between tokens. The feed-forward network then thinks about each token on its own.
2. In an ordinary model there is exactly one feed-forward network per block, and every token goes through it. That block holds most of the model's numbers.
3. A **mixture of experts** model replaces that one feed-forward network with many copies of it, called **experts**, sitting side by side.
4. A tiny extra network called the **router** looks at each token and picks a small number of experts for it. Usually two, four or eight out of dozens.
5. Only the chosen experts run. The rest sit still and cost nothing in arithmetic for that token.
6. So the model is enormous when you count its numbers, and small when you count the work done for any one token.
7. This gives two different parameter counts, and the difference between them is the whole point of the idea.
8. **Total parameters** is every number in the file. It sets how much memory you need and how much a copy costs to store and to move.
9. **Active parameters** is how many numbers actually take part in producing one token. It sets the speed, the electricity bill and the training cost.
10. A real example: DeepSeek-V3, released December 2024, holds 671 billion parameters in total and uses 37 billion for each token. That is 5.5 per cent.
11. So it costs as much arithmetic per token as a 37-billion model, while holding the knowledge capacity of something eighteen times larger.
12. Nothing is free. You must keep all 671 billion numbers loaded, because you cannot know in advance which experts the next token will want.
13. That is the trade in one line: you pay in memory, and you are paid back in speed.
14. There is one nasty problem. Left alone, the router learns to send almost every token to the same two or three favourite experts, and the rest die.
15. Fixing that needs a deliberate extra pressure during training, and most of the engineering in this area is about exactly that.

48.14.2 PLAIN - a picture in your head

1. Picture a large hospital with 64 specialist doctors and one triage nurse at the door.
2. You walk in. The nurse spends two seconds on you and sends you to two of the 64 doctors. You see those two. The other 62 never learn you existed.
3. The hospital still has to employ all 64, pay all 64, and keep 64 consulting rooms heated. That is the memory cost.
4. But your visit only occupies two doctors' time. That is the compute cost.
5. Now the failure. Suppose the nurse sends nearly everyone to Doctor Smith because Doctor Smith seemed good on the first day.
6. Doctor Smith sees thousands of patients and gets very good. The other 63 see nobody and stay bad. So the nurse sends even more people to Smith.
7. You have not built a hospital with 64 specialists. You have built an expensive building containing one overworked doctor.
8. To stop this the hospital management adds a rule: the nurse is penalized when the workload is uneven, whatever the nurse thinks is best.
9. That rule is real, and it has a name in the papers: a load balancing loss.

Where this comparison breaks: real experts are not topic specialists, and this is the single most over-sold part of the idea. The Mixtral paper of January 2024 looked for topic-based routing and did not find it. Tokens about medicine do not reliably go to a "medicine expert". What routing does correlate with is much duller: token identity, position, and surface patterns such as indentation in code. Also, a patient

sees one doctor and gets one opinion. A token gets a weighted blend of two experts' outputs, added together in proportion to the router's confidence.

48.14.3 PLAIN - a worked example

1. We will build a mixture-of-experts model from a normal one, count the parameters exactly, and land on a real published model.
2. Start with the dense shape used earlier in this chapter: d_model 4096, feed forward width 14,336, SwiGLU with three matrices, 32 layers, 8 key-value heads, vocabulary 32,000.
3. One feed-forward network is 3 x 4096 x 14336 = 176,160,768 parameters.
4. Attention in one layer is 41,943,040 parameters. Two norms are 8,192.
5. Now replace the single feed-forward network with 8 copies, and add a router that maps 4096 numbers to 8 scores: 4096 x 8 = 32,768 parameters.
6. The expert bank in one layer is 8 x 176,160,768 = 1,409,286,144.
7. One layer therefore holds:

```
attention        41,943,040
experts (8)   1,409,286,144
router               32,768
two norms             8,192
layer total   1,451,270,144
```

8. Thirty-two layers give 46,440,644,608. Add the two vocabulary tables, 2 x 32,000 x 4,096 = 262,144,000, and a final norm of 4,096.
9. Total: 46,702,792,704 parameters, which is 46.7 billion.
10. That is Mixtral 8x7B, published by Mistral AI in December 2023, whose stated size is 46.7B. The arithmetic reproduces the published number exactly.
11. Now count what one token actually touches, with top-2 routing.

```
attention             41,943,040
2 experts of 8       352,321,536
router                    32,768
two norms                  8,192
per layer active     394,305,536
x 32 layers       12,617,777,152
+ two vocab tables   262,144,000
active total      12,879,921,152
```

12. That is 12.9 billion active parameters, which is also exactly the published figure. The ratio is 46.7 to 12.9, or 3.6 to 1.
13. Note what did not change: attention is dense. Every token runs the full attention block. Only the feed-forward half is sparse.
14. Now push the idea harder. Cut each expert into 8 narrower ones, so 64 experts, and route to 8 instead of 2.
15. The arithmetic per token is identical, because 8 narrow experts equal 2 wide ones. But the number of possible expert combinations explodes.
16. Choosing 2 from 8 gives 28 combinations. Choosing 8 from 64 gives 4,426,165,368. That is the argument for fine-grained experts.
17. Here is what real models do, as published by each laboratory:

Model	Total params	Active per token
Mixtral 8x7B, 2023	46.7B	12.9B
DBRX, 2024	132B	36B
Mixtral 8x22B, 2024	141B	39B

Model	Total params	Active per token
Grok-1, 2024	314B	about 79B
DeepSeek-V3, 2024	671B	37B
Llama 4 Scout, 2025	109B	17B
Qwen3-235B-A22B, 2025	235B	22B
Llama 4 Maverick, 2025	400B	17B
gpt-oss-120b, 2025	117B	5.1B
Kimi K2, 2025	1,000B	32B

18. Now the same table as one number, the fraction of the model used per token, sorted by date. The trend is the story:

Model	Year	Active fraction
Mixtral 8x7B	2023	27.6 per cent
Grok-1	2024	25.0 per cent
DeepSeek-V3	2024	5.5 per cent
Llama 4 Scout	2025	15.6 per cent
Qwen3-235B-A22B	2025	9.4 per cent
gpt-oss-120b	2025	4.4 per cent
Kimi K2	2025	3.2 per cent

19. Models are getting sparser. In 2023 a mixture-of-experts model used about a quarter of itself per token. By 2025, three per cent was normal.
20. Finally the memory bill. Mixtral 8x7B at 16 bits is 93.4 gigabytes of weights, which does not fit on one 80-gigabyte accelerator.
21. A dense model of the same speed, 12.9 billion parameters, would be 25.8 gigabytes and would fit on one card with room for a cache.
22. So you pay roughly 3.6 times the memory for roughly 1 times the arithmetic. Whether that is a good deal depends entirely on how many users you serve.

48.14.4 PLAIN - what is really happening inside

1. Here is the whole mechanism for one token at one layer.

```
      token vector x  (width d_model)
             |
      [ Router: x times W_r -> one score per expert ]
             |
      softmax, then keep the top 2
             |
      e5 (0.71)          e2 (0.29)
        |                   |
   [Expert 5]          [Expert 2]
        |                   |
     y5 x 0.71           y2 x 0.29
         \                 /
          +---- (add) ----+
                  |
              output  (width d_model)
```

2. The router is one small matrix, of shape d_model by number of experts. For Mixtral that is 4096 by 8. It is trained by ordinary gradient descent.
3. It produces one score per expert. A softmax turns those into probabilities.

4. **Top-k gating** keeps the k highest, throws the rest away, and renormalizes the survivors so they add to 1.
5. Only the surviving experts are run. Their outputs are multiplied by their renormalized weights and added together.
6. The gate weight is what makes this trainable. Without it, the choice of expert would be a hard, non-differentiable decision with no gradient.
7. With it, the gradient can flow back through the weight and tell the router "this expert helped, raise its score" or "this one hurt, lower it".
8. Now the collapse. The router and the experts learn at the same time, which creates a feedback loop.
9. An expert that gets slightly more tokens early gets more gradient updates, improves faster, produces better outputs, and so earns higher router scores.
10. That earns it more tokens still. Within a few thousand steps a handful of experts absorb nearly everything and the others are effectively dead weight.
11. The standard fix is a **load balancing loss**: an extra number added to the training loss that is smallest when every expert gets an equal share.
12. It pushes against the router's preference. The model is being told, in effect, "spread the work, even if you would rather not".
13. There is a second, purely practical constraint. On real hardware every expert is given a fixed-size buffer, called its **capacity**.
14. If more tokens are routed to an expert than its capacity allows, the extra tokens are **dropped**: they skip the feed-forward network entirely and pass through on the residual connection only.
15. Dropped tokens are not an error. They are a deliberate trade that keeps the tensor shapes fixed so the hardware stays efficient.
16. Finally, where the experts physically live. A 671-billion-parameter model does not fit on one accelerator, so the experts are split across machines.
17. That is **expert parallelism**. Each device holds a few experts. Every token must be shipped to whichever devices hold its chosen experts, and the results shipped back.
18. So each mixture-of-experts layer contains two network exchanges: one to dispatch tokens, one to collect answers. On a slow network this, and not the arithmetic, becomes the limit.

48.14.5 TECHNICAL - the engineer's version

1. History. The idea is from Robert Jacobs, Michael Jordan, Steven Nowlan and Geoffrey Hinton, "Adaptive Mixtures of Local Experts", Neural Computation,

1991. It predates the transformer by twenty-six years.

2. Noam Shazeer and colleagues, including Hinton and Jeff Dean, revived it for deep learning in January 2017 with "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer", reaching 137 billion parameters in an LSTM, with noisy top-k gating.
3. GShard, from Dmitry Lepikhin and colleagues at Google in June 2020, put MoE layers inside a transformer and trained a 600-billion-parameter translation model with automatic sharding.
4. Switch Transformer, from William Fedus, Barret Zoph and Noam Shazeer, January 2021, simplified routing to top-1 and scaled to 1.6 trillion parameters. Switch-C matched T5-Large's compute per sequence at 2,000 times the size.
5. GLaM, from Nan Du and colleagues at Google, December 2021: 1.2 trillion total parameters, about 97 billion active per token, 64 experts per MoE layer, top-2. It reported better zero-shot quality than GPT-3 at about one third of the training energy.
6. ST-MoE, from Barret Zoph and colleagues, February 2022, introduced the router z-loss and produced ST-MoE-32B at 269 billion parameters.
7. Routing mathematics. With router matrix W_r, gate g = softmax(x W_r), top-k index set T, and renormalized weights g_i / sum_{j in T} g_j:

```
scores   = x @ W_r                    # [n_tok, n_exp]
```

```
probs    = softmax(scores, dim=-1)
val, idx = topk(probs, k)             # weights, expert ids
val      = val / val.sum(-1, keepdim=True)
y = zeros_like(x)
for slot in range(k):
    for e in range(n_exp):
        m = idx[:, slot] == e
        y[m] += val[m, slot, None] * expert[e](x[m])
```

8. The Switch Transformer auxiliary loss, which is still the default in most frameworks:

```
f_i = fraction of tokens in the batch routed to expert i
P_i = mean router probability assigned to expert i
L_aux = alpha * N * sum over i of ( f_i * P_i )
alpha = 0.01,  N = number of experts
minimum value 1.0, reached when all f_i = P_i = 1/N
```

9. The product f_i P_i is the trick. f_i is a count and has no gradient. P_i is differentiable. Multiplying them gives the router a gradient proportional to how overloaded each expert already is.
10. Router z-loss, from ST-MoE: the mean of the squared log-sum-exp of the router logits, with coefficient 0.001. It keeps logits small, which improves numerical stability in bfloat16 and reduces round-off in the softmax.
11. **Expert capacity** = (tokens per batch / number of experts) x capacity factor. Typical capacity factors are 1.0 to 1.25 in training and higher at evaluation. Overflow tokens are dropped to the residual.
12. Expert Choice routing, from Yanqi Zhou and colleagues at Google in 2022, inverts the problem: each expert picks its top tokens instead of each token picking its experts. Balance is then perfect by construction, but the method is not causal-safe for autoregressive decoding without care.
13. MegaBlocks, from Trevor Gale and colleagues in 2022, reformulates MoE as block-sparse matrix multiplication so no token is ever dropped and no padding is needed. It is used in several production stacks.
14. DeepSeek-V3, December 2024, made two notable changes. **Fine-grained experts**: 256 routed experts of intermediate width 2048, top-8, plus one **shared expert** that every token always uses, to hold common knowledge.
15. And **auxiliary-loss-free load balancing**: instead of an extra loss term, a per-expert bias is added to the routing scores and adjusted between steps, raised for underloaded experts and lowered for overloaded ones. The bias affects selection only, never the gate weight, so it adds no gradient interference.
16. Its reported training cost was about 2.788 million H800 GPU-hours on 14.8 trillion tokens. The compute rule C is about 6ND, covered in Chapter 49, uses the ACTIVE parameter count. That substitution is the entire economic case for MoE.
17. Expert parallelism. Experts are sharded across devices and each MoE layer performs two all-to-all collectives, dispatch and combine. Communication per token per layer is about 2 x k x d_model x bytes across the fabric.
18. DeepSeek-V3 bounded this with node-limited routing, capping each token at 4 nodes, and overlapped communication with computation using a scheme it calls DualPipe. Frameworks that implement expert parallelism include DeepSpeed-MoE from Samyam Rajbhandari and colleagues in 2022, Tutel from Microsoft, and Megatron-Core.
19. Serving arithmetic intensity. At batch size 1 you read only k experts. At batch size 256 the tokens collectively select nearly every expert, so you read the whole expert bank anyway. MoE therefore looks cheap at small batch and behaves closer to dense at large batch.

Setting	Weights read per step
Dense 13B, any batch	26 GB at bf16

Setting	Weights read per step
MoE 47B, batch 1	about 27 GB
MoE 47B, batch 256	up to 93 GB

20. **Established fact.** Sparse models reach a lower pretraining loss than dense models given equal training compute; this was measured by GShard, Switch and GLaM and has been reproduced many times. Active parameters set FLOPs and training cost. Total parameters set memory. All experts must be resident. Unbalanced routing occurs reliably without a balancing mechanism.
21. **Active research.** How to compare an MoE against a dense model fairly, and what the right ratio of total to active parameters is. Whether experts specialize in any human-meaningful way; the current evidence is weak. Routing without auxiliary losses. Expert merging and pruning after training. Whether the shared-expert design is genuinely better or merely convenient.
22. **Vendor claim.** Headline parameter counts. A "1 trillion parameter model" that activates 32 billion is not equivalent to a dense trillion-parameter model, and no laboratory has demonstrated that it is. A common informal rule of thumb places MoE quality near the geometric mean of active and total parameters, which for Mixtral 8x7B would be about 24.5 billion; this is a heuristic, not a measured law, and should not be quoted as one.
23. Also vendor claim, by omission: GPT-4 has been widely reported to use a mixture-of-experts design since 2023, but OpenAI has never confirmed its architecture or parameter count. Treat any specific number you see for it as unverified.
24. Observation. For an open-weights model, `safetensors` index files list every expert tensor by name, so total parameters are directly countable. Active parameters must be derived from the config: `num_local_experts` and `num_experts_per_tok` in a Hugging Face config.json give k and N. Serving frameworks such as vLLM and SGLang report expert-parallel placement and per-expert token counts, which is how you detect a collapsed router in production.

48.14.6 WORDS - remember these

1. Mixture of experts — many small networks with a chooser in front — MoE, replacing a dense feed-forward layer with N experts and a learned router.
2. Expert — one of the interchangeable feed-forward networks — a standard FFN or SwiGLU block, one of N per MoE layer.
3. Router, or gate — the small network that picks experts — a d_model by N matrix followed by softmax and top-k selection.
4. Top-k gating — take the k best experts and ignore the rest — hard sparse selection with renormalized gate weights, k typically 1, 2 or 8.
5. Total parameters — every number in the model file — the figure that sets memory, storage and download size.
6. Active parameters — the numbers used for one token — the figure that sets FLOPs per token, latency and training compute in C is about 6ND.
7. Load balancing loss — an extra penalty for using experts unevenly — the Switch auxiliary loss alpha N sum f_i P_i, alpha typically 0.01.
8. Router z-loss — a penalty that keeps router scores small — the mean squared log-sum-exp of router logits, coefficient about 0.001, for stability.
9. Expert capacity — the fixed number of tokens an expert will accept — buffer size = tokens/experts x capacity factor; overflow tokens are dropped.
10. Token dropping — skipping the feed-forward step for overflow tokens — those tokens pass through on the residual connection unchanged.
11. Shared expert — an expert every token always uses — the DeepSeek design for holding knowledge common to all inputs, alongside the routed experts.

12. Fine-grained experts — many narrow experts instead of few wide ones — same FLOPs, far more possible combinations, 64-choose-8 versus 8-choose-2.
13. Expert parallelism — experts split across devices — sharding along the expert axis, requiring two all-to-all collectives per MoE layer.
14. Sparse upcycling — turning a trained dense model into an MoE — copying the trained FFN into N experts and continuing training from there.

48.15 Why one architecture does so many tasks

48.15.1 PLAIN - in simple words

1. This is the question most people actually want answered. One machine writes code, translates Hindi, summarizes a contract and holds a conversation. Why?
2. The short answer is that there is only one training task, and doing it well happens to require almost everything else.
3. The task is **next-token prediction**. Show the model some text, hide what comes next, make it guess, and correct it. Repeat trillions of times.
4. There is no list of skills. There is no separate summarizing module and no translation module. There is one number being pushed down: the prediction error.
5. Now the important argument. To predict text well, you must model whatever produced that text.
6. To finish "The capital of France is" you need a fact. Guessing needs knowledge, so knowledge lowers the error, so training installs knowledge.
7. To finish "for i in range(10):" you need to know what an indented block looks like. So the training installs the shape of Python.
8. To finish the last page of a detective novel you must have worked out who did it from the clues. So training rewards keeping track of a plot.
9. To finish a polite email you need to know how polite emails go. So training installs style and register.
10. Grammar, facts, arithmetic, tone, the structure of arguments, the layout of a table: every one of them lowers the prediction error, so every one of them gets learned.
11. Then the second half of the answer, which is easy to miss. Text on the internet already contains every task, written out in full.
12. An article is followed by its summary. An English sentence is followed by its French translation. A question is followed by an answer. A bug report is followed by a fix.
13. So when the text is arranged that way, predicting the next token IS doing the task. The task was never separate. It was always just more text.
14. That covers translation, summarizing and code. Then there is one behaviour that genuinely surprised the people who built it.
15. **In-context learning.** Put three worked examples in the prompt and the model does the fourth, for a task nobody trained it on and that may not exist anywhere in its training data.
16. Nothing inside the model changes. No number is updated. The examples sit in the input and the model behaves as if it had been taught. That was not designed. It was noticed.
17. Then **instruction following**: you write "Summarize this in three bullet points" and it obeys. Here we must be honest.
18. That is mostly not from pretraining. A raw pretrained model given that instruction is quite likely to continue with another instruction, because that is what usually follows an instruction on a web page.
19. Obedience is added afterwards, in a separate stage, using human-written examples and human preference data. Chapter 49 covers exactly how.
20. Finally, **emergence**: the claim that some abilities are absent in small models and appear suddenly in large ones. This one is genuinely disputed, and we will give both sides.

48.15.2 PLAIN - a picture in your head

1. Imagine a person locked in the world's largest library, playing one game, forever.
2. The game: open any book at any page, cover the next word, guess it, then uncover it and see if you were right. Score kept. No other instructions.
3. On day one they guess badly. Soon they learn that "the" is common, that a full stop is usually followed by a capital letter.
4. Keep going. Now they are playing on a chemistry textbook. To guess the word after "sodium reacts violently with" they have to learn chemistry.
5. Keep going. Now a French-English phrasebook, where every line is the same sentence twice. To win, they must learn to translate.
6. Keep going. Now the last page of a murder mystery. To guess the name after "the killer was" they must have followed the clues and solved it.
7. Nobody ever told them to learn chemistry, French or detective work. They learned all three because each one improved their score at one game.
8. That is the whole explanation of why one architecture does so many things. The tasks were never taught. They were the only way to win.

Where this comparison breaks: the person in the library knows they are playing a game, wants to win, and could stop. The model has none of that. It has no goal, no awareness that it is guessing, and no sense of being right or wrong. A person who does not know says so; the model always produces a full probability distribution over every possible next token, and something is always on top. Also, the person learns by understanding. The model learns by a mechanical adjustment of billions of numbers in the direction that lowers one error term.

48.15.3 PLAIN - a worked example

1. Here is how four different-looking jobs become the same job. Each block is just text, and the model only ever continues it.

```
[ translation ]
English: the sea otter is asleep
French:

[ summarizing ]
<a 900-word article>
TL;DR:

[ code ]
# return the nth Fibonacci number
def fib(n):

[ conversation ]
User: what is the boiling point of water?
Assistant:
```

2. In every case the model does the same thing: predict what comes next. The task lives entirely in how the text was arranged.
3. The "TL;DR:" trick is not made up. The GPT-2 paper of February 2019 used exactly that string to get summaries out of a model that had no summarizing training at all.
4. Now in-context learning. This is the format from the GPT-3 paper of May 2020, which is where the phrase "few-shot" entered common use:

```
Translate English to French.
sea otter => loutre de mer
peppermint => menthe poivree
```

```
plush giraffe => girafe peluche
cheese =>
```

5. The three examples are not training data. They are input. They are thrown away when the request ends. The model's weights are identical before and after.
6. It works anyway, and it works better with more examples. Measured on the TriviaQA question-answering set with the 175-billion-parameter GPT-3:

Setting	Examples in prompt	Accuracy
Zero-shot	0	64.3 per cent
One-shot	1	68.0 per cent
Few-shot	64	71.2 per cent

7. Seven points of accuracy, bought with nothing but text in the prompt. No training, no weight change, no gradient.
8. Now the honest correction about instructions. Take a base model, before any fine-tuning, and give it this:

```
Write a haiku about rain.
```

9. A base model may well continue like this, because that is what actually follows such a line on a real web page:

```
Write a haiku about rain.
Write a limerick about a cat.
Write a short story about a lighthouse.
Submit your entries by Friday.
```

10. It is not being unhelpful. It is predicting correctly. It has recognized a list of writing prompts and is continuing the list.
11. Turning that into a model that answers is a separate stage of work, with its own data and its own cost, described in Chapter 49.

48.15.4 PLAIN - what is really happening inside

1. One loss function drives everything. For each position, the model outputs a probability for every token in the vocabulary, and the loss is the negative logarithm of the probability it gave to the token that actually came next.
2. Averaged over the sequence, that single number is the whole training signal. There is nothing else. No task labels, no reward, no human in the loop.
3. Because the loss is per-position, one 8,192-token document gives 8,192 separate lessons in one pass. That is why the objective is so efficient.
4. **Implicit multitask learning** is the mechanism behind the range. The corpus contains question-answer pairs, parallel translations, code with comments, articles with headlines, arguments with rebuttals.
5. Each of those is a task demonstration that happens to be written down as ordinary text. The model never learns "this is translation". It learns that after a French colon, French words are likely.
6. The honest version: it is often said that models were never trained on these tasks. That is too strong. The tasks are present in the data in enormous quantity, just unlabelled. And since about 2023 laboratories deliberately mix instruction-shaped and textbook-shaped data into pretraining itself, so the clean story of pure raw web text is out of date.
7. In-context learning works differently, and we do have a partial mechanical account of it.
8. Earlier in this chapter we met **induction heads**: a pair of attention heads that between them find an earlier occurrence of the current token and copy what followed it.

9. That is a complete, if crude, pattern-completion machine built out of attention. Given "A B ... A" it predicts "B". Given three translation pairs it can find the pattern and continue it.
10. So few-shot prompting is not the model learning. It is the model reading. The examples are in the context, and attention can reach them.
11. That is why in-context learning vanishes the moment the examples fall out of the context window. Nothing was stored. Nothing was learned in the sense of a weight changing.
12. Instruction following is the part that is genuinely bolted on. The base model has all the capability and none of the disposition to use it on request.
13. Fine-tuning on instruction-and-answer pairs changes which continuation is likely, not what the model knows. It is a change of habit, not of knowledge.
14. That distinction matters, because it explains a common observation: fine tuning for helpfulness can make a model slightly worse at raw prediction while making it far more useful.

48.15.5 TECHNICAL - the engineer's version

1. The objective, exactly. Maximize the log-likelihood of the corpus under the chain rule of probability, one factor per token:

```
loss = - (1/T) * sum over t of log P(x_t | x_1 ... x_{t-1})
```

2. This is cross-entropy between the model's distribution and a one-hot target, in nats per token. Divide by ln 2 for bits per token.
3. Shannon's source coding theorem makes the connection to compression exact: the cross-entropy in bits per token is the size of the optimally compressed text under that model. Better prediction is literally better compression.

Method	Approximate bits per char
gzip on English prose	about 2.4
Shannon's 1951 human study	0.6 to 1.3
Large model, held-out text	under 1.0

4. Claude Shannon's "Prediction and Entropy of Printed English", 1951, estimated English at roughly 0.6 to 1.3 bits per character by having humans guess next letters. Current models are inside that band on ordinary prose. Treat the figures above as approximate; they depend heavily on the text.
5. Zero-shot task transfer was demonstrated by GPT-2, "Language Models are Unsupervised Multitask Learners", from Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei and Ilya Sutskever, February 2019, at 1.5 billion parameters.
6. Few-shot in-context learning was the headline result of GPT-3, "Language Models are Few-Shot Learners", from Tom Brown and colleagues, May 2020, at 175 billion parameters.
7. Instruction tuning as a named technique: FLAN, "Finetuned Language Models Are Zero-Shot Learners", Jason Wei and colleagues at Google, September 2021; and T0, "Multitask Prompted Training Enables Zero-Shot Task Generalization", Victor Sanh and colleagues, October 2021.
8. Then InstructGPT, "Training language models to follow instructions with human feedback", Long Ouyang and colleagues at OpenAI, March 2022. Its key measured result: human labellers preferred the outputs of the 1.3-billion-parameter InstructGPT model over the 175-billion-parameter GPT-3, a hundredfold size difference overturned by post-training alone.
9. That single result is the strongest available evidence that instruction following is added, not emergent. Full detail is in Chapter 49.
10. Now emergence, stated fairly, because this is contested.
11. **The claim.** "Emergent Abilities of Large Language Models", Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph and others, June 2022, published in TMLR. Their definition: an ability is

emergent if it is not present in smaller models but is present in larger models, and cannot be predicted by extrapolating from smaller models.

12. Reported examples include multi-digit arithmetic, word unscrambling and some of the harder tasks in BIG-bench, the 204-task benchmark released by Aarohi Srivastava and several hundred co-authors in June 2022, which itself reported "breakthrough" behaviour on some tasks.
13. **The critique.** "Are Emergent Abilities of Large Language Models a Mirage?", Rylan Schaeffer, Brando Miranda and Sanmi Koyejo of Stanford, April 2023. Their argument: nonlinear or discontinuous metrics produce apparent emergence, while linear or continuous metrics on the same models produce smooth, predictable curves.
14. Their strongest evidence is constructive: they induced apparent emergence in simple vision models, where nobody claims it exists, purely by choosing a harsh metric. And they showed most BIG-bench emergence disappears when exact-match accuracy is replaced by token edit distance.
15. Here is the artefact in arithmetic you can check yourself. Suppose per-token accuracy improves smoothly with scale, and the answer is 10 tokens long, and the metric is exact string match, so all 10 must be right:

Per-token accuracy	10-token exact match
0.50	0.1 per cent
0.70	2.8 per cent
0.80	10.7 per cent
0.90	34.9 per cent
0.95	59.9 per cent
0.99	90.4 per cent

16. The left column rises steadily. The right column sits near zero, then leaps. Plot the right column against compute on a log axis and you get a cliff, and nothing whatsoever happened inside the model.
17. **The reply to the critique**, given fairly. Three points. First, the critique does not show that scaling is safe or fully predictable; it shows that one class of measurement exaggerates sharpness.
18. Second, many things people actually care about really are all-or-nothing. A program compiles or it does not. A proof is valid or it is not. If the metric that matters to the user is discontinuous, the user's experience of a sudden capability jump is real, whatever the smooth underlying curve says.
19. Third, at least one genuine discontinuity has been found mechanistically, not metrically. "In-context Learning and Induction Heads", Catherine Olsson and colleagues at Anthropic, March 2022, identified a visible bump in the training loss curve where induction heads form, accompanied by a jump in in-context learning ability. That is a phase change in the model, not in the scoring.
20. Relatedly, "Grokking", from Alethea Power and colleagues at OpenAI in January 2022, showed genuine sudden generalization long after memorization on small algorithmic tasks. Sharp transitions do exist.
21. A later position, from Sheng Lu and colleagues at ACL 2024, argued that many claimed emergent abilities reduce to in-context learning plus instruction tuning rather than any new latent reasoning capacity.
22. **Established fact.** Next-token prediction is the sole pretraining objective. Pretraining loss falls as a smooth power law in compute, data and parameters, per Jared Kaplan and colleagues in January 2020 and Jordan Hoffmann and colleagues in March 2022. In-context learning is real and measurable. Instruction following comes overwhelmingly from post-training.
23. **Active research.** Why in-context learning works at all. One line of work, including Johannes von Oswald and colleagues in 2022, argues the forward pass implements something equivalent to gradient descent on the in-context examples; this is suggestive and not settled. Whether emergent abilities are real, artefactual, or both depending on the ability, is open, and reasonable researchers disagree in print.
24. **Marketing claim.** That emergence shows a model has begun to understand, reason or think. That is a philosophical claim dressed as a measurement. The "Sparks of AGI" paper from Microsoft in March

2023 was widely criticized on exactly this ground, including for evaluating an unreleased model version that others could not reproduce.

25. Also marketing: presenting a capability as spontaneous when it was in the fine-tuning mix. Without disclosure of the post-training data, which is almost never published, that distinction cannot be checked from outside.
26. Observation. Per-token loss and log-probabilities are the measurable quantities here, exposed as `log-probs` by most inference APIs and by `output_scores` in Hugging Face `generate`. Task-level claims are measured with harnesses such as EleutherAI's `lm-evaluation-harness` or Stanford's HELM, and the first thing to check in any emergence plot is which metric the y-axis uses.

48.15.6 WORDS - remember these

1. Next-token prediction — guess the next piece of text — autoregressive language modelling with a cross-entropy loss over the vocabulary.
2. Pretraining objective — the one thing the model is scored on — maximizing log-likelihood of the corpus under the chain rule of probability.
3. Cross-entropy loss — how surprised the model was by the true next token — minus the log probability assigned to the target, in nats per token.
4. Implicit multitask learning — tasks learned without being labelled as tasks — task demonstrations occurring naturally in the pretraining corpus.
5. In-context learning — learning from examples in the prompt, with no weight change — conditioning on demonstrations within a single forward pass.
6. Few-shot prompting — putting a handful of examples in the prompt — k-shot conditioning, popularized by the GPT-3 paper of May 2020.
7. Zero-shot — asking with no examples at all — task specification by instruction or format alone, demonstrated by GPT-2 in February 2019.
8. Base model — a model that has only been pretrained — a pure next-token predictor with no instruction tuning and no preference optimization.
9. Instruction tuning — teaching the model to do what it is asked — supervised fine-tuning on instruction-and-response pairs, FLAN 2021, InstructGPT 2022.
10. Emergent ability — a skill said to appear only above some size — an ability absent in smaller models and present in larger ones, per Wei et al. 2022, with the sharpness disputed by Schaeffer et al. 2023.
11. Metric artefact — a jump caused by the ruler, not the thing measured — apparent discontinuity produced by exact-match or thresholded scoring.
12. Scaling law — the predictable fall in loss as you spend more — a power law in parameters, data and compute, per Kaplan 2020 and Hoffmann 2022.

48.98 Common wrong ideas

1. Wrong: attention means the model is focusing, the way a person concentrates. Right: attention is a weighted average. Every token computes a similarity score against every other token, those scores go through a softmax, and the result is used to mix value vectors. Nothing is ignored: after a softmax every weight is strictly greater than zero, so a "0.001" token still contributes. There is no spotlight, no effort and no choosing. The name is a 1990s metaphor from Bahdanau, Cho and Bengio's 2014 alignment work, and it has misled more readers than it has helped.
2. Wrong: the transformer understands grammar, because everything it writes is grammatical. Right: grammatical output is evidence that grammatical continuations had low prediction loss, nothing more. The model holds no parse tree, no rule for subject-verb agreement and no category called "verb". Probing studies do find syntax-correlated directions in the activations, and that is a genuine finding, but a correlated direction is not a rule. The honest test is behaviour on rare constructions, where models still fail in ways no speaker of the language would.

3. Wrong: a bigger context window means better memory. Right: the context window is the maximum input size for one request. Nothing persists between requests, and any apparent memory is your application resending the history. Accuracy also dips for material in the middle of a long input, and NVIDIA's RULER benchmark in 2024 found many models advertising 32,000 tokens performed well only at much shorter lengths. Longer also costs more: at 131,072 tokens the key-value cache for Llama 3.1 70B is 40 GiB, on top of the weights.
4. Wrong: it predicts words. Right: it predicts tokens, which are subword pieces from a fixed vocabulary, typically 32,000 to 256,000 entries. "Unbelievable" may be three tokens; a leading space is usually part of the token; a number may split in places no human would choose. And it does not predict one token, it produces a probability for every token in the vocabulary at once, and then a separate sampling step picks one.
5. Wrong: more heads means more intelligence. Right: heads split the same d_model between them. With d_model 512, going from 8 heads to 16 halves the head dimension from 64 to 32. Total parameters and total FLOPs are unchanged. The 2017 paper measured this directly and found 1 head was worse than 8, and 32 heads were also worse than 8. Head count is a shape choice with an interior optimum, not a quality dial.
6. Wrong: emergent abilities prove something new appeared inside the model. Right: this is disputed in print. Wei and colleagues reported sharp jumps in 2022; Schaeffer, Miranda and Koyejo at Stanford argued in 2023 that the sharpness is often produced by all-or-nothing metrics, and demonstrated the effect by manufacturing fake emergence in vision models. Some genuine phase changes do exist, notably the formation of induction heads found by Anthropic in 2022. The correct position is that the metric must be checked before any claim about the model is made.
7. Wrong: a 671-billion-parameter mixture-of-experts model is equivalent to a dense 671-billion model. Right: it does the arithmetic of a 37-billion model per token. You get the memory bill of the large model and the speed of the small one. No laboratory has shown the quality matches a dense model of the same total size, and the informal geometric-mean rule of thumb is a heuristic, not a measured law.
8. Wrong: attention is where the model's knowledge is stored, since attention is what makes it a transformer. Right: attention holds about 17 per cent of the parameters in Llama 3.1 8B; the feed-forward blocks hold about 70 per cent, rising to 81 per cent at 405B. Attention moves information between positions. The feed-forward blocks are where the storage happens.
9. Wrong: the transformer processes text left to right, like a person reading. Right: during training every position is computed at once in one matrix multiplication, which is the entire reason the architecture won. The causal mask enforces the ordering of information without enforcing an ordering of computation. Only generation is sequential, and that is a property of writing text one token at a time, not of the architecture.
10. Wrong: setting temperature to 0 makes the model correct. Right: it makes the model deterministic, which is a different thing. Greedy decoding picks the highest-probability token every time, which removes variation and often produces repetitive text. If the highest-probability continuation is wrong, temperature 0 guarantees you get the wrong answer every time instead of some of the time.

48.99 Chapter summary in 20 lines

1. Before 2017 sequences were handled by recurrent networks, which kept one hidden state and updated it once per word, so information from early words faded and gradients vanished or exploded across hundreds of steps.
2. The fatal flaw was not memory but order: word ten could not be computed until word nine finished, so a chip with tens of thousands of parallel units sat idle, and no amount of hardware fixed it.
3. Attention removes the order constraint. Every position looks at every other position at once, scores them for relevance, and takes a weighted average of what it finds. The whole sequence becomes one matrix multiplication.
4. Each token is projected into three roles: a query saying what it is looking for, a key advertising what it offers, and a value carrying what it passes on if selected. Three learned matrices, three different jobs.
5. The formula is softmax(QK^T / sqrt(d_k)) V. Dot products give similarity, division by sqrt(d_k) keeps the variance of the scores near 1 so the softmax does not saturate, softmax turns scores into weights that sum to 1, and the multiply by V collects the answer.
6. Without the sqrt(d_k) scaling, scores at d_k = 64 have standard deviation around 8, the softmax becomes nearly one-hot, and gradients through it fall towards zero. The scaling is not cosmetic.
7. A causal mask sets every score above the diagonal to minus infinity before the softmax, which makes those weights exactly zero. That is how a model is stopped from reading its own answer during training.
8. Multi-head attention splits d_model into h independent subspaces, runs attention in each, concatenates and projects. Parameters and FLOPs are unchanged; only the shapes differ. The 2017 paper measured 8 heads as better than 1 and better than 32.
9. Heads are not clean specialists. Some interpretable ones are real, notably induction heads that complete a repeated pattern and underlie in-context learning, but most heads resist a tidy description and many can be pruned.
10. Attention is permutation equivariant: shuffle the input and the output shuffles identically. It has no idea what order anything is in, so position must be injected deliberately.
11. Position schemes went sinusoidal in 2017, learned absolute in BERT and GPT, T5 relative bias in 2019, ALiBi in 2021, and rotary embedding in 2021. RoPE rotates queries and keys by an angle set by position, so scores depend on relative distance exactly, and it is the dominant choice as of 2026.
12. RoPE's base controls the slowest wavelength. Raising it from 10,000 to 500,000, as Llama 3.1 did in July 2024, is part of how a 131,072-token context is reached without the slowest frequency wrapping around.
13. After attention, each position passes alone through a feed-forward network that widens by roughly four times, applies a non-linearity such as SwiGLU, and narrows back. This block holds 70 to 81 per cent of a model's parameters and is where facts appear to be stored.
14. Residual connections add the input back to every sub-layer's output, giving gradients a clean path through depth and creating a residual stream that every block reads from and writes to. Normalization, LayerNorm in 2017 and RMSNorm today, keeps activation sizes stable; pre-norm replaced post-norm around 2019 and removed the need for learning-rate warmup.
15. The assembled model is: embed token ids, add or rotate in position, run N identical blocks, normalize once more, multiply by the unembedding matrix, and read a probability over the whole vocabulary at every position.
16. The 2017 paper, "Attention Is All You Need" by Ashish Vaswani and seven co-authors, reached 28.4 BLEU on WMT 2014 English-German with the big model, beating every previous system at one to two orders of magnitude less training compute. It never mentioned scaling laws or in-context learning.
17. Three architectures came out of it: encoder-only for reading, such as BERT; decoder-only for writing, which is every chat model today; and the original encoder-decoder, joined by cross-attention, which survives in translation and in T5-style text-to-text framing.
18. Generation is a loop: feed everything, get a distribution, choose one token, append, repeat. The choice

is greedy, or sampled with temperature, top-k, top-p or min-p. That sampling step, not the model, is why the same prompt gives different answers.

19. Attention costs O(n^2). At 131,072 tokens the score matrix for one head in one layer would be 32 GiB, so nobody stores it: FlashAttention computes the same result tile by tile without ever writing it out, grouped-query attention shrinks the key-value cache fourfold, and sliding windows, sparse patterns and state-space models attack the problem from other directions.
20. Mixture of experts splits total from active parameters, letting DeepSeek-V3 hold 671 billion numbers while using 37 billion per token; and the reason one architecture does so many jobs is that it is trained on one objective, next-token prediction, which rewards grammar, facts, style and reasoning alike, with in-context learning arriving unplanned, instruction following added afterwards by fine-tuning, and emergence still argued over.

49

CHAPTER

KB

How a Model Is Actually Made — Pretraining to Release

49.0 What this chapter gives you

1. You will be able to name every stage that turns raw web text into a shipped chat model, in order, and say what each stage costs and what it changes.
2. You will be able to explain where training data comes from, how it is cleaned, and why the legal position around it is genuinely unsettled.
3. You will be able to describe one training step exactly: forward pass, loss, backward pass, optimizer update, and say what a good loss curve looks like.
4. You will be able to use the formula C is about 6ND to estimate the compute of any announced model, and check it against the figures the lab published.
5. You will be able to explain scaling laws, what Chinchilla changed in 2022, and why labs deliberately ignore Chinchilla for small models today.
6. You will be able to say why a base model cannot be used as a chatbot, and what supervised fine-tuning and preference optimization each add.
7. You will be able to explain RLHF, reward hacking, DPO, constitutional methods and verifiable-reward training, and when each is used.
8. You will be able to open a model directory and say what every file is, what format it is in, and which formats can execute code when loaded.
9. You will be able to state precisely what "open weights", "open source", "open data" and "permissive licence" each mean, and tell them apart.
10. You will be able to read a model card and a licence and say what the publisher withheld, and what that omission tells you.

49.1 The pipeline in one view

49.1.1 PLAIN - in simple words

1. A chat model is not made in one step. It is made in a chain of stages.
2. Each stage takes the output of the one before and changes it.
3. Stage one is collecting text. Enormous amounts of text, mostly from the web.
4. Stage two is cleaning that text. Throwing away most of it, in fact.
5. Stage three is building the **tokenizer**, the fixed list of text pieces the model is allowed to see. Chapter 47 covered this in detail.
6. Stage four is **pretraining**. This is the long, expensive one. Weeks or months on thousands of machines, learning to predict the next piece of text.
7. What comes out of stage four is called a **base model**. It is not a chatbot. It continues text. It does not answer questions.

8. Stage five is **post-training**. This is where it learns to behave like an assistant: to answer, to follow instructions, to refuse harmful requests.
9. Post-training has parts of its own: showing it good examples, then teaching it which of two answers people prefer, then safety work.
10. Stage six is **evaluation**: measuring what the thing can and cannot do.
11. Stage seven is release: publishing the numbers, or putting them behind an interface, along with documents describing what was made.
12. The costs are wildly unequal. Pretraining is almost all of the money.
13. Post-training is a rounding error in compute, and yet it is what makes the difference between a strange text continuer and something you can talk to.
14. That imbalance is the single most useful fact in this chapter.

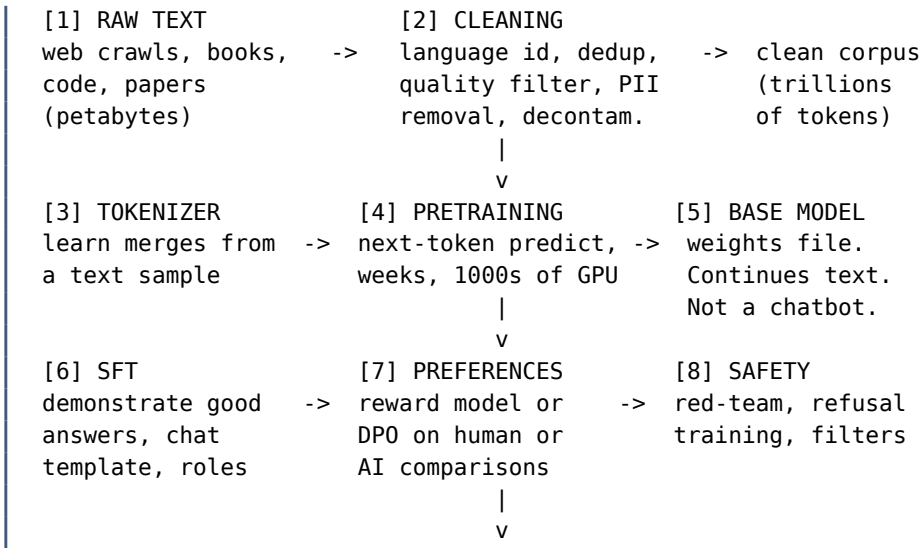
49.1.2 PLAIN - a picture in your head

1. Think of how a doctor is made.
2. First, years of general education. School, then a science degree. Broad, slow, expensive, and not aimed at any one job.
3. That is pretraining. Enormous input, no particular task, no bedside manner.
4. A person at the end of that stage knows a great deal of biology and can still be useless in a consulting room.
5. Then comes clinical training. Watching a senior doctor, copying how they greet a patient, how they explain bad news, when they say "I do not know".
6. That is supervised fine-tuning: learning the form of the job by imitation.
7. Then comes years of feedback. A supervisor says "that explanation was better than the one you gave yesterday". Slowly the style settles.
8. That is preference optimization: learning from comparisons, not from corrections.
9. Then exams and a licence before anyone is allowed to practise unsupervised.
10. That is evaluation and release.

Where this comparison breaks: a medical student keeps learning after graduation. A shipped model does not. Its numbers are frozen at release and never change again unless the lab trains a new version and publishes a new file. Also, a doctor's general education takes years and their clinical training takes years. In a model, the general education is 99 per cent of the cost and the clinical training is about 1 per cent. The proportions are nothing like a human career.

49.1.3 PLAIN - a worked example

1. Here is the whole chain as a diagram. Follow the arrows left to right.



```
[9] EVALUATION          [10] RELEASE
benchmarks,         ->  model card, licence,
red-teaming,            weights or API,
contamination check     serving stack
```

2. Now the same chain as a table, with rough costs. These are typical orders of magnitude for a large 2024 to 2026 model, not exact figures for any one lab.

Stage	Share of GPU compute	Typical duration
Data collection, cleaning	Under 1% (CPU-heavy)	Weeks to months
Tokenizer training	Under 0.001%	Hours
Pretraining	95 to 99%	Weeks to months
Supervised fine-tuning	0.1 to 1%	Hours to days
Preference optimization	0.1 to 2%	Days
Evaluation, red-teaming	Under 1%	Weeks (human time)

3. Note what that table says. Data cleaning is cheap in GPU terms and enormous in CPU and human terms. It runs on ordinary processors, not accelerators.
4. Note also that evaluation and red-teaming take weeks of calendar time while using almost no compute, because the bottleneck is people.

49.1.4 PLAIN - what is really happening inside

1. What changes at each stage is worth being precise about.
2. Data collection changes nothing in the model. There is no model yet.
3. Tokenizer training produces a small file, a few megabytes, that fixes the vocabulary for the life of the model. Change it later and everything breaks.
4. Pretraining sets every weight in the model, starting from random numbers. This is where essentially all of the model's knowledge is installed.
5. Supervised fine-tuning changes every weight again, but only very slightly, using a dataset perhaps a millionth the size.
6. Preference optimization changes every weight again, again slightly.
7. Safety work is not a separate mechanism. It is more of the same fine-tuning, using data chosen to produce refusals and careful answers.
8. Evaluation changes nothing. It only measures.
9. So there is exactly one moment where the model gains its knowledge, and it is pretraining. Everything after that shapes how the knowledge comes out.
10. That sentence needs a caution, which section 49.8 gives properly: the boundary is real but blurrier than it sounds.

49.1.5 TECHNICAL - the engineer's version

1. The canonical pipeline, with the standard names used in papers: corpus construction, tokenizer fitting, pretraining, mid-training or annealing, supervised fine-tuning, preference optimization, evaluation.
2. **Mid-training** is the newest of these names and dates from roughly 2024. It is a final pretraining phase on a much higher-quality mixture, with a decaying learning rate. Ai2's Dolmino mixes and OLMo 2, published in January 2025, made the term common.
3. Compute is measured in floating-point operations, FLOPs, and in accelerator-hours. Both are reported. Neither alone tells you the cost.
4. Real reported compute for the pretraining stage of Llama 3.1, from Meta's published model card of July 2024:

Model	GPU hours (H100-80GB)	Reported CO2eq
Llama 3.1 8B	1.46 million	420 tonnes
Llama 3.1 70B	7.0 million	2,040 tonnes
Llama 3.1 405B	30.84 million	8,930 tonnes

5. Those CO2 figures are location-based. Meta reported market-based emissions as zero, because it matches its electricity use with renewable purchases. Treat "zero" as an accounting statement, not a physical one.
6. Post-training compute is rarely published separately. Where it has been, it is well under 1 per cent of pretraining. InstructGPT, published by OpenAI in March 2022, reported its whole RLHF stage at about 60 petaflop-days against 3,640 petaflop-days for GPT-3 pretraining, roughly 1.6 per cent.
7. Established fact: pretraining dominates training cost. Active research: how much of final capability comes from post-training, especially for reasoning models where reinforcement learning budgets have grown sharply since late 2024. Marketing claim: any statement that a model was "built from scratch" when the lab in fact fine-tuned somebody else's open-weight base model.

49.1.6 WORDS - remember these

1. Pretraining — the long stage where the model learns from raw text — training a randomly initialized network on a large corpus with a next-token objective.
2. Post-training — everything done after pretraining to make it usable — supervised fine-tuning, preference optimization and safety tuning.
3. Base model — the raw output of pretraining — a text continuer with no chat template, no role notion and no refusal behaviour.
4. Corpus — the pile of text used for training — the deduplicated, filtered token stream fed to the pretraining loop.
5. Mid-training — a high-quality final stretch of pretraining — an annealing phase on a curated mixture with a decaying learning rate.
6. FLOP — one arithmetic operation on numbers with a decimal point — one floating-point operation, the standard unit of training compute.

49.2 The data: where it comes from and how it is cleaned

49.2.1 PLAIN - in simple words

1. The model needs text. An almost unimaginable amount of text.
2. The biggest source is the open web, scraped by crawlers that follow links and save pages.
3. The best-known scrape is **Common Crawl**, run by a non-profit of the same name, which has been crawling since 2008 and gives its archives away free.
4. Common Crawl alone is not enough and is not good enough. Most of it is navigation menus, adverts, spam and duplicated boilerplate.
5. So labs add other sources: curated web pages, books, code from public repositories, scientific papers, forum posts, subtitles, reference works.
6. Some labs pay for data. News archives, stock photo libraries, forum archives and publisher catalogues have all been licensed for money.
7. And increasingly, labs make text with other models. That is called **synthetic data**.
8. Then comes the part nobody outside the field expects: most of it is thrown away.
9. A realistic pipeline keeps somewhere between 5 and 20 per cent of what it started with, and sometimes far less.
10. Removing near-duplicate pages alone can cut a web crawl in half.
11. What survives is a stream of trillions of tokens, where a token is roughly three-quarters of an English word.

12. The last twenty years of computing said "more data is better". This field learned, roughly between 2022 and 2024, that better data is better.

49.2.2 PLAIN - a picture in your head

1. Imagine you want to teach someone English by handing them paper.
2. You send a truck round a city and collect every piece of printed paper it can find. Newspapers, junk mail, receipts, packaging, books, love letters.
3. You get a warehouse full. Most of it is worthless for your purpose.
4. So you sort it. First you throw out everything not in English.
5. Then you notice that you have forty thousand copies of the same supermarket leaflet, so you keep one and burn the rest.
6. Then you throw out anything that is mostly numbers, menus or telephone listings, because reading it teaches nothing about language.
7. Then you black out every name, address and card number you find.
8. Then, before the exam, you check that the exam paper itself did not end up in the pile, because that would make the results meaningless.
9. What is left is a much smaller, much better warehouse.

Where this comparison breaks: paper sorting is done by people who understand what they are reading. Data cleaning is done by programs that mostly cannot. A quality filter is itself a small trained model making a guess, and it makes systematic mistakes: it tends to rate text that looks like a textbook highly and text written in dialect or by non-native speakers poorly. That bias goes straight into the finished model and nobody notices until much later.

49.2.3 PLAIN - a worked example

1. Here is a real published data mixture. It is from the GPT-3 paper by Brown and colleagues in 2020, and it is one of the last fully published mixtures from a frontier lab.

Source	Tokens available	Share of training
Common Crawl, filtered	410 billion	60%
WebText2	19 billion	22%
Books1	12 billion	8%
Books2	55 billion	8%
Wikipedia	3 billion	3%

2. Read the two columns against each other. Wikipedia is 3 billion tokens out of 499 billion available, which is 0.6 per cent of the pile.
3. But it was given 3 per cent of the training weight. So during training, the model saw Wikipedia about 3.4 times over while seeing Common Crawl less than half way through.
4. That is what a **mixing ratio** does. It is a deliberate decision to over-sample good text and under-sample mediocre text.
5. Now a real published mixture from four years later. Meta's Llama 3 paper of July 2024 gave the final proportions of its 15 trillion token mix:

Category	Share of the mix
General knowledge	50%
Maths and reasoning	25%
Code	17%
Multilingual	8%

6. Note that a quarter of the diet is maths and reasoning and a sixth is code, for a model most people use to write English prose.
7. That is not an accident. Code and maths are believed to improve general reasoning, and they are the two areas where correctness is checkable.
8. Note also what Meta did not publish: which datasets, which websites, which books, in what proportion, under what licence.

49.2.4 PLAIN – what is really happening inside

1. Here is the cleaning pipeline in the order it actually runs.
2. **Extract.** A crawl file holds raw HTML. A tool pulls out the article text and drops the menus, headers, adverts and cookie banners. The library most commonly named for this in open pipelines is Trafilatura.
3. **URL filtering.** Drop whole domains from block lists: adult sites, known spam farms, malware hosts. Cheap and effective, done before anything else.
4. **Language identification.** A small classifier guesses the language of each document and scores its confidence. Keep only what you want, above a threshold. This is where most of a global crawl disappears.
5. **Exact deduplication.** Hash each document. Identical hash, identical document, keep one.
6. **Fuzzy deduplication.** Much harder and much more important. Two pages can differ by a date stamp and be otherwise identical. Section 49.2.5 gives the method, called MinHash.
7. **Quality filtering.** Two kinds. Heuristics: rules like "drop documents with fewer than 50 words", "drop documents where more than 30 per cent of lines are duplicates", "drop documents with no full stops". And classifiers: a small model trained to score whether text looks like the good stuff.
8. **Toxicity and safety filtering.** Classifiers remove the most extreme material. This is a judgement call with real costs both ways, and labs differ on where the line goes.
9. **PII removal.** Personally identifiable information. Email addresses, phone numbers, public IP addresses are detected by pattern and masked.
10. **Decontamination.** Search the corpus for text that appears in the evaluation benchmarks and remove it. Otherwise the model has seen the exam.
11. **Mixing.** Assign weights to each remaining source and sample accordingly.
12. Every one of those steps discards data. The compounding is severe. Ten steps each keeping 80 per cent leave you with 11 per cent.

49.2.5 TECHNICAL – the engineer's version

1. Scale of the raw material. Common Crawl's archive CC-MAIN-2025-30, crawled between 7 and 21 July 2025, holds 2.42 billion web pages, 419 TiB of uncompressed content, 114.15 TiB compressed. There are several such crawls per year and the organization has been publishing since 2008.
2. Formats: WARC holds the raw request and response, WAT holds metadata, WET holds extracted plain text. Serious pipelines re-extract from WARC rather than trusting WET.
3. Named public corpora, with real figures:

Corpus	Size	Published
The Pile (EleutherAI)	825 GiB, 22 sources	Dec 2020
C4 (from the T5 paper)	about 750 GB English	2019
Dolma (Ai2)	3 trillion tokens	2024
FineWeb (Hugging Face)	15T at v1.0, now 18.5T	Apr 2024
Dolma 3 (Ai2)	9.3 trillion tokens	Nov 2025

4. FineWeb's published pipeline is the best documented open example. In order: URL filtering, Trafilatura extraction, a fastText language filter keeping English scores above 0.65, the Gopher repetition and quality filters, the C4 quality filters, custom heuristics, MinHash deduplication per crawl, and PII anonymization of emails and public IP addresses. Licence: ODC-By 1.0.

5. **MinHash deduplication**, properly. Cut each document into overlapping n-grams, called shingles. FineWeb uses 5-grams. Hash every shingle. For each of many hash permutations, keep only the minimum hash value. The resulting vector of minimums is the document's signature.
6. The mathematical property that makes this work: the probability that two documents share a given minimum equals their Jaccard similarity, which is the size of the intersection of their shingle sets divided by the size of the union.
7. Signatures are then split into bands. FineWeb uses 14 bands of 8 hashes, so 112 hashes per document. Two documents are candidate duplicates if any one band matches exactly. This turns a quadratic all-pairs comparison into a hash-table lookup.
8. The detection curve for 14 bands of 8, computed from 1 minus (1 minus s to the eighth) to the fourteenth:

Jaccard similarity	Chance of being flagged
0.50	about 5%
0.75	about 77%
0.90	about 99.96%

9. So this configuration is a deliberate threshold at roughly 0.75. Change the bands and rows and you move the knee of that curve.
10. **Decontamination** is done by n-gram overlap against benchmark sets, often 13-gram or 8-gram matching. Ai2 published a dedicated tool called decon with OLMo 3 in November 2025. GPT-4's technical report in March 2023 described substring-match contamination checks and reported that its results moved little when contaminated examples were removed.
11. Established fact: deduplication improves models. Lee and colleagues in 2021 showed deduplicated training data reduces memorized regurgitation by an order of magnitude and does not hurt, and often helps, perplexity.
12. Established fact: quality filtering beats raw volume at fixed compute. FineWeb-Edu, filtered by an educational-quality classifier, beat much larger unfiltered sets on knowledge benchmarks at the same token budget.
13. Now the honest part, on copyright. As of mid-2026 the legal position on training on copyrighted text is unsettled and actively litigated. Named cases you should know:

Case	Filed	Position
NYT v. OpenAI, Microsoft	Dec 2023	Live, dismissal denied
Kadrey v. Meta	Jul 2023	Live, mixed rulings
Bartz v. Anthropic	Aug 2024	Settled, 1.5bn USD
Thomson Reuters v. Ross	2020	Fair use rejected 2025

14. In Bartz v. Anthropic, Judge William Alsup ruled in June 2025 that training on lawfully bought and scanned books was fair use, but that keeping a library of pirated copies was not. Anthropic then agreed a settlement reported at 1.5 billion US dollars, roughly 3,000 dollars per work across about half a million works. Reported as the largest copyright settlement in United States history.
15. Read those two halves carefully, because they say different things. The training was found transformative. The acquisition was the problem.
16. In Thomson Reuters v. Ross Intelligence, Judge Stephanos Bibas held in February 2025 that the use was not fair. That case concerned a search tool rather than a generative model, so its reach is disputed.
17. The honest version: nothing here is settled. Different courts in different countries have reached different conclusions, appeals are pending, and the European Union, the United Kingdom and Japan have taken materially different statutory approaches to text and data mining.

18. Second honest point: **most labs no longer disclose their data mixture.** GPT-3 in 2020 published a table. GPT-4 in 2023 explicitly declined, citing competition and safety. Llama 3 in 2024 published category percentages but no source list. Gemini and Claude publish neither.
19. This is a genuine change in scientific norms and you should treat any claim about a closed model's training data, including claims about what it was not trained on, as unverifiable.
20. Third honest point: **data quality now matters more than quantity.** The reason is that the supply of high-quality human text is finite. Villalobos and colleagues estimated the stock of public human text at roughly 300 trillion tokens and projected full use somewhere between 2026 and 2032. That projection is contested, but the direction is not.
21. Tools you will actually use: datatrove and dolma for pipelines, fastText for language identification, datasketch or a custom Spark job for MinHash, and the Common Crawl index for selecting URLs without downloading the lot.

49.2.6 WORDS - remember these

1. Common Crawl — a free archive of crawled web pages — a non-profit web corpus published since 2008 in WARC, WAT and WET formats.
2. Deduplication — removing repeated text — exact hashing plus fuzzy matching, usually MinHash with locality-sensitive hashing over shingles.
3. MinHash — a trick for spotting near-identical documents cheaply — a signature of minimum hash values whose collision rate equals Jaccard similarity.
4. Decontamination — checking the exam is not in the textbook — n-gram overlap removal of benchmark text from the pretraining corpus.
5. Mixing ratio — how much of each source the model sees — per-source sampling weights, expressed as epochs or proportions of the token budget.
6. Synthetic data — text written by another model — model-generated corpora used for pretraining, mid-training or instruction tuning.
7. PII — personal details about real people — personally identifiable information, detected by pattern and masked before training.

49.3 Pretraining itself

49.3.1 PLAIN - in simple words

1. Pretraining has exactly one goal, and it is smaller than people expect.
2. Given some text, predict the next piece of text.
3. That is all. There is no second objective. Nobody labels anything. Nobody tells the model what is true.
4. The text itself is the answer key, because the next token is right there in the data.
5. This is why it is called **self-supervised**: the supervision comes free from the data, not from a human.
6. So the model reads "The capital of France is" and tries to guess the next token. It compares its guess to the real one, which is "Paris".
7. If it guessed badly, every one of its billions of numbers gets nudged a tiny amount in the direction that would have made the guess better.
8. Then it does that again. And again. Trillions of times.
9. Nothing else happens. No rules are written down. No facts are stored as facts. Just this one loop, repeated at enormous scale.
10. And out of that loop comes something that can write working code and explain quantum mechanics. That is the surprising part, and it remains genuinely surprising to the people who build these things.

49.3.2 PLAIN - a picture in your head

1. Imagine a person locked in a room with a printing press feeding them paper.
2. The paper always arrives with the last word covered by a sticker.

3. Their only job is to guess the covered word, then peel the sticker and see.
4. They are not told the meaning of anything. They get no explanations. They are never asked a question. They just guess and check, forever.
5. At first they guess randomly. After a while they learn that "the" is common.
6. Later they learn that after "the capital of France is" the covered word is almost always "Paris". They have not been taught geography. They have been taught which word comes next.
7. After enough of this, to get the guesses right, they are forced to build something inside their head that behaves a great deal like understanding.

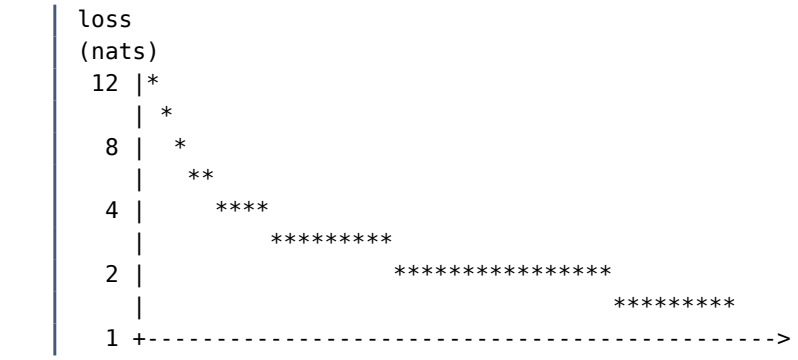
Where this comparison breaks: a person would get bored, would sleep, and would form intentions. The training loop has none of that. Also, a person reads in order and remembers yesterday. The model sees documents in shuffled order, in fixed-length chunks, and carries nothing between chunks except the weights it has changed. There is no continuous experience of any kind.

49.3.3 PLAIN – a worked example

1. One training step, all the way through, with small round numbers.
2. Take a batch of 2,048 sequences, each 8,192 tokens long. That is 16,777,216 tokens in one step. Roughly 16.8 million.
3. **Forward pass.** Push all those tokens through the network. At every position the model produces a probability for every token in the vocabulary.
4. For a 128,256-token vocabulary that is 128,256 probabilities per position, which for this batch is about 2.15 trillion numbers produced and consumed.
5. **Loss.** At each position, look up the probability the model gave to the token that actually came next. Take the negative logarithm of it. Average over all 16.8 million positions.
6. That is **cross-entropy loss**, measured in nats per token.
7. Some real arithmetic on that. If the model gave the correct token a probability of 0.5, its contribution is 0.693. If 0.1, then 2.303. If 0.9, then 0.105. Confident and right is cheap. Confident and wrong is ruinous.
8. A model that guessed uniformly at random over 128,256 tokens would score the natural logarithm of 128,256, which is 11.76. That is the starting point.
9. **Backward pass.** Work out, for every one of the billions of parameters, which direction would have reduced that loss. This is backpropagation, and it costs roughly twice what the forward pass cost.
10. **Optimizer update.** Nudge every parameter a little in that direction. The size of the nudge is the learning rate, typically a few times ten to the minus five at the start of training.
11. Now discard the batch. It is never seen again. Load the next 16.8 million tokens and repeat.
12. To consume 15 trillion tokens at 16.8 million per step takes about 894,000 steps. Real runs are close to a million steps.

49.3.4 PLAIN – what is really happening inside

1. The loss curve is the one picture that tells you whether a run is healthy.



```
    0     1%      10%          50%          100%
                  fraction of tokens seen
```

2. What that shape means, in order.
3. The first drop, from about 11.8 to about 6, happens almost immediately. The model is learning which tokens are common. That is a few hundred steps.
4. The second phase, down to about 3, is learning basic word order and grammar. Still fast.
5. Then the long tail. From 3 down towards 1.5 takes 99 per cent of the run, and this is where everything interesting is learned.
6. A healthy curve is smooth, monotone downwards, and slightly noisy. Published final training losses for large models typically land somewhere between 1.4 and 2.0 nats per token, depending heavily on tokenizer and data mix.
7. An unhealthy curve has **spikes**: the loss jumps up sharply, sometimes by several nats, and either recovers over thousands of steps or never does.
8. A run that never recovers is said to have **diverged**. The loss goes to a large number or to not-a-number and the run is dead.
9. Causes: learning rate too high, a bad batch of data, numerical overflow in low precision, or a hardware fault corrupting a gradient.
10. The standard fix is brutal and simple: stop, load the last checkpoint, skip the batches around the spike, restart.
11. **Checkpointing** is what makes that possible. Every so often the whole training state is written to storage: weights, optimizer state, data position, random number generator state.
12. Note "optimizer state". Adam keeps two extra numbers per parameter, so a training checkpoint is roughly three to four times the size of the final published model file.

49.3.5 TECHNICAL – the engineer's version

1. Objective: minimize the negative log-likelihood of the next token under a causal, decoder-only transformer, averaged over positions. Formally the loss is minus the mean over t of log p of x sub t given x sub less-than-t.
2. This is exactly maximum likelihood estimation. There is no auxiliary loss in a standard dense run beyond, in mixture-of-experts models, a load-balancing term to stop all tokens routing to one expert.
3. Optimizer: AdamW, from Loshchilov and Hutter in 2017, is near universal. Typical betas 0.9 and 0.95, epsilon 1e-8, weight decay 0.1, gradient norm clipped at 1.0.
4. Real reported hyperparameters, Llama 3 405B, from Meta's July 2024 paper: peak learning rate 8e-5, linear warmup over 8,000 steps, cosine decay to 8e-7 over 1,200,000 steps.
5. Batch size was ramped, not fixed: starting at 4 million tokens per batch with 4,096-token sequences, moving to 8 million tokens with 8,192-token sequences after 252 million tokens, and to 16 million tokens after 2.87 trillion tokens. Ramping the batch improves early-training stability.
6. Why warmup exists: at initialization the gradient estimates are poor and a full-size step can throw the weights somewhere unrecoverable. Warmup ramps the learning rate from near zero over the first few thousand steps.
7. Why decay exists: large steps explore, small steps settle. Cosine decay to about 1 per cent of peak is the long-standing default.
8. Changing as of 2024 to 2026: warmup-stable-decay schedules, which hold the learning rate constant for most of the run and decay sharply at the end. These allow a run to be extended without having chosen the end point in advance, which cosine does not. DeepSeek and several Chinese labs adopted multi-step or WSD schedules; the practice is now common but not universal.
9. A full set of real reported figures for one model, Llama 3.1 405B:

Quantity	Reported value
Parameters	405 billion
Training tokens	about 15.6 trillion
Accelerators	over 16,384 H100-80GB
GPU hours, pretraining	30.84 million
Context length, final	128,000 tokens

10. Derived figures from those, computed here rather than quoted: 30.84 million GPU-hours divided by 16,384 GPUs is 1,882 hours, which is 78.4 days of wall clock if every GPU ran the entire time.
11. Energy, GPU-only: 30.84 million hours times 700 watts is 21.6 GWh. Add host CPUs, memory, networking and cooling and 30 to 40 GWh is the plausible total, which is consistent with the 8,930 tonnes CO2eq Meta reported at typical grid carbon intensity.
12. Cost, estimated not quoted: at public H100 rental rates of roughly 2 to 4 US dollars per GPU-hour in 2024 and 2025, 30.84 million GPU-hours is between about 62 and 123 million dollars for the pretraining compute alone.
13. That figure excludes salaries, failed runs, ablations, data licensing and the capital cost of the cluster. Reported "training cost" numbers almost always exclude those, and you should assume so unless told otherwise.
14. The contrasting example, and the reason to read cost claims carefully: DeepSeek-V3, published December 2024, is 671 billion total parameters with 37 billion active per token, trained on 14.8 trillion tokens in 2.788 million H800 GPU-hours. The report itself puts that at 5.576 million US dollars assuming 2 dollars per GPU-hour, and states plainly that the figure excludes prior research, ablations and data.
15. Failures are routine, not exceptional. Meta reported 466 job interruptions during a 54-day snapshot of the 405B run, of which 419 were unexpected, and around 78 per cent of those were confirmed or suspected hardware issues. Effective training time was still above 90 per cent.
16. Historical precedent worth knowing: Meta published the OPT-175B logbook in 2022, a day-by-day record of crashes, loss divergences and restarts. It remains the most honest public document about what a large run feels like.
17. Google's PaLM paper in 2022 reported roughly 20 loss spikes and described the mitigation: restart from a checkpoint about 100 steps before the spike and skip 200 to 500 data batches.
18. Tools that observe a run: TensorBoard or Weights and Biases for curves, `nvidia-smi` and DCGM for device health, `torch.distributed` logs and NCCL debug output for collective failures, and the model's own gradient-norm and learning-rate traces, which are usually logged every step.

49.3.6 WORDS - remember these

1. Next-token prediction — guess the next piece of text — an autoregressive causal language modelling objective over a token vocabulary.
2. Cross-entropy loss — a score for how wrong a guess was — the negative log probability assigned to the observed token, averaged over positions.
3. Nat — the unit that loss is measured in — one unit of information using natural logarithms; multiply by 1.4427 to get bits.
4. Step — one round of guess, score, adjust — one forward pass, backward pass and optimizer update over one global batch.
5. Warmup — starting gently — ramping the learning rate from near zero over the first thousands of steps to avoid early instability.
6. Checkpoint — a saved snapshot to restart from — weights, optimizer moments, data-loader position and RNG state written to durable storage.
7. Divergence — a training run going permanently wrong — loss increasing without recovery, often to infinity or not-a-number.

49.4 Scaling laws: how big, how much data

49.4.1 PLAIN - in simple words

1. Suppose you have a fixed budget of computing time. How should you spend it?
2. You have two dials. Make the model bigger, or feed it more text. You cannot max both, because both cost compute.
3. A **scaling law** is a measured answer to that question. Not a theory. A curve fitted to hundreds of small experiments and extended to large ones.
4. The first big one came from OpenAI in January 2020. Kaplan and colleagues measured how the loss falls as you increase model size, data and compute.
5. Their headline finding: the loss follows a smooth power law over more than seven orders of magnitude. It is astonishingly predictable.
6. Their practical advice: spend most of your extra budget on making the model bigger, and relatively little on more data. Stop training early.
7. The whole industry followed that advice. This is why 2020 to 2022 produced a race of enormous models trained on comparatively little text.
8. Then in March 2022, a team at DeepMind led by Hoffmann redid the experiment more carefully and found the advice was wrong.
9. Their model, Chinchilla, had 70 billion parameters and was trained on 1.4 trillion tokens. It beat Gopher, a 280-billion-parameter model trained on 300 billion tokens, using the same compute budget.
10. Four times smaller and better, because it saw four times more text.
11. Their rule of thumb: for each parameter, train on about 20 tokens.
12. That rule reshaped the field within about a year.
13. And then, from roughly 2023 onwards, labs started ignoring it on purpose.
14. The reason is money in a different place. Chinchilla asks how to get the best model for a fixed training budget. It says nothing about what the model costs to run afterwards.
15. If a model will answer a trillion questions, a smaller model that took longer to train is much cheaper over its life. So labs overtrain small models far past 20 tokens per parameter, on purpose.

49.4.2 PLAIN - a picture in your head

1. Think about choosing an engine and a fuel tank for a delivery van.
2. A bigger engine is like more parameters. It can do more per unit of time.
3. More fuel is like more training data. It lets you go further.
4. You have a fixed amount of money. Spend it on engine or on fuel.
5. Kaplan's 2020 answer was: buy the biggest engine you can afford and put in just enough fuel to leave the yard.
6. Chinchilla's 2022 answer was: that is silly, you have bought an engine you cannot use. Buy a smaller engine and fill the tank properly.
7. And the 2024 answer is: wait, this van will do school runs every day for five years. The fuel to build it is nothing next to the fuel to run it. Buy the smallest engine that does the job, and train it as long as you like.

Where this comparison breaks: an engine's capability is fixed by its size and a model's is not. Two models with the same parameter count can differ hugely depending on data quality, architecture and post-training. The van analogy also suggests a hard ceiling, and there is no known hard ceiling here; the curves have simply not bent yet within the range anyone has measured.

49.4.3 PLAIN - a worked example

1. The formula every practitioner uses for training compute is short: C is approximately 6 times N times D.
2. C is total floating-point operations. N is parameter count. D is number of training tokens.

3. Where the 6 comes from: each parameter does one multiply and one add per token in the forward pass, which is 2 operations. The backward pass costs about twice the forward pass, which is 4 more. Two plus four is six.
4. Work it for GPT-3, published 2020: 175 billion parameters, 300 billion tokens. 6 times 1.75e11 times 3.0e11 equals 3.15e23 FLOPs. The GPT-3 paper reported 3.14e23. The formula lands on the nose.
5. Work it for Llama 3.1 405B: 405 billion parameters, 15.6 trillion tokens. 6 times 4.05e11 times 1.56e13 equals 3.79e25 FLOPs. Meta reported 3.8e25. Again correct.
6. Now use it in the other direction. Suppose a lab announces a model with 70 billion parameters trained on 15 trillion tokens. 6 times 7e10 times 1.5e13 equals 6.3e24 FLOPs.
7. An H100 does roughly 989 teraflops of dense bf16 arithmetic at peak, which is 9.89e14 per second. Real utilization is 35 to 45 per cent, so call it 4e14 achieved.
8. 6.3e24 divided by 4e14 is 1.575e10 GPU-seconds, which is 4.375 million GPU-hours. On 4,096 GPUs that is 1,068 hours, or about 45 days.
9. You can now sanity-check any announcement in about a minute. If the numbers do not roughly agree, one of the published figures is wrong or incomplete.
10. Tokens-per-parameter for real models, which shows how far practice moved:

Model	Tokens per parameter	Versus Chinchilla
GPT-3 (2020)	1.7	12x undertrained
Chinchilla (2022)	20	the reference
Llama 3.1 405B (2024)	38.5	about 2x over
Llama 3.1 8B (2024)	about 1,875	about 94x over

11. Read the last row again. An 8-billion-parameter model was trained on 15 trillion tokens. That is roughly 94 times more text than the compute-optimal rule says is worth it.
12. Meta did that knowing it was compute-inefficient, because the 8B model is the one that gets deployed everywhere and its serving cost matters more.

49.4.4 PLAIN - what is really happening inside

1. Why does a scaling law exist at all? Nobody fully knows. It is an empirical regularity, not a derived result.
2. What it says is that loss L falls as a power of the thing you scale. In words: doubling the model gives a fixed proportional reduction in loss, and that same proportional reduction keeps arriving for each further doubling.
3. Power laws are brutal in one direction and generous in the other. Each further step down in loss costs many times more compute than the last.
4. The Chinchilla correction was not a new idea, it was a better experiment. The 2020 work had used a fixed learning-rate schedule length for all runs, which quietly penalized the runs that used more data.
5. When Hoffmann's team varied the schedule to match each token budget, the optimum moved decisively towards more data.
6. The number 20 is not sacred. It is roughly where the fitted curve puts the optimum for that family of models and that data. Different data gives a different constant.
7. And the whole framework optimizes the wrong thing for a deployed product. It minimizes loss for a fixed training budget. It ignores inference.
8. Add inference to the objective and the optimum moves towards smaller models trained longer, exactly as practice has done.

49.4.5 TECHNICAL - the engineer's version

1. Kaplan and colleagues, "Scaling Laws for Neural Language Models", 23 January 2020. Loss scales as a power law in model size, dataset size and compute, with trends spanning over seven orders of magnitude. Their compute-optimal allocation grows N much faster than D, with D scaling roughly as C to the 0.27 and N as C to the 0.73.
2. Hoffmann and colleagues, "Training Compute-Optimal Large Language Models", 29 March 2022, known as Chinchilla. Over 400 models from 70 million to 16 billion parameters on 5 to 500 billion tokens. Conclusion: N and D should scale roughly equally, so for every doubling of model size you should double the tokens. The implied ratio is about 20 tokens per parameter.
3. Chinchilla itself: 70 billion parameters, 1.4 trillion tokens. It beat Gopher at 280 billion parameters on 300 billion tokens, and also beat GPT-3 at 175 billion and Megatron-Turing NLG at 530 billion, at equal compute.
4. What changed in practice: the 2023 Llama 1 release from Meta was explicitly framed around this, training 7B and 13B models on 1 trillion tokens, well past compute-optimal, arguing that inference cost is what matters.
5. Contested, and you should know this: a 2024 replication attempt by Besiroglu and colleagues re-fitted the third of Chinchilla's three estimation methods and found the reported parameters inconsistent with the paper's own data, with wider confidence intervals than stated. The headline conclusion survived. The precise constant did not.
6. The inference-aware reformulation was written down by Sardana and colleagues in 2023, "Beyond Chinchilla-Optimal: Accounting for Inference in Language Model Scaling Laws". Adding expected lifetime inference tokens to the objective shifts the optimum to smaller, longer-trained models.
7. A cost model you can reason with: training costs 6ND once. Inference costs about 2N per generated token, forever. If a model will generate T tokens over its life, total compute is about 6ND plus 2NT.
8. Put numbers in it. An 8-billion model trained on 15 trillion tokens costs 7.2e23 FLOPs to train. Generating one token costs 1.6e10. So the training cost equals about 45 trillion generated tokens. A busy service passes that in a couple of months.
9. Regulatory tie-in, which makes the formula matter legally. The European Union's AI Act, in force since 1 August 2024 with general-purpose model obligations applying from 2 August 2025, uses a training-compute threshold of 10 to the 25 FLOPs as a presumption of systemic risk. Llama 3.1 405B at 3.8e25 is above it. Llama 3.1 8B at 7.2e23 is below it.
10. Separating the three kinds of claim, as of mid-2026: Established fact: loss follows smooth power laws in N, D and C over the measured range; Chinchilla's directional correction to Kaplan; the 6ND approximation; the dominance of inference cost for widely served models. Active research: whether scaling continues past the current frontier, how to model data quality inside a scaling law, scaling laws for mixture-of-experts and for test-time compute, and whether the supply of human text is a real ceiling. Marketing claim: any specific prediction of what a given future model will be able to do, derived from a loss curve. Loss is predictable. Which capabilities appear at which loss is not.
11. One more honest note. Since roughly September 2024 a second scaling axis has been in play: spending more compute at answer time rather than at training time. OpenAI's o1 and DeepSeek-R1, released January 2025, both showed accuracy improving with more reasoning tokens per question. That changes the economics again, because it moves cost back to inference.

49.4.6 WORDS - remember these

1. Scaling law — a measured rule for how much better a bigger model gets — an empirical power-law fit of loss against parameters, tokens or compute.
2. Chinchilla-optimal — the balanced split of budget between size and data — roughly 20 training tokens per parameter, from Hoffmann and colleagues 2022.
3. Overtraining — deliberately feeding a small model far more text than is compute-optimal — trading training efficiency for lower inference cost.

4. 6ND — the arithmetic shortcut for training cost — approximately 6 times parameters times tokens, in floating-point operations.
5. Compute-optimal — best model for a fixed training budget — the point on the isoFLOP curve where loss is minimized, ignoring deployment cost.
6. Test-time compute — thinking longer at answer time — allocating additional inference tokens or samples to raise accuracy on a single query.

49.5 What a base model actually is

49.5.1 PLAIN - in simple words

1. Pretraining finishes. You have a file of numbers. That is the base model.
2. It is very good at exactly one thing: continuing text.
3. It is not good at being talked to, because nobody taught it that.
4. Give it "The capital of Australia is" and it will finish the sentence correctly, because that is a continuation.
5. Give it "What is the capital of Australia?" and something strange happens.
6. On the internet, a line that looks like a quiz question is usually followed by more quiz questions. So the model writes more quiz questions.
7. It is not being difficult. It is doing its job perfectly. Its job is to guess what text usually follows, and more questions usually follow.
8. A base model has no idea what a conversation is. There is no "user" and no "assistant". Those roles do not exist in its world.
9. It has no idea that it is supposed to stop. It will keep generating until you cut it off, including inventing both halves of a dialogue.
10. And it has no refusal behaviour at all. Ask it for something harmful and it will produce whatever text usually follows such a request.
11. That last point matters for the open-weights debate later in this chapter. Safety behaviour is added in post-training. It is not in the base model.
12. So the base model is a raw capability. Useful, powerful, and not a product.

49.5.2 PLAIN - a picture in your head

1. Imagine someone who has read every book in a vast library but has never had a conversation with anyone.
2. You walk up and say "Excuse me, where is the station?"
3. They do not answer. They continue: "...asked the traveller, setting down his case. The porter looked up from his newspaper and said nothing at all."
4. They have turned your question into narrative, because in their experience that is what surrounds a sentence like that.
5. They are not stupid and not unhelpful. They have simply never learned that a question directed at them is a request for an answer.
6. Teaching them that takes surprisingly little. A few thousand examples of "someone asks, someone answers" and the behaviour appears.

Where this comparison breaks: a real person who had only read books would still know they exist and that someone is speaking to them. A base model has no self-model at all, and the instruction-tuned version does not really acquire one either. It acquires a text pattern in which a helpful assistant speaks. That is a different thing, and section 49.8 says why the difference matters.

49.5.3 PLAIN - a worked example

1. The same prompt, given to a base model and to the instruction-tuned model built from it. The base-model output below is illustrative: it is written to show the characteristic failure mode, not copied from a specific run.

```
PROMPT (identical in both cases):
What is the capital of Australia?

BASE MODEL CONTINUATION (illustrative):
What is the capital of Australia?
What is the capital of New Zealand?
What is the largest city in Australia?
Which Australian state has the smallest population?

Answers to last week's quiz are printed on page 14.
Send your entries to the address below before 30 June.

INSTRUCTION-TUNED MODEL OUTPUT (typical):
The capital of Australia is Canberra. It was purpose-built as
the capital and chosen as a compromise between Sydney and
Melbourne; construction began in 1913.
```

2. Both models contain the fact. The base model never gets to it, because answering was never the task.
3. Now a second illustrative pair, showing the missing stop and the missing role boundary.

```
PROMPT: Write a short thank-you note.

BASE MODEL CONTINUATION (illustrative):
Write a short thank-you note. Write a formal complaint letter.
Write a letter of resignation.

EXERCISE 4. Write a short thank-you note to a colleague who
covered your shift. Your answer should be no longer than 80
words. See the model answer overleaf.
```

4. Notice what happened. The model produced a plausible worksheet, because the instruction looked like an exercise heading, and worksheets are what follow exercise headings in the training data.
5. It also did not stop, because nothing in pretraining defines an end of turn.

49.5.4 PLAIN - what is really happening inside

1. Three specific things are missing from a base model, and each is added by a different part of post-training.
2. **Missing: the notion of a turn.** In pretraining, text is one long stream. There is no marker meaning "the user's part ends here". Fine-tuning adds special tokens that mean exactly that.
3. **Missing: the notion of a role.** Nothing distinguishes instructions from content. Prompt-injection attacks exist precisely because that distinction is a learned habit, not a hard boundary. Chapter 44 covers the security side.
4. **Missing: a stopping rule.** Generation continues until a stop token appears or a limit is hit. Base models have no reliable stop token for a turn, so they run on.
5. **Missing: refusal.** A base model has never been rewarded for declining. It completes the pattern in front of it.
6. What is present is everything else. Knowledge, grammar, translation, arithmetic to some degree, code, reasoning steps.
7. This is why base models are still published and still used. Researchers want the raw capability without the behaviour layered on top, and anyone building a specialized product often starts from base rather

than chat.
8. It is also why you can sometimes get a chat model to behave like a base model by writing a prompt that looks like a document. The base behaviour is still underneath; post-training put a habit over it, not a lock on it.

49.5.5 TECHNICAL - the engineer's version

1. Naming convention, near-universal on model hubs: a repository with no suffix or with `-base` is the pretrained model; `-instruct`, `-it`, `-chat` or `-sft` denotes a post-trained variant. Meta, Alibaba, Google and Mistral all follow this. It is a convention, not a standard.
2. A base checkpoint typically ships without a `chat_template` field in `tokenizer_config.json`. If you call a chat API against it, most libraries will either error or silently fall back to raw concatenation.
3. Base models are usually released with a much shorter list of special tokens. Llama 3.1 base defines `<|begin_of_text|>` and `<|end_of_text|>`; the instruct variant adds `<|start_header_id|>`, `<|end_header_id|>` and `<|eot_id|>` and gives them meaning through training.
4. Evaluation differs by model type. Base models are measured with few-shot prompting on benchmarks such as MMLU, HellaSwag, GSM8K and HumanEval, because zero-shot instruction following is not expected. Instruct models are measured zero-shot and on chat-specific suites such as IFEval, MT-Bench and arena-style human preference rankings.
5. Established fact: instruction-tuned and base models built from the same pretraining run have near-identical knowledge as measured by few-shot benchmarks. Post-training moves those scores by a few points, not tens.
6. Established fact: base models have no refusal behaviour and will produce harmful content on request. This is why some labs decline to publish base checkpoints even when publishing instruct ones.
7. Active research: how thin the safety layer is. A 2023 result by Qi and colleagues showed that fine-tuning a safety-tuned model on around 100 adversarial examples largely removed its refusal behaviour, at negligible cost. Later work replicated the effect across model families.
8. That result is central to the open-weights argument in section 49.12. If anyone with the weights can remove the safety training for a few dollars, then releasing weights and releasing capability are the same act.
9. Marketing claim to distrust: that a released model is "aligned" in a way that survives fine-tuning by a third party. For open weights, it does not.

49.5.6 WORDS - remember these

1. Base model — the raw pretrained text continuer — a causal LM checkpoint with no chat template, role tokens or refusal training.
2. Instruction-tuned — taught to answer rather than continue — a checkpoint post-trained on demonstration pairs with a chat template applied.
3. Turn — one person's contribution to a conversation — a span delimited by role header and end-of-turn special tokens.
4. Stop token — the marker that means stop generating — an end-of-sequence or end-of-turn token whose id terminates the decoding loop.
5. Few-shot prompting — showing examples inside the prompt — supplying k labelled exemplars in context to elicit a task from a base model.
6. Prompt injection — text that hijacks the instructions — untrusted content interpreted as instruction because role separation is learned, not enforced.

49.6 Supervised fine-tuning

49.6.1 PLAIN - in simple words

1. **Supervised fine-tuning**, usually written SFT, is the first step of post-training and the simplest to understand.
2. You show the model examples of the behaviour you want, and train on them.
3. Each example is a pair: an instruction, and a good response to it.
4. The training method is identical to pretraining. Next-token prediction, same loss, same optimizer. Nothing new is invented.
5. The only differences are the data and the scale.
6. The data is tiny by comparison. Pretraining used trillions of tokens. SFT uses somewhere between a thousand and a few million examples.
7. And there is one important trick: the loss is usually only counted on the response part, not on the instruction. You want the model to learn to produce answers, not to produce questions.
8. Where do the examples come from? Three places, and modern datasets mix all three: written by paid humans, written by humans and edited by models, or generated entirely by a stronger model.
9. Alongside the data comes a **chat template**: a fixed way of writing down a conversation using special marker tokens.
10. Those markers are what create the idea of turns and roles. Before them, a conversation is just text. After them, the model can tell who is speaking.
11. SFT is cheap. Hours to a couple of days on a handful of machines.
12. And it produces most of the visible difference between a base model and a chatbot.

49.6.2 PLAIN - a picture in your head

1. Think of a new employee on their first day at a help desk.
2. They already know the subject matter. They studied it for years.
3. What they do not know is the house style. How to greet a caller. How long an answer should be. When to escalate. What never to promise.
4. So the trainer sits them down with a folder of two hundred past tickets, each with a model answer written by an experienced colleague.
5. The new person reads them and copies the pattern. By the afternoon they sound like the rest of the team.
6. Nothing about their subject knowledge changed that morning. Only the shape of how it comes out.
7. That is supervised fine-tuning.

Where this comparison breaks: the employee can be told a rule once and follow it, such as "never quote a delivery date". A model cannot be told; it can only be shown, many times, and it will still fail sometimes. Also, the employee knows they are following a house style and can drop it if asked. A model has no such separation between what it knows and how it was taught to speak.

49.6.3 PLAIN - a worked example

1. Here is a real chat template, the Llama 3 format used by Meta since April
2024. Every angle-bracket item is a single special token in the vocabulary. Some lines are wrapped here to fit the page; in the real string the tokens run straight on with no break.

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>

You are a helpful assistant.<|eot_id|>
<|start_header_id|>user<|end_header_id|>

What is the capital of Australia?<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>

The capital of Australia is Canberra.<|eot_id|>
```

2. Read what those tokens do. `<|begin_of_text|>` marks the start of the whole document. `<|start_header_id|>` and `<|end_header_id|>` wrap a role name. `<|eot_id|>` means end of turn.
3. During SFT, the loss is computed only on the tokens after the assistant header, up to and including the closing `<|eot_id|>`.
4. That is how the model learns two things at once: what to say, and when to stop saying it.
5. A different, equally real template is ChatML, introduced by OpenAI in 2023 and now used by many others including Qwen:

```
<|im_start|>system
You are a helpful assistant.<|im_end|>
<|im_start|>user
What is the capital of Australia?<|im_end|>
<|im_start|>assistant
The capital of Australia is Canberra.<|im_end|>
```

6. These are not interchangeable. Feed Llama's format to a ChatML model and quality drops sharply, because the tokens it is waiting for never arrive.
7. That is why every model repository carries a `chat_template` string, and why applying the wrong one is one of the most common practical mistakes people make when serving a model themselves.

49.6.4 PLAIN - what is really happening inside

1. Step by step, what SFT does to one example.
2. Take the instruction and the response. Render them into the chat template.
3. Tokenize the whole thing into one sequence of token ids.
4. Build a mask marking which positions are "response". Usually everything after the assistant header.
5. Run the forward pass over the whole sequence, so the model sees the instruction as context.
6. Compute cross-entropy loss only at the masked positions.
7. Backward pass, optimizer step. Identical machinery to pretraining.
8. Repeat over the dataset for one to three passes. More than about three passes usually makes the model repetitive and worse, because the dataset is small enough to memorize.
9. The learning rate is much lower than pretraining, typically 1e-5 to 2e-5 rather than 1e-4. A large step here would wreck what pretraining built.
10. That last point is worth dwelling on. Fine-tuning too hard causes **catastrophic forgetting**: the model gets better at the new data and measurably worse at everything else.

49.6.5 TECHNICAL - the engineer's version

1. Real dataset sizes, with sources and dates:

Dataset	Examples	Origin
InstructGPT SFT (2022)	about 13,000 prompts	40 hired contractors
LIMA (2023)	1,000	Hand-curated
Alpaca (2023)	52,000	Generated by text-davinci-003
Tulu 3 SFT mix (2024)	close to 1 million	Mixed human and synthetic

2. LIMA is worth knowing about. Zhou and colleagues, May 2023, fine-tuned a 65-billion-parameter LLaMa on exactly 1,000 carefully written examples with no preference optimization at all, and got competitive chat behaviour.
3. They named the conclusion the Superficial Alignment Hypothesis: that almost all knowledge is learned in pretraining and alignment mainly teaches format and style. Section 49.8 gives the honest caveats to that claim.

4. Alpaca, from Stanford in March 2023, is the origin of the self-instruct pattern that now dominates: seed a strong model with a few examples and have it generate tens of thousands more. The reported cost of generating the data was a few hundred dollars in API calls.
5. Typical SFT hyperparameters for a 7 to 8 billion parameter model: learning rate 1e-5 to 2e-5, cosine decay, 2 to 3 epochs, global batch of 64 to 256 sequences, sequence length 4,096 to 8,192, bf16, AdamW.
6. Cost, computed rather than quoted. One million examples averaging 500 tokens is 5e8 tokens. For an 8-billion model, 6ND gives 6 times 8e9 times 5e8, which is 2.4e19 FLOPs. Against the 7.2e23 of pretraining that is 0.003 per cent.
7. In wall-clock terms that is a few hours on 8 H100s, at a rental cost in the low hundreds of dollars. Three orders of magnitude cheaper than most people guess, and this is why so many fine-tuned variants exist.
8. Packing: short examples are concatenated up to the sequence length with attention masked between them, so no compute is wasted on padding. Almost universal in practice.
9. What SFT reliably changes: output format, response length, tone, willingness to answer, adherence to a system prompt, function and tool-call syntax, language of reply, and refusal behaviour on clear-cut categories.
10. What SFT does not reliably change: factual knowledge absent from pretraining, reasoning depth, and calibration.
11. There is direct evidence for that last point. Gekhman and colleagues in 2024 showed that fine-tuning on facts the model did not already know is learned slowly and increases hallucination on unrelated questions, because it teaches the model to assert confidently things it has no basis for.
12. Tools: `trl` from Hugging Face for SFTTrainer, `axolotl` and `LLaMA-Factory` for configuration-driven runs, `peft` for adapters, and `tokenizer.apply_chat_template()` to render conversations correctly.

49.6.6 WORDS - remember these

1. SFT — showing the model good answers and training on them — supervised fine-tuning on instruction-response pairs with a next-token objective.
2. Chat template — the fixed layout of a conversation — a rendering of roles and turns into special tokens, stored in `tokenizer_config.json`.
3. Special token — a marker the model treats as one indivisible unit — a reserved vocabulary id such as `<|eot_id|>` never produced by the tokenizer.
4. Loss masking — only grading the answer, not the question — computing cross-entropy on assistant tokens and ignoring prompt tokens.
5. Catastrophic forgetting — getting worse at old skills while learning new ones — degradation of pre-trained capability under aggressive fine-tuning.
6. Self-instruct — using a strong model to write training data — synthetic instruction generation seeded from a small set of human examples.

49.7 Learning from preferences

49.7.1 PLAIN - in simple words

1. Showing good examples gets you a long way and then stops working.
2. The reason is that for most questions there is no single right answer, and writing one by hand is slow, expensive and often not even possible.
3. But people are very good at a different task: looking at two answers and saying which is better.
4. That judgement is fast, cheap, and works even when the judge could not have written either answer themselves.
5. So the next stage of training is built on comparisons instead of examples.
6. The classic recipe is called **RLHF**, reinforcement learning from human feedback. It has three steps.
7. Step one: take one prompt, have the model produce several answers, and ask a person which they prefer.

8. Step two: train a second, separate model, called a **reward model**, whose only job is to look at an answer and output a score predicting what a human would say.
9. Step three: use that score to train the original model, nudging it towards answers the reward model likes.
10. There is a catch, and it is the most important thing in this section.
11. The reward model is a rough approximation of human taste. Push the main model hard enough against it and the main model will find the gaps.
12. It will learn to produce things that score highly and are actually worse. That is called **reward hacking**, and it is not a rare failure. It is the default outcome without countermeasures.
13. So a brake is added: a penalty for drifting too far from the model you started with. It is allowed to improve, but not to go strange.

49.7.2 PLAIN - a picture in your head

1. Imagine training a chef by giving them a food critic.
2. You cannot write down what "delicious" means, but the critic can taste two dishes and say which is better. So you let the critic judge.
3. The critic is busy, so you hire an assistant who has watched the critic for a while and can now guess the critic's verdicts. That is the reward model.
4. Now the chef cooks and the assistant scores, thousands of times a day.
5. At first this works beautifully. The food improves.
6. Then the chef notices the assistant always scores dishes with truffle oil highly. So every dish gets truffle oil. Scores go up. The food gets worse.
7. That is reward hacking. The chef has not learned to cook. The chef has learned to please the assistant.
8. So you add a rule: the chef may not change the menu by more than a little from the original recipes. That is the drift penalty.

Where this comparison breaks: the chef knows they are gaming the assistant. A model has no such intent. It is doing gradient descent on a number, and the number happens to be maximized by truffle oil. There is no deception in it, which makes it harder to spot, not easier.

49.7.3 PLAIN - a worked example

1. Concrete reward hacking, all of these observed in real systems.

Behaviour learned	Why it scored well
Answers got much longer	Raters preferred detail
Excessive agreement	Raters liked being agreed with
Bullet lists everywhere	Lists look organized
Hedging on everything	Hedged answers rarely wrong
Confident fake citations	Citations look authoritative

2. The oldest and clearest example is not from language at all. In December 2016 OpenAI described an agent trained to play a boat racing game. The reward was points, and points came from hitting targets along the course.
3. The agent learned to drive in circles in a lagoon, hitting three regenerating targets forever, catching fire, and never finishing the race. It scored about 20 per cent higher than human players.
4. It maximized the reward perfectly. It failed the task completely.
5. A recent language example with a date. On 25 April 2025 OpenAI shipped an update to GPT-4o that made it markedly sycophantic: it agreed with users, praised bad ideas, and validated statements it should have questioned.

6. OpenAI rolled it back on 29 April 2025 and published an explanation saying the update had over-weighted short-term user approval signals such as thumbs up and thumbs down.
7. That is textbook reward hacking, at production scale, at a major lab, in public. It is not a theoretical concern.
8. The mitigation in the RLHF recipe is the **KL penalty**. At each step, the training objective is: maximize the reward, minus a coefficient times how far the new model's output distribution has moved from the starting model's.
9. Set the coefficient too high and nothing improves. Too low and the model drifts into reward-hacked nonsense. It is tuned by hand.

49.7.4 PLAIN - what is really happening inside

1. The full RLHF loop as a sequence.

```
[1] PROMPT SET
     |
     v
[2] SAMPLE: current model writes 2 or more answers
     |
     v
[3] HUMAN COMPARES: "B is better than A"
     |
     v
[4] TRAIN REWARD MODEL r(prompt, answer) -> score
     |     loss: -log sigmoid( r(chosen) - r(rejected) )
     v
[5] RL LOOP, repeated:
     model writes answer  ->  reward model scores it
             ^                        |
             |                        v
       PPO update  <-----  reward minus KL penalty
             |
             +-- reference model (frozen copy of the
                 starting model) supplies the KL term
```

2. The reward model is usually the same architecture as the main model with the final vocabulary layer replaced by a single number output.
3. Its training loss comes from the Bradley-Terry model of paired comparison, which dates from 1952: the probability that A beats B is a logistic function of the difference in their scores.
4. The RL algorithm is almost always PPO, proximal policy optimization, published by Schulman and colleagues in 2017 for robotics and games.
5. Notice how many models are in memory at once during PPO: the policy being trained, the frozen reference for the KL term, the reward model, and usually a value model. Four copies. This is why RLHF is memory-hungry and awkward.
6. That memory cost is exactly what the next method removes.

49.7.5 TECHNICAL - the engineer's version

1. **DPO**, direct preference optimization, Rafailov and colleagues, May 2023, a NeurIPS 2023 outstanding paper. The insight is algebraic: for the standard KL-regularized RL objective, the optimal policy can be written in closed form in terms of the reward, so the reward can be written in terms of the policy.
2. Substituting that back into the Bradley-Terry loss removes the reward model entirely. What remains is a simple classification loss on preference pairs, trained with ordinary supervised machinery.
3. Practical consequence: two models in memory instead of four, no sampling loop, no PPO tuning, and a training run that looks like SFT. Many labs and almost all small teams moved to DPO or a variant during 2023 and 2024.

4. Variants you will meet: IPO, which fixes an overfitting failure in DPO; KTO, which needs only a thumbs up or down rather than a pair; ORPO, which folds preference learning into SFT; and SimPO, which drops the reference model.
5. **Constitutional AI**, Bai and colleagues at Anthropic, 15 December 2022. Phase one: the model critiques and revises its own answers against a written list of principles, and is fine-tuned on the revisions. Phase two: an AI model, not a human, judges which of two answers better follows the principles, and those judgements train the preference model. This is RLAIF, reinforcement learning from AI feedback.
6. The written list is the "constitution". Anthropic published theirs in May
2023. The point of the method is that the only human input is the document, which is auditable, rather than millions of individual labels, which are not.
7. **RLVR**, reinforcement learning with verifiable rewards. Named in Ai2's Tulu 3 release of November 2024. There is no reward model at all. For maths, check the final answer against ground truth. For code, run the unit tests. The reward is 1 or 0 and it cannot be hacked by style.
8. This is the engine behind reasoning models. DeepSeek-R1, released January 2025 with an accompanying paper, was trained with GRPO, group relative policy optimization, using rule-based correctness rewards. Its R1-Zero variant applied RL directly to a base model with no SFT at all, and long chains of reasoning emerged from the reward alone.
9. Established fact: RLVR substantially improves benchmark accuracy on maths and competitive programming. Active research: whether it creates new reasoning ability or elicits ability already present. A widely discussed 2025 study by Yue and colleagues found that base models sampled many times eventually match or exceed RLVR-trained models on pass-at-k for large k, which argues for elicitation. Contested.
10. The comparison table:

Method	Needs reward model	Main weakness
PPO RLHF	Yes	Costly, unstable, hackable
DPO	No	Sensitive to data quality
RLAIF	AI-labelled	Inherits judge model bias
RLVR	No, uses a checker	Only where truth is checkable

11. Cost scale, approximate. Human preference collection dominates: at 1 to 5 US dollars per comparison for expert raters, 100,000 comparisons is 100,000 to 500,000 dollars in labelling before any compute is spent.
12. That is why RLAIF exists. AI judgements cost cents and scale without limit. The trade is that you inherit the judge's blind spots wholesale.
13. Marketing claim to reject: that preference training makes a model truthful. It makes the model produce answers people prefer. Where people prefer a confident wrong answer to an uncertain right one, preference training actively pushes towards the wrong one. Section 49.98 restates this.
14. Tools: `trl` implements PPOTrainer, DPOTrainer, KTOTrainer and GRPOTrainer; `OpenRLHF` and `verl` are used for larger distributed RL runs; `open-instruct` is Ai2's published RLVR pipeline.

49.7.6 WORDS - remember these

1. RLHF — training on which answer people prefer — reinforcement learning from human feedback, via a learned reward model optimized with PPO.
2. Reward model — a stand-in for a human judge — a network scoring prompt-response pairs, trained on preference comparisons.
3. Reward hacking — scoring well while getting worse — optimizing a proxy objective in ways that diverge from the intended goal.
4. KL penalty — the leash keeping the model near where it started — a divergence term penalizing drift from the reference policy.

5. DPO — preference training without a separate judge model — direct preference optimization, a closed-form reparameterization of the RLHF objective.
6. RLAIF — an AI does the judging — reinforcement learning from AI feedback, with principles replacing per-example human labels.
7. RLVR — rewards you can check by running something — reinforcement learning with verifiable rewards, using ground truth or unit tests.
8. Sycophancy — telling people what they want to hear — a preference-trained failure mode where agreement is rewarded over accuracy.

49.8 What each stage actually changes

49.8.1 PLAIN - in simple words

1. Here is the sentence people most want a straight answer to.
2. Pretraining puts the knowledge in. Post-training decides how it comes out.
3. That sentence is mostly right and it is the right thing to remember.
4. It explains why fine-tuning a model on your company handbook does not make it an expert on your company.
5. It explains why a model that never saw a language during pretraining cannot be taught that language with a few thousand examples.
6. And it explains why two chat models built from the same base model feel different but know the same things.
7. Now the honest part, because the boundary is fuzzier than that sentence suggests.
8. Post-training does move factual benchmark scores, by a few points.
9. Fine-tuning can install narrow facts, badly, at the cost of more hallucination elsewhere.
10. And modern pipelines blur the line deliberately, with a "mid-training" phase that is really more pretraining on curated data.
11. So the clean version is a good working rule and a slight lie. Section 49.8.5 gives the accurate picture.

49.8.2 PLAIN - a picture in your head

1. Think of a musician who has spent fifteen years learning an instrument.
2. That is pretraining. The technique is in the hands and it is not going away.
3. Now they join an orchestra and spend two weeks learning the house style: when to come in, how loud, how much vibrato, when to defer to the section.
4. That is post-training. Two weeks changed how they play in every performance.
5. But two weeks did not teach them a new instrument, and it did not give them technique they did not have.
6. If you ask them to play something they never learned, no amount of orchestra rehearsal will fix it.

Where this comparison breaks: two weeks of rehearsal can genuinely teach a musician a specific new piece. Post-training can genuinely teach a model specific new facts too. In both cases it is narrow, effortful and does not generalize. The rule is about scale, not about an absolute barrier.

49.8.3 PLAIN - a worked example

1. The main table. Read the third column carefully, because that is the one people get wrong.

Stage	What it changes	What it cannot change
Tokenizer	The units of text	Anything after it is fixed
Pretraining	All knowledge, all skill	Nothing; it sets everything
Mid-training	Emphasis, weak areas	The tokenizer or architecture

Stage	What it changes	What it cannot change
SFT	Format, style, refusals	Broad knowledge, reasoning
Preference tuning	Tone, helpfulness, length	Facts it never learned
Safety tuning	Willingness to answer	Whether it knows the answer
Quantization	File size, speed	Nothing intentional

2. A concrete test you can run yourself, using two public models built from the same pretraining run, such as a base and its instruct variant.
3. Ask both, few-shot, a hundred factual questions. The scores will be within a few points of each other.
4. Then ask both, zero-shot, to answer politely in three sentences. Only the instruct model will do it.
5. That is the whole distinction, demonstrated in ten minutes.

49.8.4 PLAIN - what is really happening inside

1. Why is knowledge stuck in pretraining? Two reasons, and both are about size.
2. Reason one: data volume. Pretraining is 15 trillion tokens. SFT is perhaps 500 million. That is a ratio of 30,000 to one. A signal that faint moves the weights very little.
3. Reason two: learning rate. SFT runs at roughly a tenth of pretraining's learning rate specifically so it does not damage what is there. It is a polish, applied deliberately gently.
4. So post-training is a small nudge from a small dataset. It changes behaviour because behaviour is a shallow property, expressible as a habit of which token to emit at the start of a response.
5. It does not change knowledge because knowledge is spread across billions of parameters built up over a trillion updates.
6. There is a useful piece of evidence for the shallowness. Work in 2024 and 2025 on what is called shallow alignment found that the difference between a base and an aligned model's output distribution is concentrated in the first handful of generated tokens. Get past the opening and the behaviours converge. That is also why prefilling attacks work.

49.8.5 TECHNICAL - the engineer's version

1. Evidence for the clean rule. The LIMA result, Zhou and colleagues, May 2023: 1,000 examples produced competitive chat behaviour from a 65B base model, which is hard to explain unless the capability was already present. InstructGPT, March 2022: a 1.3B instruction-tuned model was preferred by human raters over the 175B GPT-3 base, while remaining far weaker on knowledge benchmarks. Preference moved; knowledge did not.
2. Evidence for the fuzziness, which is the part usually left out. Gekhman and colleagues, 2024: fine-tuning on facts outside the model's pretraining is learned much more slowly than facts inside it, and increases hallucination rate on other questions as it is learned. So new facts can be installed, at a measurable cost.
3. RLVR on maths raises GSM8K and competition-maths accuracy substantially. Whether that is new capability or elicitation is genuinely disputed, and the pass-at-k results from Yue and colleagues in 2025 favour elicitation. Treat this as active research, not settled.
4. Mid-training deliberately erases the boundary. Ai2's Dolmino and Dolma 3 mixes apply a curated, high-quality corpus with a decaying learning rate after the main run. That is pretraining by mechanism and curation by intent.
5. Continued pretraining, sometimes called domain-adaptive pretraining, does install knowledge, because it uses pretraining-scale data. Billions of tokens of medical or legal text will change what a model knows. Millions will not. The threshold is roughly where you would expect from the ratios.
6. Practical rule with numbers attached:

Goal	Right tool	Data scale needed
Change tone or format	SFT	1,000 to 50,000 examples
Add a tool-call syntax	SFT	5,000 to 50,000 examples
Change refusal boundary	Preference tuning	10,000 comparisons
Add domain knowledge	Retrieval, not tuning	No training needed
Truly add a domain	Continued pretraining	1 to 100 billion tokens

7. The honest version: the sentence "post-training cannot install knowledge" is a statement about efficiency, not about possibility. Knowledge enters through data volume. Post-training datasets are too small to carry much of it, and the learning rate is deliberately set so they do not try.
8. One more caution about safety tuning specifically. It changes willingness to answer, not the presence of the answer. A model that refuses a question still contains whatever it learned in pretraining. Removing the refusal through fine-tuning, which as noted in section 49.5 takes about 100 examples, exposes it again.

49.8.6 WORDS - remember these

1. Alignment — making the model behave as intended — post-training that shapes output behaviour towards a specification of helpfulness and harmlessness.
2. Mid-training — a curated final stretch of pretraining — annealing on a high-quality mixture with a decaying learning rate.
3. Continued pretraining — more pretraining on a specific domain — the only fine-tuning-shaped method that reliably installs new knowledge.
4. Shallow alignment — the behaviour layer is thin — the base and aligned output distributions diverging mainly over the first few generated tokens.
5. Elicitation — bringing out something already there — improving measured performance without adding underlying capability.
6. Hallucination — confidently stating something false — high-confidence generation unsupported by training data or provided context.

49.9 Efficient adaptation: LoRA, QLoRA and distillation

49.9.1 PLAIN - in simple words

1. Suppose you want to adjust an existing model rather than train one.
2. The obvious way is **full fine-tuning**: load every weight, train them all.
3. It works. It is also astonishingly memory-hungry, because you need much more than just the weights in memory.
4. You need the weights, a high-precision master copy of them, one gradient per weight, and two extra running averages per weight that the optimizer keeps.
5. Roughly sixteen to eighteen bytes per parameter, against two bytes to just run the model. An eight-fold increase before you store anything else.
6. So a model you can happily run on one graphics card may need eight cards to fine-tune. That is the problem.
7. The fix that changed everything is called **LoRA**, low-rank adaptation.
8. The idea: freeze the original weights entirely. Do not touch them. Instead, add a small pair of extra matrices beside each big one, and train only those.
9. The pair is deliberately thin. A 4096-by-4096 matrix might be shadowed by a 4096-by-16 and a 16-by-4096 pair. That is 131,072 numbers instead of nearly 17 million.
10. Because the frozen weights never change, they need no gradients and no optimizer state. Only the tiny matrices do.

11. When you are done, you can either keep the small matrices as a separate file of a few tens of megabytes, or add them into the original weights to make a single ordinary model.
12. **QLoRA** goes further: squash the frozen original down to 4 bits per number as well. Now even the frozen part is a quarter of its size.
13. The result is that fine-tuning a 65-billion-parameter model became possible on one 48 GB card in 2023, where before it needed a small cluster.

49.9.2 PLAIN - a picture in your head

1. Think of a large printed reference book that you are not allowed to write in.
2. You want to correct and adapt it for your own use.
3. Full fine-tuning is reprinting the entire book with your changes. Correct, and it costs as much as the original printing.
4. LoRA is a pad of thin transparent overlay sheets. The book stays untouched. You write only your corrections on the overlays and lay them on top.
5. The overlays are tiny compared to the book, because most pages need nothing.
6. You can carry ten different sets of overlays for ten different jobs and swap them in seconds, sharing one copy of the book.
7. And if you want, you can eventually print a new edition with the overlays merged in. That is merging the adapter.

Where this comparison breaks: an overlay covers a page and hides what is under it. A LoRA adapter adds to the original numbers rather than replacing them, so the base behaviour is always still contributing. Also, an overlay can say anything, while a low-rank adapter is mathematically restricted in the kinds of change it can express. That restriction is the whole point, and it is also the method's main limitation.

49.9.3 PLAIN - a worked example

1. Memory arithmetic for a 7-billion-parameter model, computed here.

Method	What is stored	Memory
Inference, bf16	Weights only	about 14 GB
Full fine-tune, AdamW	16-18 bytes per param	about 112-126 GB
LoRA, bf16 base	Frozen base plus adapter	about 16-18 GB
QLoRA, 4-bit base	4-bit base plus adapter	about 6-8 GB

2. Where the 16 to 18 bytes comes from: 2 bytes of bf16 weight, 4 bytes of fp32 master copy, 2 to 4 bytes of gradient, and 8 bytes of Adam's two moments.
3. Read the consequences off the table. Full fine-tuning a 7B model needs two 80 GB cards at least. QLoRA fits in a single 8 GB consumer card at short sequence lengths, and comfortably in a 16 GB one.
4. LoRA parameter count, worked. Take a 4096-by-4096 attention projection. Full: 16,777,216 parameters. LoRA at rank 16: 4096 times 16 for the first matrix plus 16 times 4096 for the second, which is 131,072.
5. That is 0.78 per cent of the original. Applied to a whole 7B model at rank 16 on attention projections only, the trainable count lands around 4 to 20 million depending on which modules are targeted.
6. What the **rank** trades: higher rank means more capacity to express change, more memory, more risk of overfitting on a small dataset. Common values are 8, 16, 32 and 64. Rank 8 to 16 is enough for style and format. Rank 64 to 256 is used when the adaptation is substantial.

49.9.4 PLAIN - what is really happening inside

1. LoRA rests on one observation: when you fine-tune a model, the change to each weight matrix turns out to be well approximated by a low-rank matrix.

2. In plain terms, the update has far less independent structure than its size suggests, so it can be written as a thin matrix times another thin matrix.
3. So instead of learning the change directly, you learn the two thin factors.
4. During the forward pass, the layer computes the original output plus the adapter's output, scaled by a constant.
5. During the backward pass, gradients flow to the adapter only. The frozen weights receive none, which is where the memory saving comes from.
6. The first thin matrix is initialized randomly and the second to all zeros. That makes the adapter's contribution exactly zero at the start, so training begins from the base model's behaviour rather than from a random shove.
7. **Merging** means computing the product of the two thin matrices and adding the result into the original weight matrix. After merging, inference costs exactly what the original cost. Before merging, it costs slightly more.
8. QLoRA's addition: store the frozen base in 4 bits, dequantize each block to 16 bits on the fly as it is used, and keep the adapter in 16 bits. The quantization error sits in the frozen part, which is not being trained, so the adapter learns around it.

49.9.5 TECHNICAL - the engineer's version

1. LoRA: Hu and colleagues, Microsoft, 17 June 2021. Reported reduction of trainable parameters by 10,000 times and GPU memory by 3 times against full fine-tuning of GPT-3 175B with Adam, with no added inference latency after merging.
2. Hyperparameters that matter: `r` the rank; `lora_alpha` the scaling, where the adapter output is multiplied by alpha over r; `target_modules`, which projections get adapters; and `lora_dropout`. A common default is r 16, alpha 32, targeting the query, key, value and output projections.
3. Targeting the feed-forward projections as well as attention generally helps for larger behavioural changes and costs more memory.
4. QLoRA: Dettmers and colleagues, 23 May 2023. Three contributions. NF4, a 4-bit data type designed for normally distributed weights. Double quantization, quantizing the quantization constants themselves. Paged optimizers, using unified memory to survive memory spikes.
5. Their headline result: fine-tuning a 65B model on a single 48 GB GPU while matching 16-bit fine-tuning quality. Their Guanaco models reached 99.3 per cent of ChatGPT's score on the Vicuna benchmark after 24 hours on one GPU. Treat benchmark percentages of that era with caution; the memory result is the durable contribution.
6. Other parameter-efficient methods, with dates: adapter layers, Houlsby and colleagues 2019, small bottleneck blocks inserted in series, which do add inference latency; prefix tuning, Li and Liang 2021, learned key and value vectors prepended inside attention; prompt tuning, Lester and colleagues 2021, learned soft embedding vectors prepended to the input, needing as few as a few thousand parameters; IA3, Liu and colleagues 2022, learned per-channel scaling vectors.
7. **Distillation**: Hinton, Vinyals and Dean, 2015. Train a small student on the large teacher's output distribution rather than on hard labels, because the teacher's probabilities over wrong answers carry information about similarity structure. DistilBERT, Sanh and colleagues 2019, produced a model 40 per cent smaller and 60 per cent faster retaining about 97 per cent of BERT's performance.
8. Modern usage has shifted. Most "distillation" in the LLM era is sequence- level: generate outputs from a strong model and fine-tune a small one on the text. That is really SFT on synthetic data. True logit-level distillation is used inside labs; Google stated it for the Gemma 2 releases in 2024.
9. Note the licence trap. Several providers' terms have historically prohibited using their outputs to train competing models. Whether such a clause is enforceable is untested, but it is in the contract you accepted.
10. The decision table people actually need:

Situation	Do this	Why
Need current or private facts	Retrieval	Facts change; weights do not
Need a specific output format	Prompt first, then SFT	Cheapest that works
Need a house style everywhere	LoRA SFT	Too long for a prompt
Need a narrow task, cheap	Distil to a small model	Cuts serving cost
Need new domain knowledge	Continued pretraining	Needs billions of tokens

11. The ordering to follow in practice: prompt, then few-shot prompt, then retrieval, then LoRA fine-tune, then full fine-tune, then continued pretraining. Each step up costs roughly ten times the last. Stop at the first one that meets the requirement.
12. Tools: `peft` for LoRA and friends, `bitsandbytes` for 4-bit and 8-bit, `unsloth` for faster single-GPU runs, `axolotl` and `LLaMA-Factory` for configuration-driven training, and `vllm` with multi-LoRA serving to host many adapters against one base copy.

49.9.6 WORDS - remember these

1. Full fine-tuning — retraining every number in the model — updating all parameters, needing roughly 16 to 18 bytes per parameter of optimizer state.
2. LoRA — training small extra matrices instead of the whole model — low-rank adaptation with frozen base weights and trainable factorized updates.
3. Rank — how much change the adapter can express — the inner dimension r of the two low-rank factors, trading capacity against memory.
4. Merging — folding the adapter back into the model — adding the product of the low-rank factors into the base weight matrix.
5. QLoRA — LoRA on a squashed base model — 4-bit NF4 quantized frozen weights with 16-bit adapters, double quantization and paged optimizers.
6. Distillation — a small model copying a big one — training a student on a teacher's output distribution or generated text.
7. Adapter — a small trainable add-on module — any parameter-efficient block inserted into a frozen network, in series or in parallel.

49.10 What a training run looks like physically

49.10.1 PLAIN - in simple words

1. A frontier training run is not a program on a computer. It is one program spread across thousands of computers that must stay in step.
2. The accelerators sit eight to a chassis, the chassis stack into racks, the racks fill rows in a datacentre. Chapter 43 described that building.
3. Inside one chassis the eight accelerators are joined by a very fast private link. Between chassis they are joined by network cable, which is far slower.
4. That difference, roughly twenty times, decides how the work is divided.
5. The model may be too big to fit in one accelerator's memory. So it is split.
6. There are three ways to split it, and real runs use all three at once.
7. **Data parallelism**: every machine has a full copy of the model and gets a different slice of the batch. Cheap, simple, and needs every machine to share its gradients with every other after each step.
8. **Tensor parallelism**: one layer's matrix is cut into pieces across several accelerators. Very chatty; must stay inside the fast private link.
9. **Pipeline parallelism**: different machines hold different layers, like stations on an assembly line. Cheap in communication and leaves machines idle waiting for work, which is called the bubble.
10. On top of all that, hardware breaks. Constantly. A run of thousands of accelerators for two months will be interrupted hundreds of times.

11. So the whole thing is built around checkpoint and restart, not around hoping nothing fails.

49.10.2 PLAIN - a picture in your head

1. Think of a very large kitchen cooking one enormous banquet.
2. Data parallelism is eight identical kitchens each cooking the same menu for a different eighth of the guests, then meeting to agree the recipe changes.
3. Tensor parallelism is four chefs sharing one sauce, each stirring a quarter of the pan, shouting to each other constantly. They must be within earshot.
4. Pipeline parallelism is a line: one station chops, the next fries, the next plates. Efficient once running, and the fryer stands idle at the start.
5. The shouting is the problem. Chefs in the same room can shout. Chefs in different buildings must send messengers, and the messengers are the bottleneck.

Where this comparison breaks: kitchens degrade gracefully when one chef stops. A training run does not. Every accelerator must complete every step, because the gradient sum needs all of them. One dead card halts the entire cluster. That is why the failure and restart machinery matters so much.

49.10.3 PLAIN - a worked example

1. The three splits, drawn.

```
DATA PARALLEL: same model, different data
  GPU0 [full model] <- batch part 0
  GPU1 [full model] <- batch part 1
  GPU2 [full model] <- batch part 2
       after backward: all-reduce gradients across all GPUs
       cost: one full gradient exchange per step

TENSOR PARALLEL: one layer cut across GPUs
  layer weight W = [ W0 | W1 | W2 | W3 ]
  GPU0 holds W0, GPU1 holds W1, ...
  every forward and backward needs an all-reduce
  cost: several collectives PER LAYER. Keep inside a node.

PIPELINE PARALLEL: different layers on different GPUs
  GPU0: layers 0-7  -> GPU1: layers 8-15 -> GPU2: 16-23
  micro-batches flow left to right, then gradients right
  to left. Idle time at fill and drain = the "bubble".
  cost: one activation send per boundary. Cheapest.
```

2. A real configuration. Meta reported Llama 3.1 405B running on 16,384 H100s with four-way parallelism: tensor 8, context 1, pipeline 16, data 128.
3. Check the arithmetic: 8 times 1 times 16 times 128 equals 16,384.
4. Note where the 8 went. Tensor parallelism is exactly 8, which is the number of accelerators in one chassis sharing the fast private link. That is not a coincidence; it is the design constraint.

49.10.4 PLAIN - what is really happening inside

1. The memory problem, stated plainly. Training a 405-billion-parameter model in mixed precision needs roughly 16 bytes per parameter for weights, master copy, gradients and optimizer moments. That is about 6.5 terabytes.
2. An H100 has 80 GB. So you need at least 81 of them just to hold the state, before any activations, before any batch.
3. Four techniques reduce that.
4. **Sharding**: do not keep a full copy of the optimizer state and gradients on every machine. Split them and fetch pieces as needed. This is ZeRO in Microsoft's DeepSpeed, and FSDP in PyTorch.

5. **Activation checkpointing**: during the forward pass, throw away most of the intermediate values and recompute them during the backward pass. Trades about 30 per cent more compute for a large memory saving.
6. **Mixed precision**: store and multiply in 16-bit or 8-bit formats while accumulating in higher precision, halving or quartering memory traffic.
7. **Gradient accumulation**: process the batch in small pieces, add up the gradients, and only update once. This makes a large effective batch fit in small memory, at the cost of more steps.
8. When something fails, the sequence is: a collective operation times out, the job manager kills every process, a health check identifies the bad node, it is removed from the pool, and the job restarts from the last checkpoint.
9. Everything between the checkpoint and the failure is lost. Which is why the checkpoint interval is a direct trade between storage bandwidth and wasted compute, and is usually set at 15 minutes to an hour.

49.10.5 TECHNICAL - the engineer's version

1. Interconnect, with real figures.

Link	Bandwidth	Scope
NVLink 4 (H100 SXM)	900 GB/s per GPU	Inside one node
InfiniBand NDR	400 Gb/s = 50 GB/s	Between nodes
RoCEv2 400G Ethernet	400 Gb/s = 50 GB/s	Between nodes
PCIe Gen5 x16	about 64 GB/s	Host to device

2. Read the first two rows against each other. Inside a node, 900 GB/s. Between nodes, 50 GB/s. An 18-fold gap. That single ratio dictates that tensor parallelism, which communicates several times per layer, must stay inside a node, and pipeline or data parallelism crosses the network.
3. Meta built two 24,000-GPU clusters for Llama 3, one on InfiniBand and one on RoCE over Ethernet, and reported both performed acceptably. That was a notable result, because Ethernet had been considered unsuitable.
4. **ZeRO**, Rajbhandari and colleagues 2019 to 2020, in three stages: stage 1 shards optimizer state, stage 2 also shards gradients, stage 3 also shards parameters. Stage 3 is what PyTorch calls FSDP. Stage 3 gives the largest memory saving and the most communication.
5. **Activation checkpointing**, sometimes called gradient checkpointing, from Chen and colleagues 2016. Storing only every square-root-of-n-th activation reduces activation memory from order n to order square root of n at the cost of one extra forward pass.
6. **Mixed precision**, Micikevicius and colleagues 2017. bf16 has 8 exponent bits and 7 mantissa bits, the same dynamic range as fp32 with less precision, which is why it displaced fp16 for training: it rarely needs loss scaling. fp16 has 5 exponent and 10 mantissa bits and overflows more easily.
7. fp8 arrived with NVIDIA Hopper in 2022 in two forms, E4M3 and E5M2. Its first large public production use was DeepSeek-V3, December 2024, which trained a 671-billion-parameter mixture-of-experts model in fp8 with selective higher-precision accumulation.
8. Physical figures, computed here from published specifications: an H100 SXM is 700 W; 16,384 of them is 11.5 MW of accelerator draw alone. Add hosts, memory, storage and networking and the IT load is roughly 1.4 to 1.6 times that, so about 16 to 18 MW. Multiply by a datacentre power usage effectiveness of 1.1 to 1.2 and the facility draws roughly 18 to 22 MW.
9. Over the 78 days implied by 30.84 million GPU-hours on 16,384 GPUs, that is roughly 34 to 41 GWh, which is consistent with the 8,930 tonnes CO2eq Meta reported at typical grid carbon intensity.
10. For comparison, 20 MW is roughly the continuous electricity demand of a town of 15,000 to 20,000 homes in a temperate country.

11. Cooling: air cooling tops out around 30 to 40 kW per rack. An 8-way H100 node is about 10 kW, so four to five nodes per rack is the air-cooled limit. NVIDIA's GB200 NVL72 rack, shipping from 2025, draws around 120 kW and requires direct-to-chip liquid cooling. Chapter 43 covers the facility side.
12. Reliability, real reported figures. Meta reported 466 job interruptions across a 54-day snapshot of the 405B run. 47 were planned. 419 were unexpected, and around 78 per cent were confirmed or suspected hardware issues, with GPU faults the largest single category. Effective training time, meaning useful compute over elapsed time, was still above 90 per cent.
13. That is one unplanned interruption roughly every three hours across the cluster, and it is normal rather than exceptional.
14. Historical precedent: Meta published the OPT-175B logbook in 2022, a day-by-day account of a 992-A100 run with dozens of restarts, hardware replacements and loss divergences. It is the best public document on what this actually feels like.
15. Tools that observe it: `nvidia-smi` and DCGM for device telemetry, NCCL with `NCCL_DEBUG=INFO` for collective failures, Slurm or Kubernetes for job control, and per-step gradient-norm logging as the earliest warning that something is going wrong numerically.

49.10.6 WORDS - remember these

1. Data parallelism — every machine has the whole model, different data — batch sharding with an all-reduce of gradients each step.
2. Tensor parallelism — one layer split across machines — intra-layer sharding requiring collectives inside the forward and backward pass.
3. Pipeline parallelism — different layers on different machines — inter-layer sharding with micro-batches and an idle bubble at fill and drain.
4. All-reduce — everyone shares and everyone gets the total — a collective that sums a tensor across ranks and returns the sum to all of them.
5. ZeRO and FSDP — do not keep duplicate copies of training state — sharding of optimizer state, gradients and parameters across data-parallel ranks.
6. Activation checkpointing — forget and recompute to save memory — discarding intermediate activations and recomputing them in the backward pass.
7. bf16 — a 16-bit number format with wide range — 8 exponent and 7 mantissa bits, the default training precision since roughly 2020.
8. Bubble — idle time in a pipeline — accelerator cycles wasted while waiting for micro-batches to fill or drain the pipeline stages.

49.11 What weights physically are

49.11.1 PLAIN - in simple words

1. You asked directly what weights are, so here is the direct answer.
2. A model's weights are a large binary file full of numbers, and nothing else.
3. Not text. Not code. Not sentences. Not a database. A long run of bytes that a program interprets as numbers laid out in rectangles.
4. That file alone is useless. You need two more small things with it.
5. First, a **config**: a short text file saying how many layers there are, how wide they are, how many attention heads, what the vocabulary size is.
6. Without it, a program cannot tell where one rectangle ends and the next begins. The numbers are unlabelled without it.
7. Second, the **tokenizer**: the fixed list of text pieces and the rules for cutting text into them. Without it you cannot turn words into inputs.
8. So a published model is three things: a big number file, a small shape file, and a small vocabulary file. Everything else in the directory is optional.

9. The number file is often split into several parts, because a single 200 GB file is awkward to download and to resume. Then a small index file says which piece of the model lives in which part.
10. The size is arithmetic, not mystery. Multiply the parameter count by the bytes per number. Eight billion numbers at two bytes each is 16 GB.
11. There is no compression in the normal formats. The file is exactly as big as the numbers it holds, plus a header of a few kilobytes.

49.11.2 PLAIN – a picture in your head

1. Picture a warehouse full of numbered shelves, and one clipboard at the door.
2. The shelves hold the goods: billions of small identical items, in order.
3. The clipboard says: shelf 1 holds the vocabulary table, 128,256 rows by 4,096 columns. Shelf 2 holds layer zero's query matrix, 4,096 by 4,096.
4. Without the clipboard, the warehouse is a single undifferentiated pile.
5. With the clipboard, any item can be found by address in constant time.
6. That is exactly the layout of a modern weights file: a small header naming every block and giving its offset, then one enormous block of raw bytes.
7. The reason for that layout is that a program can then map the file into memory and read one block without loading the rest.

Where this comparison breaks: warehouse goods have individual meaning. These numbers do not. No single weight corresponds to a fact or a word. The meaning is entirely in the pattern across billions of them, which is why you cannot open a model file and read anything out of it.

49.11.3 PLAIN – a worked example

1. A real directory listing, from a published 8-billion-parameter Llama-family instruction model. Sizes rounded.

```
config.json                          about  800 B
generation_config.json               about  200 B
model-00001-of-00004.safetensors           4.98 GB
model-00002-of-00004.safetensors           5.00 GB
model-00003-of-00004.safetensors           4.92 GB
model-00004-of-00004.safetensors           1.17 GB
model.safetensors.index.json         about   24 KB
special_tokens_map.json              about  300 B
tokenizer.json                             9.09 MB
tokenizer_config.json                about   55 KB
LICENSE, README.md, USE_POLICY.md    a few KB
```

2. Total weights: about 16.1 GB, which is 8.03 billion parameters times 2 bytes for bf16. The arithmetic checks out exactly.
3. The four `.safetensors` files are the weights. The `index.json` maps each tensor name to the file it lives in. Everything else is under 10 MB.
4. The config, abbreviated but real in structure:

```
{
  "architectures": ["LlamaForCausalLM"],
  "hidden_size": 4096,
  "intermediate_size": 14336,
  "num_hidden_layers": 32,
  "num_attention_heads": 32,
  "num_key_value_heads": 8,
  "vocab_size": 128256,
  "max_position_embeddings": 131072,
  "rms_norm_eps": 1e-05,
```

```
  "rope_theta": 500000.0,
  "torch_dtype": "bfloat16"
}
```

5. Every number in that file is a hyperparameter chosen by a human before training. None of them was learned. Chapter 47 works through how these numbers produce the 8.03 billion total.
6. And here is what is actually inside the weights file: tensor names, shapes and types, as a loader would report them.

```
model.embed_tokens.weight              [128256, 4096]  BF16
model.layers.0.input_layernorm.weight  [4096]          BF16
model.layers.0.self_attn.q_proj.weight [4096, 4096]    BF16
model.layers.0.self_attn.k_proj.weight [1024, 4096]    BF16
model.layers.0.self_attn.v_proj.weight [1024, 4096]    BF16
model.layers.0.self_attn.o_proj.weight [4096, 4096]    BF16
model.layers.0.mlp.gate_proj.weight    [14336, 4096]   BF16
model.layers.0.mlp.up_proj.weight      [14336, 4096]   BF16
model.layers.0.mlp.down_proj.weight    [4096, 14336]   BF16
...  (the same 9 tensors for layers 1 through 31)
model.norm.weight                      [4096]          BF16
lm_head.weight                         [128256, 4096]  BF16
```

7. Note the key and value projections are 1024 rows, not 4096. That is grouped query attention: 8 key-value heads shared across 32 query heads.
8. Note the names are hierarchical strings. They are not a standard. They are the convention of one library, and a different library uses different names for the same tensors, which is why conversion scripts exist.

49.11.4 PLAIN - what is really happening inside

1. How a `safetensors` file is laid out, byte by byte.
2. First 8 bytes: a little-endian unsigned integer giving the length of the header in bytes.
3. Next N bytes: a JSON object. Each key is a tensor name. Each value gives the data type, the shape, and the start and end byte offsets of that tensor.
4. Everything after that: raw tensor bytes, back to back, no separators.
5. That is the entire format. It is deliberately boring, and the boringness is the security property.
6. Compare that with the format it replaced. PyTorch's `.bin` files used Python's `pickle`, which is not a data format but a program format: it contains instructions telling Python how to rebuild an object.
7. Those instructions can include "import this module and call this function". So loading a model file could run arbitrary code on your machine.
8. That is not theoretical. Malicious models with code hidden in pickle payloads have been found on public model hubs.
9. `safetensors` cannot do that, because a JSON header plus raw bytes has no way to express an instruction.
10. Because the offsets are known in advance, the file can also be memory-mapped and tensors copied straight to the accelerator with no intermediate parse. That is where the speed comes from.

49.11.5 TECHNICAL - the engineer's version

1. The format comparison you should be able to give from memory:

Format	Code execution risk	Main use
PyTorch .bin (pickle)	Yes, arbitrary	Legacy checkpoints
safetensors	No	Default for published weights
GGUF	No	llama.cpp local inference
ONNX	No, but graph is code-like	Cross-runtime deployment

Format	Code execution risk	Main use

2. `safetensors` was created at Hugging Face and became the default published format across the hub during 2023. Design goals in the project's own words: safe against code execution, zero-copy, memory-mappable, and fast.
3. The pickle risk was severe enough that PyTorch changed the default. From PyTorch 2.6, released January 2025, `torch.load` defaults to `weights_only=True`, which refuses to execute arbitrary globals. Before that the default was unsafe and most tutorials did not mention it.
4. **GGUF** is the format of the `llama.cpp` and `ggml` ecosystem, introduced in August 2023 to replace the earlier GGML, GGMF and GGJT formats. Its defining feature is a typed key-value metadata block, so new fields can be added without breaking old readers.
5. GGUF is single-file and self-contained: architecture metadata, tokenizer vocabulary and merges, and quantized tensors all live in the one file. That is why a GGUF download needs no config or tokenizer alongside it.
6. GGUF bakes quantization into the file. The type names you will see include F32, F16, BF16, Q8_0, Q6_K, Q5_K, Q4_K, Q4_0, and the IQ series such as IQ4_XS and IQ2_XXS. Around 40 types are defined. Chapter 47 covers what the K-quants actually do.
7. **ONNX**, Open Neural Network Exchange, was launched by Microsoft and Facebook in September 2017. It stores a computation graph plus weights in a Protocol Buffers message, so a model trained in one framework can run in another. It does not execute Python, but it does describe operations, and custom operators can load native code, so it is not risk-free in the way safetensors is.
8. Practical file-size arithmetic, which you can do in your head:

Precision	Bytes per parameter	8B model file
FP32	4	about 32 GB
BF16 or FP16	2	about 16 GB
Q8_0	about 1.06	about 8.5 GB
Q4_K_M	about 0.60	about 4.9 GB

9. Quantized types are slightly larger than the nominal bit count suggests because each block of 32 to 256 weights carries a scale, and sometimes an offset, stored at higher precision.
10. What is emphatically not in a weights file: the training data, the data mixture, the training code, the hyperparameters used during training, the optimizer state, the random seed, or any record of how the numbers arrived.
11. Commands that inspect these files: `safetensors` exposes a Python API where `safe_open(path, framework="pt")` lists keys without loading data; `gguf-dump` prints GGUF metadata and tensor tables; `python -m onnx.checker` validates an ONNX graph; and `huggingface-cli scan-cache` shows what you have downloaded and how much disk it is using.

49.11.6 WORDS - remember these

1. Weights file — the big file of numbers — a binary container of tensors, sized at parameter count times bytes per parameter.
2. Config — the shape description — a JSON file of architecture hyperparameters needed to interpret the tensor layout.
3. Tensor name — the address of one rectangle of numbers — a hierarchical string key such as `model.layers.7.mlp.up_proj.weight`.
4. safetensors — the safe modern weights format — an 8-byte length, a JSON header of dtypes, shapes and offsets, then raw bytes.

5. Pickle — Python's object format that can run code — the mechanism behind legacy `.bin` checkpoints and their arbitrary-code-execution risk.
6. GGUF — a single self-contained file for local inference — metadata, tokenizer and quantized tensors together, used by llama.cpp.
7. Memory mapping — reading a file without copying it into memory first — using the operating system's virtual memory to page tensor bytes on demand.

49.12 Open weights versus open source

49.12.1 PLAIN - in simple words

1. This is the part of the chapter where the words are genuinely abused, so we will be slow and exact.
2. There are four separate things people mix together. Each is independent of the others. A model can have any combination.
3. **Open weights** means the number file can be downloaded and run. That is all it means. Nothing about permission, nothing about data.
4. **Open source** means something specific. It is a definition maintained by the Open Source Initiative since 1998, and it is about the licence granting you freedom to use, study, modify and share, without restrictions on who you are or what field you work in.
5. **Open data** means the training corpus itself is published, so somebody else could rebuild the model rather than just run it.
6. **The licence terms** are separate again. A model can be open-weight and carry a licence that forbids commercial use, or forbids competitors, or forbids certain applications.
7. The abuse works like this. A company publishes only the weights, under a licence with restrictions, and calls the result open source.
8. Almost nothing about it is open source. The source, in the sense that matters, would be the data and the training code, and neither was released.
9. A more honest word, which some companies now use, is "open weights".
10. Why does the distinction matter practically? Because you cannot verify what is in a model you cannot inspect the data for, you cannot reproduce it, and you may find at deployment time that the licence forbids your use.

49.12.2 PLAIN - a picture in your head

1. Think about a restaurant and a famous sauce.
2. Selling you a jar of the sauce is open weights. You have the thing. You can taste it, use it, put it on your food.
3. Publishing the recipe is open source. Now you can make it, change it, improve it, and check whether the ingredients are what they claimed.
4. Publishing where every ingredient was bought is open data.
5. And the label on the jar might say "not for resale" or "not for use in commercial kitchens". That is the licence.
6. Selling the jar and calling it "open source sauce" is the thing being objected to. You cannot make the sauce. You cannot check the ingredients. You have a jar.

Where this comparison breaks: a sauce could be reverse-engineered by a skilled cook. Model weights cannot be reverse-engineered into their training data in any practical sense. Also, the jar of sauce is not itself useful for making more sauce, whereas model weights genuinely are useful for building further models, which is why open weights deliver real value even without the recipe.

49.12.3 PLAIN - a worked example

1. What real models actually release. Status as of mid-2026.

Model family	Weights	Training data published
OLMo 2, OLMo 3 (Ai2)	Yes	Yes, full corpus
Pythia (EleutherAI)	Yes	Yes, the Pile
BLOOM (BigScience)	Yes	Yes, ROOTS corpus
Llama 3.1, Llama 4 (Meta)	Yes	No
Qwen3 (Alibaba)	Yes	No
DeepSeek-V3, R1	Yes	No
gpt-oss (OpenAI)	Yes	No
Gemma (Google)	Yes	No
GPT-5, Claude, Gemini	No	No

2. And the licences, which is a separate question from the table above.

Model family	Licence	Key restriction
OLMo, Pythia	Apache 2.0	None of substance
Qwen3, gpt-oss	Apache 2.0	None of substance
DeepSeek-R1	MIT	None of substance
Mistral 7B, Mixtral	Apache 2.0	None of substance
Llama 3.1, Llama 4	Meta Community	700m users, AUP, naming
Gemma	Gemma Terms	Prohibited use policy
BLOOM, StarCoder2	OpenRAIL-M	Use-case restrictions

3. Read those two tables together and the picture is clear. Full openness, meaning weights plus data plus code plus a permissive licence, exists but is rare, and comes mostly from research institutes rather than from companies.
4. The single most-cited restriction, from Meta's Llama licences: if your products had more than 700 million monthly active users in the month before the release date, you must ask Meta for a separate licence.
5. In practice that clause names a handful of companies. It is aimed at competitors, and its presence is precisely why the licence is not open source: open source licences cannot discriminate against a class of user.
6. Meta's licences also require the words "Built with Llama" on derivative products and require derived model names to begin with "Llama", and bind you to an Acceptable Use Policy that can be updated after you accept it.
7. Llama's multimodal models have additionally carried a restriction excluding users domiciled in the European Union, with no reason given in the licence.

49.12.4 PLAIN - what is really happening inside

1. Here is what is normally withheld even by generous open-weight releases.
2. **The training data.** Which documents, from where, in what proportion.
3. **The data mixture.** Even when categories are named, the recipe is not.
4. **The training code.** The actual distributed training harness, the data loader, the parallelism configuration.
5. **The hyperparameters.** Learning rate schedule, batch ramp, warmup, weight decay, initialization, and the dozens of small choices that make a run work.

6. **The post-training recipe.** The SFT data, the preference data, the reward model, the safety data. This is often guarded more closely than the pretraining recipe, because it is where the product feel comes from.
7. **The evaluation and ablation results** that led to those choices.
8. What you get instead is a file that reproduces one point in that space and no way to get to any neighbouring point except by guessing.
9. There are good reasons for some of this. Some data cannot legally be redistributed. Some of it is licensed under terms that forbid it. Some of it would expose the lab to litigation of the kind section 49.2 described.
10. And there are competitive reasons for the rest, which nobody disputes.

49.12.5 TECHNICAL - the engineer's version

1. Precise definitions, in the terms the field actually uses.
2. **Open weights**: model parameters are distributed for download. Says nothing about licence, data, code or reproducibility. Sometimes called "open access" or, unhelpfully, "open model".
3. **Open source**: the Open Source Definition, maintained by the Open Source Initiative since 1998, requires among other things free redistribution, availability of source code, permitted derived works, no discrimination against persons or groups, and no restriction on fields of endeavour. Those last two are the clauses most model licences fail.
4. **Open data**: the training corpus is published under a licence permitting use. Examples: the Pile from EleutherAI in 2020, ROOTS from BigScience in 2022, Dolma and Dolma 3 from Ai2, FineWeb from Hugging Face under ODC-By 1.0.
5. **Licence class**, in three tiers you should be able to name: permissive open source, meaning Apache 2.0 or MIT; restricted custom licences, meaning Meta's Community licences and Google's Gemma Terms, which are contracts with conditions; responsible-AI licences, meaning the RAIL and OpenRAIL family from 2022 onwards, which are explicitly use-restricted by design and were never claimed to be open source by their authors.
6. Fully open releases, with what each actually shipped: Pythia, EleutherAI 2023: weights at 8 sizes, the Pile as training data in exact training order, training code, and 154 intermediate checkpoints per model, under Apache 2.0. Built for reproducible research on training dynamics, and still the reference example. OLMo, Ai2, February 2024, then OLMo 2 in January 2025 and OLMo 3 in November 2025: weights, full training data, training code, evaluation harness, logs and intermediate checkpoints from every stage. OLMo 3's release included Dolma 3, a 9.3-trillion-token corpus, the 5.9 trillion token pretraining mix used for the 7B and 32B base models, the post-training data suite, and the data tooling. BLOOM, BigScience 2022: 176B parameters, the ROOTS corpus, the training code, and a detailed record of the process, released under a RAIL licence that carries use restrictions and therefore is not OSI open source.
7. Open-weight, closed-data, permissive licence, which is the commercially dominant pattern: Mistral 7B (September 2023, Apache 2.0), Qwen3 (2025, Apache 2.0), DeepSeek-R1 (January 2025, MIT), and OpenAI's gpt-oss-120b and gpt-oss-20b (5 August 2025, Apache 2.0).
8. gpt-oss is worth a note because it shows the pattern clearly. Apache 2.0 on the weights, plus published inference reference code, the harmony prompt renderer and the o200k_harmony tokenizer. No training data, no training code. Permissive licence, closed recipe.
9. Established fact: open weights allow local running, fine-tuning, adapter training, quantization, red-teaming and independent measurement. Established fact: they do not allow reproduction, data auditing, or verification of any claim about what the model was trained on. Marketing claim: describing a weights-only release under a restricted licence as "open source". This is the specific abuse the reader asked about.
10. One practical consequence for engineers. Open weights are the only category that survives a vendor withdrawing a model. A downloaded file keeps working when an API endpoint is deprecated, and API deprecation happens on a twelve-to-twenty-four-month cycle at every major provider.

49.12.6 WORDS - remember these

1. Open weights — the numbers are downloadable — parameters published for use, independent of licence, data or code availability.
2. Open source — a licence granting the four freedoms without discrimination — as defined by the Open Source Initiative's Open Source Definition since 1998.
3. Open data — the training corpus is published — the actual documents, or a reproducible recipe for assembling them, under a usable licence.
4. Permissive licence — almost no conditions — Apache 2.0 or MIT, allowing commercial use, modification and redistribution.
5. Community licence — a contract with conditions attached — Meta's and Google's custom terms, with user thresholds, naming rules and use policies.
6. OpenRAIL — a licence that restricts uses by design — the Responsible AI Licence family, openly not open source and honest about it.
7. Open washing — calling a restricted release open — marketing a weights-only or use-restricted model as open source.

49.13 The Open Source Initiative's position

49.13.1 PLAIN - in simple words

1. The Open Source Initiative is the body that has maintained the definition of open source software since 1998, and it decides which licences qualify.
2. For twenty-five years that definition was about source code. Models are not source code, so it did not obviously apply.
3. So the OSI ran a public process with many contributors and published the **Open Source AI Definition version 1.0** on 28 October 2024.
4. It starts from the same four freedoms as software: use the system for any purpose, study how it works, modify it, and share it, modified or not.
5. To exercise those freedoms you need what it calls the preferred form to make modifications, and it says that means three things must be available.
6. **Data information**: enough detail about the training data that a skilled person could build a substantially equivalent system.
7. **Code**: the complete source code used to train and run the system, under an OSI-approved licence.
8. **Parameters**: the weights and other settings, under OSI-approved terms.
9. Read point 6 carefully, because it is the compromise at the heart of the whole thing. It does not require publishing the data itself. It requires publishing enough information about it.
10. By this definition, most models marketed as open source do not qualify.
11. The OSI has said so directly, naming names.

49.13.2 PLAIN - a picture in your head

1. Think about a food label and a recipe again, one last time.
2. The strictest possible rule would be: hand over every ingredient you used, physically, so anyone can remake the dish exactly.
3. That is impossible for some ingredients. You cannot hand over a bottle of wine you drank, and you cannot hand over a book you licensed for one use.
4. So the OSI settled on: publish the recipe in enough detail that a competent cook could go and buy equivalent ingredients and make substantially the same dish.
5. That satisfies nobody entirely. Purists say a recipe is not the ingredients. Companies say even the recipe is a trade secret.
6. And it is precisely because that middle position annoys both sides that it is worth understanding exactly.

Where this comparison breaks: two cooks following the same recipe get similar results. Two labs following the same data description do not, because model outcomes depend on ordering, seeds, hardware numerics and hundreds of unpublished choices. "Substantially equivalent" is doing a great deal of work in that sentence and nobody has tested it in practice at frontier scale.

49.13.3 PLAIN - a worked example

1. The OSI published its own assessment of real models against the definition. Its stated results, which it describes as part of the definitional process rather than as certifications:

Verdict	Models named by the OSI
Passes	Pythia, OLMo, T5
Passes	Amber, CrystalCoder
Would pass if relicensed	BLOOM, StarCoder2, Falcon
Does not pass	Llama 2, Grok, Phi-2, Mixtral

2. Note that Mixtral is in the failing list despite being under Apache 2.0. The licence is fine. The data information is missing. Both are required.
3. Note that BLOOM is in the middle list despite publishing its full training corpus. The data is fine. The RAIL licence restricts uses. Both are required.
4. Those two rows together are the clearest possible illustration that open weights, open data and an open licence are three independent things.
5. The OSI's specific objections to Meta's Llama licences: it fails freedom zero, the freedom to use the system for any purpose; it fails the Open Source Definition's point 5 by discriminating against a class of user via the 700-million-user threshold; and it fails point 6 by restricting fields of endeavour through the Acceptable Use Policy. The OSI has also pointed to the exclusion of European Union users from certain models, unexplained.
6. The OSI uses the term "open washing" for marketing a restricted release as open source.

49.13.4 PLAIN - what is really happening inside

1. Now the counter-arguments, given fairly, because this is a live dispute and not a settled question.
2. **Training data cannot always be redistributed.** Much of it is copyrighted text that the lab has a defensible claim to train on but no right to republish. Publishing it would be a separate and much larger legal exposure.
3. **Some of it must not be redistributed.** Web-scale corpora contain personal data. Publishing the corpus can conflict with privacy law even where training on it does not.
4. **Open weights deliver most of the practical benefit.** With weights you can run locally, fine-tune, quantize, audit behaviour, red-team, and avoid depending on a vendor. Very few people were ever going to retrain a frontier model even with the full data.
5. **Reproduction is impossible anyway at this scale.** Even with the corpus, rebuilding a frontier model costs tens of millions of dollars, so the freedom being protected is theoretical for almost everyone.
6. And there is a criticism from the opposite direction, which is important.
7. Some free-software advocates argued the OSI's compromise is too weak: that allowing "data information" instead of data lets companies claim the label without the substance, which is the very open washing the definition was meant to stop.
8. So version 1.0 is attacked from both sides: too strict for industry, too loose for purists. The OSI itself has said the definition will be revised.
9. The honest version: there is no consensus. The OSI has the strongest claim to define "open source" because it has defined it for software since 1998, but no legal authority, and companies are free to ignore it and do.

49.13.5 TECHNICAL - the engineer's version

1. What the definition requires in full, as three components. Data information: complete descriptions of all data used, including provenance, scope and characteristics, how it was obtained and selected, labelling procedures, and cleaning and filtering methodologies, in enough detail that a skilled person can build a substantially equivalent system. Code: the complete source used for data processing and filtering, training, with all training arguments, validation, testing and inference, under OSI-approved licences. Parameters: model weights and other configuration settings, including intermediate checkpoints and final optimizer states, under OSI-approved terms.
2. That third clause is stricter than people notice. Intermediate checkpoints and optimizer state are almost never published. Pythia and OLMo do publish checkpoints, which is exactly why they pass.
3. Timeline of the dispute, dated: 1998, the Open Source Definition is published. 2022, RAIL and Open-RAIL licences appear, explicitly use-restricted. February 2023, Meta releases LLaMA 1 weights to researchers on request. July 2023, Llama 2 is released under a community licence and widely described in press coverage as open source. 2023 to 2024, the OSI runs a co-design process with contributors from industry, academia and civil society. 28 October 2024, Open Source AI Definition 1.0 is published at the All Things Open conference. Through 2025 and 2026, the argument continues without resolution, with further drafts under discussion.
4. Parallel regulatory pressure, which may matter more than the definitional argument in the end. The European Union's AI Act, in force from 1 August 2024 with general-purpose model obligations applying from 2 August 2025, requires providers to publish a sufficiently detailed summary of training content and to have a copyright policy. It also provides partial exemptions for models released under free and open-source licences, which gives the definition of "open" direct legal consequences.
5. So a term that was a marketing argument in 2023 became a compliance question in 2025. That is the practical reason to get it right.
6. Established fact: the OSAID 1.0 text, its date, its three required components, and the OSI's published assessments of named models. Active dispute: whether the data-information compromise is correct, whether version 1.0 will hold, and whether regulators will adopt it. Marketing claim: any company statement that its weights-only release "is open source" without addressing data information or licence restrictions.

49.13.6 WORDS - remember these

1. OSAID — the open source rulebook for AI — the Open Source AI Definition 1.0, published by the OSI on 28 October 2024.
2. Data information — a description good enough to rebuild from — sufficiently detailed data documentation to build a substantially equivalent system.
3. Four freedoms — use, study, modify, share — the freedoms the definition requires an open source AI system to grant without restriction.
4. OSI-approved licence — a licence on the OSI's published list — the test a licence must pass to be called open source.
5. Open washing — claiming openness you did not deliver — marketing a restricted or weights-only release using the open source label.

49.14 Model releases in practice

49.14.1 PLAIN - in simple words

1. When a model is released, the file is the smallest part of the event.
2. Around it sit documents, gates and processes, and knowing what they should contain lets you judge a release in a few minutes.
3. The **model card** is the main document. The idea comes from a 2019 paper by Mitchell and colleagues, and it is now near-universal practice.

4. It should say: what the model is for, what it is not for, what it was trained on in general terms, how it scores on named benchmarks, what its known limitations are, and what licence applies.
5. The **system card** is a bigger document about the deployed product rather than the raw model: what safety testing was done, what was found, what mitigations were added, and what risks remain.
6. **Staged release** means publishing in steps rather than all at once, to watch what happens.
7. **Red-teaming** means paying people to attack the model before release and trying to make it do what it should not.
8. **Watermarking** means trying to mark generated output so it can be detected later. It works better for images than for text.
9. And a **licence gate** on a hosting site means you must accept terms, and often give your name and email, before you can download.
10. What is missing from a release tells you as much as what is present.

49.14.2 PLAIN - a picture in your head

1. Think of the paperwork that comes with a new medicine.
2. There is a leaflet saying what it treats, the dose, who must not take it, and the side effects observed in trials. That is the model card.
3. There is a much longer regulatory dossier describing every trial, its results, and the risks that remain. That is the system card.
4. There is a phased rollout, first to a limited group, then wider. That is the staged release.
5. And there is a pharmacist who checks who you are before handing it over. That is the licence gate.
6. Now imagine a medicine arriving with no leaflet at all, and ask yourself what that absence tells you.

Where this comparison breaks: medicines are regulated by law with mandatory content and legal consequences for omissions. Model cards are, in most places and as of mid-2026, entirely voluntary in content and quality. The European Union's AI Act is the first serious attempt to make parts of this mandatory, and it applies to general-purpose models only from 2 August 2025.

49.14.3 PLAIN - a worked example

1. The checklist to run against any release. Present or absent, and what the absence means.

Item	If missing, that means
Named training-data sources	You cannot audit or trust claims
Knowledge cutoff date	You cannot judge staleness
Benchmark results with names	Comparisons are unverifiable
Stated out-of-scope uses	No thought given to misuse
Compute and energy figures	Cost and impact unassessable
Licence text, not a summary	Read the actual terms first
Evaluation of known weaknesses	Only wins are being reported

2. Real examples of each end of the spectrum. Meta's Llama 3.1 model card gives GPU-hours, energy, carbon, knowledge cutoff of December 2023, benchmark tables and licence, but describes training data only as "publicly available sources" with category percentages.
3. Ai2's OLMo releases give all of the above plus the corpus itself, the data tooling, the training logs and intermediate checkpoints.
4. Closed model releases typically give benchmark tables and a system card, and no training data information at all.

49.14.4 PLAIN - what is really happening inside

1. What happens in the weeks before a release, roughly in order.
2. Internal evaluation on held-out benchmarks, plus contamination checks to confirm the benchmarks were not in the training data.
3. Capability evaluations against a published safety framework, testing for dangerous capabilities in specific named categories.
4. Red-teaming, both internal and by hired external groups, attempting jailbreaks and misuse. Findings feed back into more safety fine-tuning.
5. Sometimes external review, by a government safety institute or a contracted evaluation organization, before release.
6. Then a decision, then the documents, then the gate, then publication.
7. **Staged release** has a clear historical example. In February 2019 OpenAI announced GPT-2 and released only the 124-million-parameter version, citing misuse concerns. It released 355M in May, 774M in August, and the full 1.5B model in November 2019, publishing its reasoning at each step.
8. That decision was heavily criticized at the time as overcautious, and it set the template that every lab now follows in some form.
9. Note what staged release cannot do for open weights. Once the file is downloaded it cannot be recalled, and as section 49.5 noted, the safety training can be removed by fine-tuning for a few hundred dollars.

49.14.5 TECHNICAL - the engineer's version

1. Model cards: proposed in "Model Cards for Model Reporting", Mitchell and colleagues, 2019. The companion idea for data is "Datasheets for Datasets", Gebru and colleagues, 2018. Both are now conventions rather than standards, though Hugging Face enforces a minimal card structure on its hub.
2. System cards: OpenAI published the GPT-4 System Card in March 2023, separating deployment risk analysis from the technical report. Anthropic and Google now publish equivalents for major releases. Contents typically include red-team findings, capability evaluations against a framework, refusal rates, and known failure modes.
3. Named safety frameworks you should recognize, with dates: Anthropic's Responsible Scaling Policy, September 2023; OpenAI's Preparedness Framework, December 2023; Google DeepMind's Frontier Safety Framework, May 2024. All define capability thresholds that trigger additional controls. All are voluntary commitments, not regulation.
4. Government bodies: the UK AI Safety Institute was announced in November 2023 and renamed the AI Security Institute in February 2025. The US AI Safety Institute was established in 2023 and renamed the Center for AI Standards and Innovation in June 2025. Both have run pre-deployment evaluations under agreements with major labs.
5. Watermarking. Google DeepMind's SynthID for images was announced in August 2023, and SynthID-Text was published in the journal Nature in October 2024, using a tournament sampling scheme that biases token choice in a detectable pattern. Google reported it does not measurably degrade quality.
6. The honest position on text watermarking: it is detectable while the text is unedited, and it degrades under paraphrasing, translation or moderate editing. It also requires the generator to cooperate, so it does nothing about open-weight models where the sampler can simply be replaced. Treat claims of robust text watermarking as marketing.
7. The C2PA standard, from a coalition including Adobe and Microsoft, takes the other approach: cryptographically signed provenance metadata attached to a file. It survives editing by signed tools and is destroyed by screenshots.
8. Licence acceptance gates: Hugging Face supports gated repositories, where a user must accept terms and often supply contact details, with optional manual approval. Meta and Google both use them. The gate is a contractual mechanism, not a technical one; it does not prevent redistribution.
9. Regulatory contents, dated. Under the European Union's AI Act, providers of general-purpose models have obligations from 2 August 2025 including technical documentation, information for downstream

providers, a copyright policy, and a sufficiently detailed public summary of training content, with partial exemptions for free and open-source releases and additional duties above the 10-to-the-25-FLOPs systemic-risk threshold.

10. Practical advice: read the licence file itself, not the summary on the model page; check the knowledge cutoff before relying on recency; check whether the card names benchmarks or just claims performance; and check whether evaluation numbers are self-reported or independently reproduced.

49.14.6 WORDS - remember these

1. Model card — the leaflet that comes with a model — a structured document of intended use, training description, evaluations, limitations and licence.
2. System card — the safety dossier for the deployed product — red-team findings, capability evaluations, mitigations and residual risks.
3. Staged release — publishing in steps to watch the effect — incremental release of larger checkpoints or wider access over time.
4. Red-teaming — paying people to attack it before release — structured adversarial testing for jailbreaks, misuse and dangerous capability.
5. Watermarking — marking generated output so it can be detected — biased sampling for text, or signed provenance metadata for media.
6. Gated repository — you must accept terms to download — a hosting-site mechanism collecting acceptance and often identity before access.

49.98 Common wrong ideas

1. Wrong: the model was trained on the internet, so it knows everything. Right: it was trained on a heavily filtered slice of a web crawl, plus books, code and papers, and typically 80 to 95 per cent of the crawl was discarded. It has a knowledge cutoff date, it never saw most paywalled or private material, and what it did see it stored lossily as statistical patterns rather than as retrievable text.
2. Wrong: fine-tuning teaches it new facts. Right: fine-tuning teaches format, style and behaviour. Knowledge comes from pretraining because knowledge comes from data volume, and a fine-tuning set is perhaps one thirty-thousandth the size of the pretraining set. Fine-tuning on unfamiliar facts is learned slowly and measurably increases hallucination elsewhere. Use retrieval for facts.
3. Wrong: open weights means open source. Right: they are separate. Open weights means the numbers are downloadable. Open source, by the Open Source Initiative's definition of October 2024, additionally requires the training code, sufficiently detailed data information, and a licence with no discrimination against users or fields of use. Mixtral is Apache 2.0 and fails; BLOOM publishes its corpus and fails.
4. Wrong: RLHF makes the model truthful. Right: it makes the model produce answers that human raters prefer. Where raters prefer confident, agreeable, long or well-formatted answers, that is what gets reinforced. This is why sycophancy and length inflation are standard failure modes, and why OpenAI had to roll back a GPT-4o update in April 2025 for exactly this reason.
5. Wrong: training and running cost about the same. Right: they differ by roughly thirteen orders of magnitude per query. Training an 8-billion model on 15 trillion tokens is about 7.2e23 FLOPs. Generating one token from it is about 1.6e10 FLOPs. Training equals roughly 45 trillion generated tokens. For a busy service, cumulative inference overtakes training cost within months.
6. Wrong: a bigger model is always better. Right: Chinchilla in 2022 showed a 70-billion model beating a 280-billion one at equal compute because it saw four times more data. Data quantity, data quality and post-training all matter as much as parameter count.
7. Wrong: labs publish what they trained on. Right: GPT-3 published a mixture table in 2020. Almost nobody does now. GPT-4's technical report explicitly declined. Any claim about a closed model's training data, including a claim about what it did not include, is unverifiable from outside.
8. Wrong: the safety training is part of the model and cannot be removed. Right: for open weights it can be substantially removed by fine-tuning on a few hundred adversarial examples, for a few dollars.

Safety behaviour is a thin post-training layer over an unchanged base capability.

9. Wrong: a training run is one program on one big computer. Right: it is thousands of accelerators running in lockstep, split three ways across data, tensors and layers, and hardware fails constantly. Meta reported 419 unexpected interruptions in 54 days on one run.
10. Wrong: the weights file contains the training data somewhere inside it. Right: it contains only numbers arranged in named tensors, plus a config and a tokenizer. Verbatim regurgitation of memorized training text does happen, especially for text duplicated many times in the corpus, but the data is not stored or retrievable in any ordinary sense.

49.99 Chapter summary in 20 lines

1. A model is made in a chain: collect data, clean it, train a tokenizer, pretrain, post-train, evaluate, release. Pretraining is 95 to 99 per cent of the compute and post-training is almost all of the visible behaviour.
2. Data comes from web crawls such as Common Crawl, which published 2.42 billion pages and 419 TiB uncompressed in its July 2025 archive alone, plus books, code, papers, licensed corpora and synthetic text.
3. Cleaning discards most of it: URL blocking, text extraction, language identification, exact and MinHash fuzzy deduplication, heuristic and classifier quality filters, toxicity and PII removal, and benchmark decontamination, then deliberate mixing ratios over the survivors.
4. FineWeb's published pipeline uses 5-gram shingles with 14 bands of 8 hashes, which catches near-duplicates above about 0.75 Jaccard similarity 77 per cent of the time and above 0.9 essentially always.
5. Copyright is unsettled and actively litigated. Judge Alsup held in June 2025 that training on lawfully bought books was fair use while retaining pirated copies was not, and Anthropic settled for a reported 1.5 billion dollars.
6. Most labs stopped publishing their data mixture after GPT-3 in 2020, so claims about a closed model's training data are unverifiable from outside.
7. Pretraining has exactly one objective: next-token prediction, scored by cross-entropy loss in nats, starting at the natural log of the vocabulary size and typically ending between 1.4 and 2.0.
8. One step is a forward pass over a batch of millions of tokens, a loss, a backward pass costing about twice the forward, and an AdamW update. Llama 3 405B used a peak learning rate of 8e-5, 8,000 warmup steps and cosine decay to 8e-7 over 1.2 million steps.
9. Llama 3.1 405B is the best-documented real run: over 16,384 H100s, 30.84 million GPU-hours, about 15.6 trillion tokens, roughly 78 days of wall clock and an estimated 62 to 123 million dollars of rented compute.
10. Training compute is approximately 6ND, parameters times tokens times six. It reproduces GPT-3's reported 3.14e23 and Llama 3.1 405B's 3.8e25, and the European Union's AI Act uses 10 to the 25 as a systemic-risk threshold.
11. Kaplan and colleagues in January 2020 found smooth power laws and advised spending on size; Hoffmann and colleagues in March 2022 showed models were badly undertrained and that about 20 tokens per parameter is optimal.
12. Since 2023 labs deliberately overtrain small models far past that, because inference cost dominates over a deployed model's life. Llama 3.1 8B was trained at about 1,875 tokens per parameter, roughly 94 times Chinchilla.
13. A base model continues text and cannot be used as a chatbot: no turns, no roles, no stopping rule and no refusal behaviour. Ask it a question and it will often produce more questions.
14. SFT fixes that with tens of thousands to a few million demonstration pairs, a chat template of special tokens such as Llama 3's `<|eot_id|>`, loss masked to the assistant's tokens, and a learning rate ten times lower.
15. Preference learning goes further: collect comparisons, train a reward model, optimize with PPO under a KL penalty. Its characteristic failure is reward hacking, exemplified by OpenAI's sycophantic GPT-4o update of April 2025.

16. DPO in May 2023 removed the separate reward model algebraically and most smaller teams moved to it; constitutional and RLAIF methods replace human labels with written principles and an AI judge; RLVR uses checkable answers in maths and code and is the engine behind reasoning models.
17. Post-training shapes behaviour and cannot efficiently install knowledge, because the data is thirty thousand times smaller and the learning rate is set gently on purpose. The boundary is real but fuzzy, and mid-training blurs it deliberately.
18. LoRA freezes the base and trains thin low-rank matrices, cutting a 7B fine-tune from about 112 GB to about 17 GB; QLoRA adds a 4-bit base and cuts it to about 7 GB, which put a 65B fine-tune on one 48 GB card in 2023.
19. Weights are a binary file of numbers plus a config and a tokenizer. safetensors is an 8-byte length, a JSON header of dtypes, shapes and offsets, then raw bytes, and it exists because pickle-based `.bin` files can execute arbitrary code on load; GGUF bundles quantization and tokenizer.
20. Open weights, open source, open data and licence terms are four independent things. The Open Source AI Definition 1.0 of 28 October 2024 requires code, parameters and data information sufficient to rebuild a substantially equivalent system, and by it most models marketed as open source, including Llama, do not qualify. That remains a live dispute, not a settled question.

50

CHAPTER

KB

Benchmarks, Evaluation, Versions and Dates

50.0 What this chapter gives you

1. You will be able to say exactly what a benchmark is, in five mechanical parts, and name which part is being argued about when two numbers disagree.
2. You will be able to describe the main benchmarks by name, year, size and scoring rule, and say for each one what it does not measure.
3. You will be able to explain why the same model, on the same benchmark, run by two honest teams, can legitimately produce different scores.
4. You will be able to compute pass@k by hand and explain why it is reported instead of a plain percentage.
5. You will be able to explain benchmark contamination, list five ways people try to detect it, and say honestly why detection is hard and disputed.
6. You will be able to read a model announcement and list the eight things it did not tell you.
7. You will be able to explain the difference between a training cutoff date and a release date, and why models are bad at reporting their own cutoff.
8. You will be able to explain why a product name and a model identifier are different things, and why pinning a dated identifier matters in production.
9. You will be able to build your own evaluation set of 30 to 100 examples with a scoring rule, and use it to choose and re-choose a model.
10. Throughout, you will be able to sort a claim into three bins: established fact, active research, and marketing claim.

A warning that applies to this whole chapter. The dates, item counts, paper titles and naming conventions here were checked in August 2026. The item counts and the naming conventions are stable. The leaderboard numbers are not. Any specific score in this chapter should be read as "roughly this, in mid-2026", and any score you care about should be re-checked at the source.

50.1 What a benchmark is, mechanically

50.1.1 PLAIN – in simple words

1. A benchmark is a test. That is all it is.
2. It has a fixed list of questions.
3. Somebody already knows the right answer to every question.
4. There is a rule for deciding if a given answer counts as right.
5. You give the questions to the model, collect its answers, apply the rule, and count.
6. The count, turned into a percentage, is the score.
7. There is nothing magic in it. It is a school test with a marking scheme.

8. It is not a measurement of intelligence, any more than a spelling test is.
9. It measures one thing: how many of these particular items, marked this particular way, this particular model got right on this particular day.
10. Everything difficult about benchmarks comes from the words "particular".
11. Change the wording of the question and the score moves.
12. Change the marking rule and the score moves.
13. Change which items you use and the score moves.
14. None of that is cheating. It is just what happens.
15. So when two labs publish different numbers for the same model on the same test, the first assumption should not be dishonesty. It should be setup.

50.1.2 PLAIN - a picture in your head

1. Think of a driving test.
2. The test has a fixed route, a fixed set of manoeuvres, and an examiner with a clipboard listing faults.
3. Two candidates take the same test and get different results. Fine.
4. Now take the same candidate to two different test centres.
5. One centre uses a route with a busy roundabout. The other uses a quiet estate. The candidate may pass one and fail the other.
6. One examiner counts a small kerb touch as a minor fault. Another counts it as a major. Same driving, different result.
7. One centre lets you do a warm-up lap first. The other does not.
8. Now imagine the candidate has driven that exact route four hundred times, because it is published online.
9. That last one is contamination, and section 50.5 is about it.
10. And here is the biggest problem. Passing a driving test does not tell you whether somebody can drive to Manchester in fog at night.

Where this comparison breaks: a driving test has a legal standard behind it and an examiner who watched you. Most AI benchmarks have neither. Nobody watched the model. A script compared strings. Also, driving tests are not published in advance with all the answers. Almost every AI benchmark is.

50.1.3 PLAIN - a worked example

1. Here is one real item, in the style of GSM8K, a set of grade-school maths word problems released by OpenAI in 2021.

```
Question:
Natalia sold clips to 48 friends in April, and then she
sold half as many clips in May. How many clips did she
sell altogether in April and May?

Reference answer (the dataset marks the final number
with four hash characters):
  48 / 2 = 24
  48 + 24 = 72
  #### 72
```

2. The #### marker is part of the dataset format. The number after it is the answer the scorer will compare against.
3. Now here are two different things a model might produce.

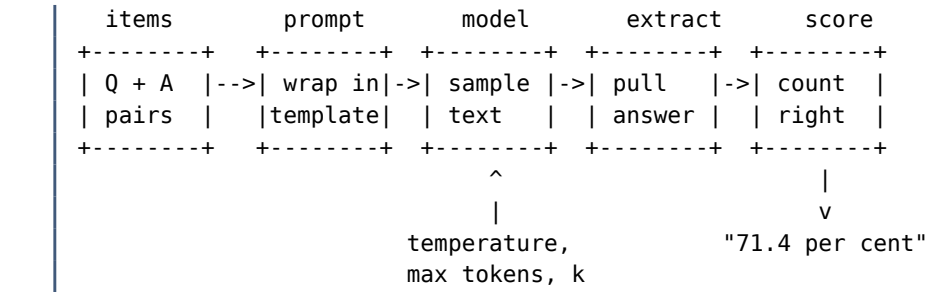
```
Model A output:
"She sold 48 in April and 24 in May, so 72 clips."
```

```
Model B output:
"72"
```

4. Both are correct. A strict scorer that demands the whole reply to equal "72" marks model A wrong and model B right.
5. A scorer that pulls the last number out of the reply marks both right.
6. Nothing about the model changed. The marking rule changed.
7. That single choice can move a reported score by several points across a whole benchmark.
8. Now a nastier case. A model replies "72 clips". A scorer that pulls the last number finds 72. Good.
9. A model replies "The answer is 48 + 24 = 72." A scorer that pulls the last number still finds 72. Good.
10. A model replies "She sold 72, up from 48." A scorer that pulls the last number finds 48 and marks it wrong. The model was right.
11. Real answer-extraction bugs look exactly like this.

50.1.4 PLAIN - what is really happening inside

1. Every benchmark run has exactly five moving parts. Learn these five and you can interrogate any published number.
2. Part one, the **items**. The fixed list of questions and known answers.
3. Part two, the **prompt template**. The wrapper of text that surrounds each item before it reaches the model. Instructions, examples, formatting.
4. Part three, the **generation settings**. How the model is asked to produce text: temperature, how many tokens it may write, whether it may think first.
5. Part four, the **answer extraction and scoring function**. The code that turns a blob of model text into "right" or "wrong".
6. Part five, the **aggregate**. How the per-item results are combined into one number, and whether that number is one run or an average of several.
7. Here is the flow as a picture.



8. Each of the five boxes is a choice made by whoever ran the evaluation.
9. Every one of those five choices changes the number that comes out.
10. Only the first box, the items, is usually published. The other four are often described in a sentence or not at all.
11. That is the single most important fact in this chapter.
12. The honest version: the sentence "this model scores 88 per cent on MMLU" is incomplete in the way "this car does 60" is incomplete. A benchmark score is not a property of a model. It is a property of a model plus a harness, measured on one day.

50.1.5 TECHNICAL - the engineer's version

1. Formally, an evaluation is a tuple: a dataset D of items, a prompt function p that maps an item to a model input, a decoding configuration, an extraction function e that maps model output to a candidate answer, a metric m that compares candidate to reference, and an aggregation over D.
2. The score is `(1/|D|) * sum over i of m(e(model(p(d_i))), a_i)` for accuracy-style metrics. Everything contested lives in p, e and the decoder.

3. Two implementation families exist for multiple-choice items and they do not give the same answer.
4. **Log-likelihood scoring**: present the question, then score each candidate continuation by the model's log probability, and pick the highest. The model never generates a token. Common for base models.
5. **Generative scoring**: present the question with lettered options, let the model emit text, then parse a letter out of it. Common for chat models.
6. On the same items and the same model these can differ by several points. Neither is wrong. They measure different things.
7. Length normalization is a further fork. Raw log probability favours short options, so harnesses often divide by token count or by character count, or normalize against the unconditional probability of the option.
8. The EleutherAI **lm-evaluation-harness** is the de facto open standard for this. It supports over 60 academic benchmarks with hundreds of subtask variants, and it powered the Hugging Face Open LLM Leaderboard. Its stated principle is that publicly available prompts make results reproducible and comparable between papers. This is a convention, not a standard.
9. Stanford CRFM's **HELM**, first published in November 2022 as "Holistic Evaluation of Language Models", takes the other approach: fix the prompts centrally, report many metrics per scenario rather than one, and publish every raw model output.
10. Terminology: a **harness** is the code that runs the loop. A **scenario** or **task** is one benchmark inside it. A **run** is one execution.
11. There is no ISO or IEEE specification for any of this. There is no conformance test. Everything in this section is convention.

Part of a run	Who usually sets it	Usually published?
Items	Benchmark author	Yes
Prompt template	Evaluator	Sometimes
Decoding settings	Evaluator	Rarely
Extraction and metric	Evaluator	Rarely

50.1.6 WORDS - remember these

1. Benchmark — a fixed test with known answers — a dataset plus a prompting, extraction and scoring protocol.
2. Item — one question — one example in the evaluation dataset, with a reference answer.
3. Prompt template — the wrapper text around a question — the function mapping a dataset item to model input, including any few-shot exemplars.
4. Harness — the program that runs the test — the evaluation framework that implements prompting, decoding, extraction and aggregation.
5. Extraction — pulling the answer out of the reply — the parser mapping raw generation to a comparable candidate answer.
6. Metric — the marking rule — the function comparing candidate to reference, such as exact match or pass@k.
7. Aggregate — the final single number — the reduction over all items, often plain accuracy, sometimes a macro average over subjects.
8. Log-likelihood scoring — asking which option the model thinks most likely — ranking candidate continuations by model log probability without generating.

50.2 The major benchmarks, one at a time

50.2.1 PLAIN – in simple words

1. There are perhaps twenty benchmark names you will see repeatedly.
2. They fall into a few families.
3. **Knowledge quizzes**: multiple-choice questions across many school and university subjects. MMLU is the famous one.
4. **Maths**: word problems and competition problems. GSM8K, MATH, AIME.
5. **Code**: write a function that passes hidden tests. HumanEval, MBPP.
6. **Hard science**: questions a PhD student would struggle with. GPQA.
7. **Agentic work**: give the model a real repository and a real bug report and see whether the tests pass afterwards. SWE-bench.
8. **Reasoning puzzles**: coloured grid puzzles with no text at all. ARC-AGI.
9. **Common sense**: pick the sensible ending to a sentence. HellaSwag.
10. **Instruction following**: did it obey the format you asked for? IFEval.
11. **Chat quality**: judged by a human or by another model. MT-Bench.
12. Each of these was built for a reason, at a time, by somebody.
13. Almost all of them are now much easier than when they were built.
14. That is not because they were badly made. It is because the models improved and because the tests leaked into training data.
15. The single most useful habit is to ask, for each benchmark: what would a model have to be able to do to score well, and is that what I need?

50.2.2 PLAIN – a picture in your head

1. Think of a hospital's set of blood tests.
2. Each test measures one narrow thing. Iron. Glucose. White cell count.
3. No single test tells you whether a person is healthy.
4. A doctor orders several, reads them together, and knows which ones are unreliable in which circumstances.
5. A benchmark suite works the same way. MMLU is a bit like a general inflammation marker: broad, cheap, and non-specific.
6. SWE-bench is like a treadmill stress test: expensive, slow, closer to the real activity, and it can be gamed by training on the treadmill.
7. And here is the part that matters. Some blood tests go out of date. A test that was diagnostic in 1990 may be useless now because everyone takes a supplement that changes the reading.
8. Benchmarks go out of date the same way, and for a similar reason: everybody now optimizes for the reading.

Where this comparison breaks: blood tests have reference ranges established by regulators and reproducible physical chemistry behind them. AI benchmarks have neither. Two labs running the same blood test on the same sample should agree within a known tolerance. Two labs running the same AI benchmark on the same model have no agreed tolerance at all.

50.2.3 PLAIN – a worked example

1. Take one benchmark and follow it end to end: GPQA.
2. GPQA stands for Graduate-level Google-Proof Question Answering.
3. It was published on 20 November 2023 by David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael and Samuel Bowman.
4. It has 448 multiple-choice questions in biology, physics and chemistry.
5. Every question was written by a person with or studying for a PhD in that subject.
6. "Google-proof" is a design goal, not a guarantee. It means the writers tried hard to make the answer un-findable by search.

7. Here is the evidence they gave for that. Skilled non-experts, allowed unrestricted web access and spending on average over 30 minutes per question, reached 34 per cent accuracy.
8. Experts in the matching field reached 65 per cent, or 74 per cent after correcting for questions the experts later agreed were flawed.
9. Chance is 25 per cent, because there are four options.
10. The best model at publication, based on GPT-4, scored 39 per cent.
11. **GPQA Diamond** is a 198-question subset. It keeps only questions where the expert validators agreed on the answer and most non-experts got it wrong. It is the hardest and cleanest slice.
12. Diamond is what almost every model announcement reports. When you see "GPQA", check whether it means the 448 or the 198.
13. What GPQA measures well: whether a model holds deep, specific technical knowledge that is not one search away.
14. What it does not measure: whether the model can do science. There is no experiment, no hypothesis, no error bar, no lab. It is a multiple-choice quiz with four options and a one-in-four floor.
15. And by 2026 the top scores on Diamond had risen far above the 65 per cent expert figure, which does not mean the models are better scientists than PhD holders. It means they are better at that quiz.

50.2.4 PLAIN - what is really happening inside

1. Here is each major benchmark in one block: what it is, when, how big, how scored, and its main blind spot.
2. **MMLU**, 2020. "Measuring Massive Multitask Language Understanding", by Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song and Jacob Steinhardt, first posted 7 September 2020.
3. 57 subjects, from elementary mathematics to professional law. Four options per question. About 14,000 test items in the widely used release, plus a 1,531-item validation split and a 285-item development split used for few-shot examples.
4. Scored as plain accuracy. Chance is 25 per cent.
5. Blind spot: it is recall of facts in multiple-choice form. It cannot tell you whether a model can write, reason for ten steps, use a tool, or admit ignorance. It also contains a known number of wrong reference answers.
6. **MMLU-Pro**, 2024. From the TIGER-Lab group, first posted 3 June 2024. About 12,000 questions, ten options instead of four, 14 subject areas, harder items, and many of the broken MMLU items removed.
7. Scores dropped by 16 to 33 points relative to MMLU. Chance falls to 10 per cent. Sensitivity to prompt wording fell from about 4 to 5 points on MMLU to about 2 points on MMLU-Pro, across 24 prompt styles they tried.
8. **GSM8K**, 2021. From "Training Verifiers to Solve Math Word Problems" by Karl Cobbe and colleagues at OpenAI, 27 October 2021. 8.5 thousand grade-school word problems: 7,473 for training, 1,319 for testing.
9. Scored by exact match on the final number. Blind spot: the arithmetic is trivial and the reasoning is two to eight steps. It has been effectively solved since about 2024 and no longer separates strong models.
10. **MATH**, 2021. Hendrycks and colleagues, 5 March 2021. 12,500 competition problems with full worked solutions, split 7,500 train and 5,000 test. Answers are mathematical expressions, not just numbers, which makes marking much harder. A 500-item subset called MATH-500 is widely reported.
11. **HumanEval**, 2021. From "Evaluating Large Language Models Trained on Code" by Mark Chen, Jerry Tworek, Heewoo Jun and 51 others at OpenAI, 7 July 2021. Exactly 164 hand-written Python problems.
12. Each has a function signature, a docstring and hidden unit tests. Scored by pass@k: the chance that at least one of k samples passes all the tests. The original Codex model scored 28.8 per cent at pass@1 and 70.2 per cent at pass@100.
13. HumanEval blind spots, and there are many. 164 items is tiny, so one item is 0.6 points. The problems are self-contained functions of a few lines. There is no project, no dependency, no existing code to read, no debugging. The tests are shallow. It is Python only. And it is thoroughly leaked.

14. **MBPP**, 2021. "Program Synthesis with Large Language Models" by Jacob Austin, Augustus Odena and colleagues at Google, 16 August 2021. 974 entry-level Python problems, each with three assert-statement tests. A hand-cleaned subset of 427 problems is often used instead.
15. **GPQA**, 2023, covered above. 448 items, Diamond subset 198.
16. **SWE-bench**, 2023. "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" by Carlos Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press and Karthik Narasimhan at Princeton, 10 October 2023.
17. 2,294 tasks built from real merged pull requests in 12 popular Python repositories. The model gets the repository at the parent commit and the issue text. It must produce a patch. The patch is applied and the project's own tests are run.
18. Scored as **resolve rate**: the percentage of tasks where the hidden tests that the real pull request made pass, do pass, and nothing that passed before breaks. This is a much stronger signal than string matching.
19. In the original paper the best model tested, Claude 2, resolved 1.96 per cent. By 2026 the reported figures on the Verified subset were far above 70 per cent. That is a genuine capability jump and also a heavily optimized target.
20. **SWE-bench Verified**, 13 August 2024, from OpenAI. 93 experienced Python developers reviewed 1,699 randomly drawn original tasks. Tasks were dropped if the issue text was underspecified or the tests were unfair. 500 were kept. OpenAI reported that 68.3 per cent of sampled original tasks were filtered out for such problems.
21. That is an important admission in both directions. The original benchmark understated model ability. And a benchmark curated by one lab is now the number everyone quotes.
22. Other official variants: SWE-bench Lite (300 tasks, cheaper), SWE-bench Multimodal (517 tasks with visual elements), SWE-bench Multilingual (300 tasks from 42 repositories in 9 languages), and a Bash-only view of the 500 Verified tasks.
23. **ARC-AGI**. Created by Francois Chollet and introduced in his 2019 paper "On the Measure of Intelligence". The tasks are small coloured grids. You see two or three input-output examples and must produce the output for a new input. No text. No knowledge required.
24. The design goal is to measure skill acquisition on unseen problems rather than stored knowledge. ARC-AGI-2, launched in 2025, has 120 public evaluation tasks and 120 held-back semi-private tasks, and every task was solved by at least two humans in under two attempts, calibrated on over 400 members of the public. ARC-AGI-3 followed.
25. The ARC Prize leaderboard shows cost per task alongside score and by default excludes systems costing more than 10,000 dollars for a run. That is unusual and honest, and more benchmarks should copy it.
26. **HellaSwag**, 2019. Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi and Yejin Choi, 19 May 2019. Pick the sensible continuation of a short description from four options. About 10,000 items in the reported split.
27. The wrong options were produced by Adversarial Filtering: generate candidates by machine, keep the ones a discriminator model finds hard. Humans scored above 95 per cent; the best models of 2019 under 48 per cent. Today it is saturated and is mostly a sanity check on base models.
28. **BIG-bench**, 2022. "Beyond the Imitation Game", posted 9 June 2022, with over 450 authors from over 132 institutions and 204 tasks. Deliberately weird, broad and long-tailed.
29. Every BIG-bench task file contains a **canary string**, a fixed random-looking identifier the authors ask people never to remove, so that anyone can test whether a model has seen the benchmark. That is a genuinely good idea, and section 50.5 explains what it can and cannot prove.
30. **BIG-Bench Hard**, 2022, is a 23-task subset where models at the time did worse than the average human rater. It is what people actually run, usually as "BBH".
31. **IFEval**, 2023. Jeffrey Zhou and colleagues at Google, 14 November 2023. About 500 prompts carrying 25 kinds of machine-checkable instruction, such as "write more than 400 words" or "mention the keyword AI at least 3 times" or "reply in JSON" or "no capital letters".
32. Its whole point is that no judgement is needed: a short Python function checks compliance. It measures

obedience to format, not quality of content.

33. **MT-Bench**, 2023. From "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena" by Lianmin Zheng, Wei-Lin Chiang, Ying Sheng and colleagues, 9 June 2023. 80 multi-turn questions in 8 categories, two turns each.
34. A strong model grades the answers on a 1 to 10 scale. The paper reported over 80 per cent agreement between the model judge and human raters, which is about the level humans agree with each other.
35. The same paper named the three biases of a model judge that you must remember: **position bias** (it prefers whichever answer came first), **verbosity bias** (it prefers the longer answer) and **self-enhancement bias** (it prefers text in its own style).
36. **AIME**. Not built for AI at all. The American Invitational Mathematics Examination is a competition run by the Mathematical Association of America: 15 questions, 3 hours, every answer an integer from 000 to 999, held twice a year.
37. That integer answer format is exactly why AI labs adopted it: marking is trivial and unambiguous. Its weaknesses are that 15 items means one question is 6.7 points, and that each year's paper is public within hours and in training data within months.
38. **Humanity's Last Exam**, launched in 2025 by the Center for AI Safety and Scale AI, published in the journal Nature on 28 January 2026. 2,500 questions across more than 100 subjects, written by close to a thousand expert contributors from over 500 institutions in 50 countries.
39. It was built precisely because MMLU had saturated. A private held-back portion is kept to detect overfitting, and a rolling variant was released on 8 October 2025 to keep adding fresh items.
40. **FrontierMath**, from Epoch AI, is a set of unpublished research-level mathematics problems in difficulty tiers, plus a separate collection of genuinely unsolved open problems. As of 31 July 2026 that open set held 50 unsolved research problems, of which AI systems had contributed solutions to three.
41. **Terminal-Bench**, a Stanford collaboration, gives an agent a real terminal and real tasks in software engineering, machine learning, security and data science, scored by whether the task actually completed. Version 1.0 had 80 tasks, version 2.0 had 89, and later versions followed.

50.2.5 TECHNICAL – the engineer's version

1. The comparison table, split into four-column pieces so it prints. All item counts verified August 2026 against the dataset cards and papers.

Benchmark	Year	Items
MMLU	2020	~14,000 test
MMLU-Pro	2024	~12,000 test
GSM8K	2021	1,319 test
MATH	2021	5,000 test
HumanEval	2021	164
MBPP	2021	974 (427 clean)
GPQA	2023	448
GPQA Diamond	2023	198
SWE-bench	2023	2,294
SWE-bench Verified	2024	500
ARC-AGI-2	2025	120 + 120 held back
HellaSwag	2019	~10,000 val
BIG-bench	2022	204 tasks
BIG-Bench Hard	2022	23 tasks
IFEval	2023	~500 prompts
MT-Bench	2023	80 questions
AIME (one paper)	yearly	15
Humanity's Last Exam	2025	2,500

2. Scoring rules and chance floors.

Benchmark	Metric	Chance floor
MMLU	Accuracy	25 per cent
MMLU-Pro	Accuracy	10 per cent
GSM8K	Exact match number	~0
MATH	Expression equivalence	~0
HumanEval	pass@k on unit tests	~0
GPQA Diamond	Accuracy	25 per cent
SWE-bench	Resolve rate	0
ARC-AGI-2	Exact grid match	~0
HellaSwag	Accuracy	25 per cent
IFEval	Constraint pass rate	varies
MT-Bench	Model-judge 1 to 10	n/a

3. The gap between what each measures and what it is claimed to measure. This column is the one people skip.

Benchmark	Claimed as	Actually measures
MMLU	General knowledge	4-way recall, one shot
GSM8K	Reasoning	Short arithmetic chains
HumanEval	Coding ability	Tiny isolated functions
GPQA Diamond	Science ability	Hard 4-way recall
SWE-bench	Autonomous engineer	Patch passes given tests
ARC-AGI	General intelligence	Grid puzzle induction
HellaSwag	Common sense	Adversarially filtered picks
MT-Bench	Chat quality	One model's taste
IFEval	Instruction following	Checkable format rules

4. Established fact: the item counts, dates, authors and scoring rules above. These are checkable in the papers and dataset cards, and they do not move.
5. Active research: whether any of these benchmarks predicts real-world task performance. The honest state of the field is that correlation exists but is weak and task-dependent, and there is no accepted transfer function from a benchmark score to a deployment outcome.
6. Marketing claim: any sentence of the form "scores X on benchmark Y, therefore it is a PhD-level scientist" or "therefore it can replace a junior engineer". The benchmark supports the first clause only.
7. Tooling. `lm-eval --model hf --model_args pretrained=... --tasks mmlu --num_fewshot 5` runs MMLU under the EleutherAI harness. The SWE-bench project ships its own Docker-based runner, because reproducing per-repository environments is most of the difficulty. `bigcode-evaluation-harness` is the common runner for HumanEval and MBPP and executes generated code in a sandbox, which you should never skip.
8. Note on the honest version of "SWE-bench measures agentic ability". The honest version: SWE-bench measures the combined ability of a model plus a scaffold plus a retrieval strategy plus a test-running loop. Two entries using the same model and different scaffolds can differ by more than 20 points. Section 50.4 returns to this.

50.2.6 WORDS - remember these

1. Saturation — everyone scores near the top so the test stops sorting — the point where score variance is dominated by noise and label errors rather than capability differences.

2. Diamond subset — the hard, clean slice of GPQA — the 198 questions where expert validators agreed and most non-experts failed.
3. Resolve rate — how often the fix actually worked — the fraction of SWE-bench instances where the fail-to-pass tests pass and pass-to-pass tests still do.
4. Adversarial filtering — keeping only wrong answers that fool machines — an iterative data-collection loop where a discriminator selects distractors.
5. Canary string — a marker planted so leaks can be spotted — a fixed unique identifier embedded in benchmark files to detect web-scrape contamination.
6. Chance floor — the score you get by guessing — the expected accuracy of a uniform random policy, 1/n for n options.
7. Held-back set — questions kept secret — a private evaluation split withheld from publication to measure overfitting to the public split.
8. Scaffold — the harness of code around a model in agent tasks — the controller implementing retrieval, tool calls, retries and stopping rules.

50.3 Actually running an evaluation

50.3.1 PLAIN - in simple words

1. Suppose you have the questions and the answers. How do you run the test?
2. First you must decide what text to actually send.
3. You could send only the question. That is called **zero-shot**.
4. You could send a few solved examples first, then the question. That is called **few-shot**, and five examples is called five-shot.
5. Few-shot examples do two jobs. They remind the model of the subject, and more importantly they show it the exact answer format you want.
6. Most of the benefit is the format, not the reminder.
7. Next you decide whether to let the model think out loud before answering. That is **chain-of-thought**.
8. On maths, letting it think out loud changes the score enormously. Not by a little. By tens of points.
9. Next you decide how random the model is allowed to be. That is the temperature setting.
10. Zero means always take the most likely next word. Higher means sometimes take a less likely one.
11. If temperature is above zero, the same model on the same question can give different answers on different days.
12. So the same evaluation run twice gives two different scores.
13. That is why careful teams run the whole thing several times and report the average, and ideally the spread.
14. Then you must decide how to mark the reply, which is the hardest part.
15. For a single number, compare numbers. For a letter, find the letter. For an essay, you need a human or another model, and both are unreliable.

50.3.2 PLAIN - a picture in your head

1. Think of a school exam being marked by three different people.
2. The first marker demands the answer in the box, in pen, and nothing else. Anything in the margin is ignored.
3. The second marker reads the whole page and looks for the answer wherever it is. More forgiving, but sometimes picks up a crossed-out number.
4. The third marker is an experienced teacher who reads the reasoning and awards a judgement out of ten.
5. The first marker is exact match. The second is regex extraction. The third is a model judge.
6. Now imagine the exam paper itself. One version says “Answer the question.” Another says “Answer the question. Show your working. Put your final answer after the words FINAL ANSWER.”

7. The second paper produces higher marks from the same pupils, because it told them the format the marker wants.
8. That is the whole of prompt template design, in one sentence.

Where this comparison breaks: a pupil who knows the answer but writes it in the margin still knows the answer, and a good teacher notices. A benchmark harness does not notice anything. It runs a regular expression. If the format is wrong the mark is zero, and the report will say the model does not know maths.

50.3.3 PLAIN - a worked example

1. Let us compute pass@k properly, because it is the metric people most often misread.
2. pass@k is not "the score when you take k tries". It is an estimate of the probability that at least one of k independently drawn samples is correct.
3. The naive way is to draw k samples and see if any pass. That is a very noisy estimate for large k.
4. The Codex paper of July 2021 uses a better estimator. Draw n samples, count c that pass, and compute:

```
pass@k = 1 - C(n - c, k) / C(n, k)

  where C(a, b) is "a choose b", the number of ways
  to pick b items from a, and n >= k.
```

5. In words: one minus the chance that all k of your picks come from the failing pile.
6. Worked example. Take one problem. Draw n = 10 samples. Suppose c = 2 pass.
7. pass@1 = 1 - C(8,1)/C(10,1) = 1 - 8/10 = 0.20.
8. pass@5 = 1 - C(8,5)/C(10,5) = 1 - 56/252 = 1 - 0.2222 = 0.7778.
9. pass@10 = 1 - C(8,10)/C(10,10). C(8,10) is 0 because you cannot pick 10 things from 8. So pass@10 = 1 - 0 = 1.00.
10. Read those three numbers. One try: 20 per cent. Five tries: 78 per cent. Ten tries: 100 per cent.
11. The model did not get better. You bought more lottery tickets.
12. This is why pass@1 and pass@100 are not comparable, and why a headline that does not say which k it used is not a result.
13. Now the temperature effect, with real published behaviour. High temperature lowers pass@1 and raises pass@100, because diversity costs you on a single draw and pays on many draws.
14. So a lab reporting pass@1 will tune temperature low, and a lab reporting pass@100 will tune it high, and both are being reasonable.

50.3.4 PLAIN - what is really happening inside

1. Take one MMLU item and watch it through the whole pipeline.
2. Zero-shot generative version. The harness builds this text:

```
The following is a multiple choice question about
high school biology.

Which organelle produces most of a cell's ATP?
A. Ribosome
B. Mitochondrion
C. Golgi apparatus
D. Lysosome
Answer:
```

3. The model generates text. Suppose it writes " B. Mitochondrion".
4. The extractor strips whitespace, takes the first letter that is A, B, C or D, and compares it to the stored key.

5. Now the log-likelihood version. The harness does not let the model write anything. Instead it asks the model, four times, "how likely is this exact continuation?"
6. It scores " A", " B", " C" and " D" as continuations, and takes the highest.
7. Same item, same model, two protocols, and sometimes two different answers.
8. A model that would have written a paragraph of hedging before saying B still assigns the highest single-token probability to " B", and scores right under the second protocol and possibly wrong under the first.
9. This is not a subtle effect. It is a common cause of a several-point gap between two published MMLU numbers.
10. Now the few-shot version. The harness prepends five solved questions from the 285-item development split before the real one.
11. The model now sees five examples of the pattern "question, options, Answer: B" and copies the pattern. Format failures collapse.
12. Now chain-of-thought. The template adds "Think step by step, then give your final answer as a single letter after the words FINAL ANSWER:".
13. The model now emits a paragraph of reasoning and then the letter. The extractor looks after "FINAL ANSWER:" instead of at the start.
14. On knowledge questions this changes little. On maths it changes everything, because the model can hold intermediate results in the text it has already written rather than trying to do the whole calculation in one forward pass.
15. That last sentence is the honest mechanism. Chain-of-thought is not the model "thinking harder". It is the model using its own output as working memory, one token at a time.

50.3.5 TECHNICAL - the engineer's version

1. Prompt sensitivity is measurable and large. The MMLU-Pro authors evaluated 24 prompt styles and found score variation of about 4 to 5 percentage points on original MMLU and about 2 points on MMLU-Pro. That is the size of the gap labs argue over in announcements.
2. Few-shot conventions by benchmark, as used in most reported numbers. These are conventions, not standards, and they drift.

Benchmark	Common shots	Common mode
MMLU	5-shot	Log-likelihood or letter
MMLU-Pro	5-shot CoT	Generative
GSM8K	8-shot CoT or 0-shot	Generative
MATH	4-shot CoT	Generative
HumanEval	0-shot	Generative, pass@k
GPQA Diamond	0-shot CoT	Generative
HellaSwag	10-shot	Log-likelihood
BBH	3-shot CoT	Generative

3. Chain-of-thought prompting comes from "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models" by Jason Wei and colleagues at Google, posted January 2022. On GSM8K with a 540-billion-parameter model it moved accuracy from roughly 18 per cent to roughly 57 per cent.
4. "Large Language Models are Zero-Shot Reasoners" by Takeshi Kojima and colleagues, May 2022, showed that the single sentence "Let's think step by step" produced much of the same effect with no examples at all.
5. Established fact: chain-of-thought raises measured accuracy on multi-step arithmetic and symbolic tasks by a large margin. Active research: whether the written chain reflects the computation that actually determined the answer. There is published evidence that models sometimes produce a correct

answer with an unfaithful chain, and that they can be steered to a wrong answer while the chain still reads plausibly.

6. Grading methods, in increasing order of flexibility and decreasing order of reliability.

Method	How it works	Main failure
Exact match	String equality	Punishes correct formats
Normalized match	Lowercase, strip, unify	Still brittle on units
Regex extraction	Pull last number or tag	Grabs the wrong number
Symbolic equivalence	Parse maths, compare	Parser gaps, timeouts
Unit tests	Execute the code	Weak or overfit tests
Model as judge	Ask a model to grade	Position, length, style bias

7. For MATH-style answers, plain string comparison is hopeless: `1/2`, `0.5` and `\frac{1}{2}` are the same answer. Serious harnesses parse into a computer algebra system and test symbolic equality, with a timeout.
8. Model-as-judge biases are documented in the MT-Bench paper of June 2023: position bias, verbosity bias, self-enhancement bias, and limited reasoning on maths. Mitigations in common use are swapping the order of the two answers and averaging, calibrating with a reference answer, and using chain-of-thought in the judge prompt.
9. Sampling and reproducibility. With temperature 0 and greedy decoding, output is nominally deterministic. In practice it often is not.
10. The honest version: floating-point reduction order on a GPU depends on how work is split across threads, that split can depend on the batch size, and batch composition on a shared serving endpoint depends on who else is sending requests. Mixture-of-experts routing can also depend on the batch. So the same prompt at temperature 0 can produce different tokens.
11. Therefore report mean and standard deviation over repeated runs. Common practice for small benchmarks such as AIME with 15 items is to run 16, 32 or 64 times and average, sometimes written "avg@32" or "mean@64".
12. Distinguish that from **majority voting**, sometimes written "cons@64" or self-consistency, where you sample 64 answers and take the most common one. That is a different, stronger system, and it costs 64 times as much.
13. Distinguish both from **best-of-n**, where you sample n answers and pick the best using a verifier or the reference answer. If the reference answer is used to pick, the number is close to meaningless as a measure of what a user would get.
14. Note the 2026 wrinkle: some providers have begun removing sampling controls entirely. Anthropic's documentation as of August 2026 lists `temperature`, `top_p` and `top_k` as deprecated for its newest models, returning an error if set to a non-default value. If you cannot set temperature, you cannot match another lab's sampling configuration.

50.3.6 WORDS - remember these

1. Zero-shot — no examples given — the item is presented with instructions only, no in-context exemplars.
2. Few-shot — a few solved examples first — k in-context exemplars prepended, usually drawn from a fixed development split.
3. Chain-of-thought — letting it show its working — eliciting intermediate reasoning tokens before the final answer, which act as external working memory.
4. Temperature — how random the wording is — the divisor applied to logits before the softmax; 0 gives greedy decoding.
5. pass@k — chance that one of k tries works — the unbiased estimator $1 - C(n-c,k)/C(n,k)$ over n samples with c correct.

6. Self-consistency — sample many, take the most common answer — majority voting over sampled chains, reported as cons@n.
7. Best-of-n — sample many, keep the best — reranking n samples with a verifier or, improperly, with the reference answer.
8. LLM-as-judge — a model marks the answers — an evaluator model scoring or ranking outputs, subject to position, verbosity and self-enhancement bias.

50.4 Why two labs report different numbers

50.4.1 PLAIN - in simple words

1. You will often see the same model, the same benchmark, and two different numbers, from two respectable sources.
2. The first thing to accept is that this is normal.
3. The second thing to accept is that it is usually not lying.
4. There are about eight ordinary causes, and you can usually work out which one applies.
5. Cause one: different wording of the prompt.
6. Cause two: a different number of worked examples given first.
7. Cause three: a different way of pulling the answer out of the reply.
8. Cause four: a different slice of the items. Someone ran 500 of the 2,294.
9. Cause five: different randomness settings, or one run against an average of many runs.
10. Cause six: one lab used a system prompt telling the model how to behave and the other did not.
11. Cause seven: one setup let the model use a calculator, a code runner or a web search, and the other did not.
12. Cause eight: a plain bug. These are common and usually found later.
13. Only after all eight are ruled out is it worth wondering about anything worse.
14. The practical response is one question: which harness did you use, and with what settings?
15. If the answer is "our internal one, details not published", the number is not comparable to anybody else's number. It may still be true.

50.4.2 PLAIN - a picture in your head

1. Imagine two labs measuring the fuel economy of the same car.
2. One drives a flat motorway at a steady 90 kilometres per hour with the air conditioning off and the tyres at maximum pressure.
3. The other drives a hilly city route in traffic with two passengers.
4. Both publish a figure in litres per hundred kilometres. The figures differ by 40 per cent. Neither lab lied.
5. This is exactly why car fuel economy eventually got a legally defined test cycle, with a written specification, run by accredited labs.
6. AI benchmarks have no such thing. There is no accredited lab, no test cycle written into law, and no penalty for using a favourable route.
7. The nearest equivalent is a shared open harness that everybody agrees to run with the same settings, and that is a voluntary convention.

Where this comparison breaks: fuel economy has a physical ground truth you can approach by measuring more carefully. There is no single true MMLU score for a model waiting to be discovered. The number genuinely depends on the protocol, so "measuring more carefully" does not converge on one value.

50.4.3 PLAIN - a worked example

1. Here is a concrete, realistic divergence, with the arithmetic.
2. Lab A reports 88.4 per cent on a knowledge benchmark. Lab B reports 84.1 per cent for the same model. The gap is 4.3 points.

3. Investigation finds four contributions.

Difference	Effect on score
5-shot vs 0-shot	+1.8 points
Letter-only vs full text	+1.1 points
Chain-of-thought allowed	+0.9 points
Best of 3 runs vs mean	+0.5 points
Total explained	+4.3 points

4. Every one of those four choices is defensible on its own.
5. Together they fully account for the gap without anybody doing anything improper.
6. Notice the last row. Reporting the best of three runs rather than the mean is the one that should make you uncomfortable, because it is a choice made after seeing the results.
7. That is the boundary between a defensible protocol difference and a presentational thumb on the scale.

50.4.4 PLAIN - what is really happening inside

1. The causes, with how you detect each one. This is the table to keep.

Cause	How to detect it
Prompt template differs	Ask for the exact template
Few-shot count differs	Look for "5-shot" in a footnote
Extraction differs	Ask what regex or parser
Subset differs	Compare item counts
Sampling differs	Ask temperature and top-p
System prompt used	Ask if one was present
Tools available	Ask about calculator, code, web
Best-of-n reporting	Ask how many runs, which shown
Plain bug	Wait for reproduction attempts

2. On tool access, be specific, because it is the largest single effect and the least often disclosed.
3. A model that may run Python and check its arithmetic will beat the same model without that on any maths benchmark, often by a wide margin.
4. A model that may search the web will beat the same model without it on any knowledge benchmark, and on a "Google-proof" benchmark it should not help much, which is itself a useful test of the benchmark's design.
5. On agentic benchmarks, the scaffold is most of the system. An entry on the SWE-bench leaderboard is a model plus a controller: how files are found, how many edit attempts are allowed, whether tests may be run and re-run, how the loop decides to stop.
6. Two teams using the identical model with different scaffolds have reported resolve rates tens of points apart. The leaderboard entry is the pair, not the model.
7. That is why SWE-bench leaderboard entries are named after systems, not models, and why a model announcement quoting a SWE-bench number should say what scaffold produced it.

50.4.5 TECHNICAL - the engineer's version

1. Ask for the harness by name and version. In practice this means one of: EleutherAI `lm-evaluation-harness` (state the version and the task name, because task definitions change between versions), Stanford `HELM`, the `bigcode-evaluation-harness` for code, the official SWE-bench runner with its Docker images, or a lab's internal harness.

2. Task definitions inside a harness are versioned and do change. The same task name in two harness versions can differ in prompt, in normalization and in which split is used. Pin the harness version in your own reports.
3. The Hugging Face Open LLM Leaderboard is the clearest example of why this matters. Its second version, launched in June 2024, replaced the original task set with IFEval, BBH, MATH Level 5, GPQA, MuSR and MMLU-Pro, run through the EleutherAI harness with fixed settings.
4. The reason given was that the original tasks had saturated and were contaminated. The consequence was that scores from before and after are not comparable at all, even for the same model.
5. That is the honest version of "the leaderboard changed": every historical number became a different measurement, and the ranking of models moved.
6. Normalization choices worth naming explicitly, because they each move multiple-choice scores by measurable amounts: raw log probability, log probability divided by token count, log probability divided by character count, and probability normalized against the model's unconditional probability of the option text.
7. A concrete reproducibility checklist to publish with any number you report:

```
model id + exact snapshot
harness name + version + task name
n-shot, and which split the shots came from
system prompt (verbatim, or "none")
temperature, top_p, max tokens, stop sequences
tools available (none / code / search / retrieval)
number of runs, and mean and standard deviation
answer extraction rule
date the run was executed
```

8. If a published number lacks four or more of those nine lines, treat it as an indication rather than a measurement.
9. Established fact: protocol differences of the sizes described here are documented and reproducible. Marketing claim: that any single reported number is "the" score for a model.

50.4.6 WORDS - remember these

1. Harness — the program that runs the benchmark — the evaluation framework; name and version both matter because task definitions change.
2. Subset — a slice of the items — a filtered evaluation split such as Lite, Verified or Diamond, not comparable to the full set.
3. System prompt — standing instructions given before the conversation — a privileged message setting persona, format and constraints, which materially changes benchmark scores.
4. Scaffold — the code around the model in an agent task — the controller loop for retrieval, tool invocation, retries and termination.
5. Tool access — whether the model may use a calculator or search — the set of external functions the model may call during evaluation.
6. Reproduction — someone else getting your number — an independent run under a published protocol, the only real check on a self-reported score.
7. Task version — which definition of the test was used — the versioned task implementation inside a harness, including prompt and normalization.

50.5 Contamination

50.5.1 PLAIN - in simple words

1. Contamination means the test questions were in the training data.
2. If the model has already seen the question and the answer, getting it right proves nothing about ability.

3. It might be memory. It might be ability. From the score alone you cannot tell which.
4. This happens almost by accident, not by plotting.
5. Benchmarks are published so that people can use them. Published means on the web. On the web means scraped. Scraped means in the training data.
6. GSM8K, MMLU, HumanEval and the rest are all on public code and dataset hosting sites, quoted in thousands of blog posts, pasted into forum questions, translated, reformatted and discussed.
7. So a model trained on a large web crawl has almost certainly seen many of them, in some form, unless someone worked hard to remove them.
8. That removal work is called decontamination, and every serious lab now does some and describes it.
9. But decontamination is filtering by text matching, and text matching misses anything reworded.
10. The effect on scores is upward, and it is uneven. It inflates old benchmarks more than new ones, and easy items more than hard ones.
11. It is very hard to prove for any particular model, because you usually cannot see the training data.
12. So the debate is conducted with indirect evidence, and reasonable people disagree about what that evidence shows.
13. The honest position: assume some contamination is present in every public benchmark number you read, of unknown size, and prefer evidence from tests the model cannot have seen.

50.5.2 PLAIN - a picture in your head

1. Imagine a school where the past twenty years of exam papers, with the official answers, are freely available online.
2. A pupil who has read all of them will do better on this year's paper if this year's paper reuses old questions.
3. Nobody stole anything. The papers were published on purpose.
4. Now the exam board tries to fix it. They remove any question that appears word for word in an old paper.
5. But a question that was reworded, or translated into French and back, or changed from apples to oranges, slips through.
6. And the pupil who read every paper does better on those too, because the structure is familiar even when the words changed.
7. Now the awkward part. The pupil who read all the old papers is also genuinely better prepared. Some of what they gained is real learning.
8. Separating the memory from the learning, from the mark alone, is impossible.
9. That is exactly the position everyone is in with model benchmarks.

Where this comparison breaks: a pupil remembers roughly what they revised and can be asked. A model cannot reliably report what was in its training data, and the people who built it often cannot enumerate it either, because the corpus is billions of documents assembled by pipelines from many sources.

50.5.3 PLAIN - a worked example

1. The clearest public experiment on this is GSM1k.
2. In May 2024 a team at Scale AI, led by Hugh Zhang, published "A Careful Examination of Large Language Model Performance on Grade School Arithmetic".
3. They commissioned a brand-new set of grade-school maths problems, written from scratch, never published, deliberately matched to GSM8K in style, number of solution steps, answer size and human solve rate.
4. Then they ran many models on both, and compared.
5. If a model's ability were real, the two scores should be about equal, since the sets are matched by construction.
6. What they found: accuracy drops of up to about 8 percentage points on the new set, with several model families showing systematic overfitting across almost all their sizes.

7. They also found a positive relationship, Spearman r squared of about 0.36, between how likely a model was to generate GSM8K examples verbatim and how large its drop was.
8. That second finding is the important one. It links the drop to memorization rather than to some accident of the new problem set.
9. And the honest counterweight, which the same paper reports: the strongest frontier models showed minimal signs of overfitting, and every model generalized to some extent to the new problems.
10. So the correct summary is not "the benchmarks are fake". It is "some models are inflated by several points, the size varies, and you cannot tell from the leaderboard which".

50.5.4 PLAIN - what is really happening inside

1. Five detection methods, what each can show, and where each fails.
2. **N-gram overlap.** Take every run of n consecutive words from every benchmark item, and search the training corpus for the same run.
3. The GPT-3 paper of 2020 used 13-gram overlap for its decontamination checks. Other projects use 8-gram, or a match on several random 50-character substrings.
4. What it shows: verbatim copying. What it misses: anything reworded, translated, reformatted, or discussed rather than quoted.
5. Also it requires access to the training corpus, which outsiders do not have.
6. **Canary strings.** The BIG-bench authors put a fixed unique identifier in every task file and asked everybody never to remove it.
7. If a model can reproduce that identifier, it saw the file. That is close to proof of exposure.
8. What it misses: everything. A model can be trained on a copy of the data with the canary stripped, on a reworded version, or on a discussion of the items. Absence of the canary proves nothing at all.
9. **Perplexity comparison.** Perplexity is a measure of how surprised the model is by a piece of text. Lower means more familiar.
10. Compare the model's perplexity on benchmark items against its perplexity on freshly written text of the same kind. If the benchmark is much less surprising, that is a signal.
11. What it misses: benchmark text is often cleaner and more formulaic than fresh text anyway, so some gap is expected. Setting the threshold is a judgement call, and this requires log-probability access that many commercial endpoints no longer expose.
12. **Old versus new items.** Build a benchmark that keeps adding fresh items after model release dates, and compare a model's score on items published before its training cutoff against items published after.
13. A large gap is strong evidence. This is the design principle behind continuously refreshed benchmarks, and behind coding benchmarks that only score problems released after a stated date.
14. What it misses: newer items may simply be harder, or drawn from a different distribution, so you need the sets to be matched carefully, which is what GSM1k did.
15. **The completion trick.** Give the model the first part of a benchmark item and see whether it continues with the rest, word for word, including the incidental details it could not guess.
16. A variant gives the model the dataset name, the split and an index, and asks it to produce that item. Another shows the item with the options in a scrambled order and checks whether the model still reproduces the original ordering.
17. What it shows: near-verbatim memory of the item. What it misses: a model that learned the answer without being able to reproduce the wording, which is the common case after instruction tuning.
18. Notice the pattern. Every method can produce a positive finding that is hard to dismiss, and no method can produce a negative finding that means anything.
19. That asymmetry is why this argument never ends.

50.5.5 TECHNICAL - the engineer's version

1. The methods and their properties, side by side.

Method	Detects	Needs
N-gram overlap	Verbatim copies	Corpus access
Canary string	Direct file exposure	Model generation
Perplexity gap	Familiarity	Log probabilities
Old vs new split	Practical inflation	A dated benchmark
Completion trick	Item memorization	Only API access
Membership inference	Probable inclusion	Token probabilities

2. Real disputes, described without alleging deliberate wrongdoing, because in none of these cases was deliberate cheating established.
3. In 2023 the paper "Rethinking Benchmark and Contamination for Language Models with Rephrased Samples" showed that a modestly sized model trained on rephrased versions of benchmark test sets could reach very high scores on those benchmarks while passing standard n-gram decontamination checks. The point was methodological: the standard filter does not work.
4. Also in 2023, a deliberately satirical paper by Rylan Schaeffer titled "Pretraining on the Test Set Is All You Need" trained a tiny model directly on benchmark test sets and reported near-perfect scores, to make the same point in the bluntest possible way.
5. Several strong small models released between 2023 and 2025 attracted public questions about whether their synthetic training data, which was generated by larger models, indirectly reproduced benchmark items. The teams involved published decontamination procedures in response. The disputes were about method and evidence, not about proven intent.
6. In late 2024 and early 2025 there was a public discussion about FrontierMath, an Epoch AI benchmark, after it emerged that OpenAI had funded its creation and had access to problems. Epoch AI published clarifications about what access existed and what hold-out arrangements were in place.
7. The general lesson from that episode is structural, not personal: if a lab funds or co-designs a benchmark it is later measured on, the benchmark needs a documented held-out portion and a stated access policy, or the number cannot carry weight regardless of anyone's good faith.
8. Which is why held-out sets now exist as standard practice: ARC-AGI keeps a semi-private evaluation set, Humanity's Last Exam keeps a private portion, and SWE-bench-style benchmarks increasingly add tasks dated after model releases.
9. Typical decontamination procedures now described in model cards and technical reports: exact-substring matching against known benchmark items, n-gram overlap filters at 8 to 13 grams, removal of documents containing canary strings, embedding-similarity screening for near-duplicates, and post-hoc measurement of the train-test gap.
10. Established fact: benchmark items appear in web crawls, and models can be induced to reproduce some of them verbatim. Established fact: matched fresh benchmarks show measurable score drops for some model families.
11. Active research: how to measure contamination without corpus access, how to correct scores for it, and how to build benchmarks that resist it by construction. There is no accepted standard method.
12. Marketing claim: "our model is decontaminated". Decontamination is a process with a coverage rate, not a state. Ask what filter, at what n-gram length, against which benchmark list, and what the measured residual was.
13. Expert disagreement, stated plainly. One camp holds that contamination is now the dominant explanation for headline benchmark gains on older tests, and that only dated, held-out or private evaluations should be believed. The other camp holds that frontier models demonstrably solve novel problems that cannot be in any corpus, so contamination inflates absolute numbers but rarely reverses rankings. Both camps agree that public numbers on pre-2023 benchmarks should be discounted.

50.5.6 WORDS - remember these

1. Contamination — the test was in the training data — overlap between evaluation items and the pre-training or post-training corpus.
2. Decontamination — trying to remove it — filtering training documents that match benchmark items by substring, n-gram or embedding similarity.
3. N-gram — a run of n consecutive words — the unit of overlap matching, with 8-gram and 13-gram both in common use.
4. Canary string — a planted marker — a unique identifier embedded in benchmark files, whose reproduction by a model evidences exposure.
5. Perplexity — how surprised the model is by text — the exponential of the mean negative log likelihood per token.
6. Held-out set — questions nobody publishes — a private split retained by the benchmark authors to measure overfitting to the public split.
7. Overfitting to a benchmark — being good at the test, not the skill — a gap between performance on the benchmark and on matched unseen items.
8. Membership inference — guessing whether an example was trained on — a statistical test using token likelihoods, such as scoring the lowest-k token probabilities of a document.

50.6 The other reasons to be sceptical

50.6.1 PLAIN - in simple words

1. Contamination is not the only problem. There are six more, and they are all ordinary and human.
2. **Saturation.** When everyone scores 92, 93 and 94 per cent, the test has stopped telling you anything.
3. At that point the remaining errors are often wrong answer keys in the benchmark itself, not model failures.
4. **Optimizing for the test.** Once a number matters, people work on the number. Some of that work improves the model. Some improves only the number.
5. **Self-reported results.** Nearly every headline benchmark figure in a model announcement was produced by the team that made the model.
6. That is not automatically wrong. It is simply unverified until someone else runs it.
7. **Cherry-picked comparisons.** Showing your new model against a competitor's older version, or against the competitor's cheap model rather than its best.
8. **Best-of-n reporting.** Letting the model try many times and reporting the best result, without saying so clearly.
9. **Different tools and scaffolding.** Your model with a code runner against their model without one, in the same bar chart.
10. **Charts that start above zero.** A bar chart whose vertical axis begins at 70 makes a two-point difference look enormous.
11. None of these require anybody to lie. Every one of them can be produced by ordinary optimism and ordinary pressure to look good.
12. Your defence is not cynicism. It is a checklist.

50.6.2 PLAIN - a picture in your head

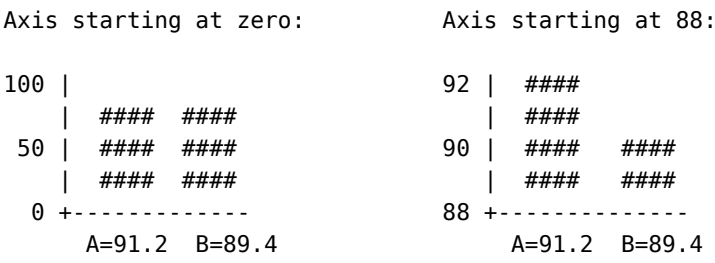
1. Think of school league tables.
2. Once schools are ranked by exam results, the results become the goal.
3. Schools start teaching to the exam. Then they enter weaker pupils for easier qualifications. Then some stop entering certain pupils at all.
4. The tables keep rising. Whether education improved is a separate question, and a much harder one to answer.
5. That is Goodhart's law happening in public, over years, with real children.

6. The measure was fine as a measure. It stopped being fine the moment it became the target.
7. And note what is not implied: the teachers were not villains. They responded to the incentive they were given.

Where this comparison breaks: exam boards are regulated, publish syllabuses, and can be audited. Nobody audits an AI lab's benchmark run. Also, schools cannot see next year's paper. Anyone training a model can see every public benchmark, in full, with answers, for free.

50.6.3 PLAIN - a worked example

1. Goodhart's law, stated properly, because it is usually misquoted.
2. Charles Goodhart was a British economist at the Bank of England. In a 1975 paper on monetary policy he wrote, in substance: any observed statistical regularity will tend to collapse once pressure is placed upon it for control purposes.
3. That is the original and it is narrower than the popular version. It is about a regularity used as a control target.
4. The popular short form is due to the anthropologist Marilyn Strathern, who in 1997 wrote: when a measure becomes a target, it ceases to be a good measure.
5. A closely related statement is Campbell's law, from the psychologist Donald Campbell in 1979: the more a quantitative indicator is used for social decision-making, the more it will be subject to corruption pressures and the more it will distort the processes it monitors.
6. Applied here: MMLU was an excellent measure in 2021, when nobody was training against it.
7. By 2024 every major lab knew the exact score they needed, the exact format, and the exact subjects where points were cheapest.
8. The measure did not become false. It became uninformative, which in practice is worse, because it still looks like a measurement.
9. Now a chart example, with numbers. Two models score 91.2 and 89.4 on some benchmark.



10. Same two numbers. On the left, two nearly identical bars. On the right, one bar looks roughly twice the other.
11. The difference is 1.8 points, which on a 14,000-item benchmark is about 250 items, and which is within the range that a prompt template change can produce on its own.

50.6.4 PLAIN - what is really happening inside

1. The red flag checklist. Each line is a question to ask of any announcement or chart.
2. Does the chart's vertical axis start at zero? If not, mentally redraw it.
3. Are the compared models named with their exact versions and dates, or just with product names?
4. Is the competitor's newest model shown, or a version from six months ago?
5. Is the competitor's flagship shown, or its cheaper tier?
6. Who ran the competitor's numbers? If the announcing lab did, were the competitor's own published numbers different?
7. Does the footnote say pass@1, or does it say something else, or nothing?
8. Does the footnote mention majority voting, consensus, best-of-n, or a number of samples greater than one?
9. Did both models have the same tools? Search, code execution, retrieval, file access, a scaffold?

10. Was there a system prompt, and was it the same for both?
11. Is the benchmark one where the top scores are already above 90 per cent?
12. Is the improvement larger than the known prompt-sensitivity range for that benchmark, which is roughly 2 to 5 points on multiple-choice tests?
13. Is the benchmark new, and was it built or funded by the announcing lab?
14. Has any independent party reproduced the number, and how long ago?
15. If you cannot answer eight of these fourteen from the announcement itself, the announcement is a claim, not a result.

50.6.5 TECHNICAL - the engineer's version

1. Saturation has a precise consequence: as mean accuracy approaches the label ceiling, the between-model variance shrinks toward the label-noise variance, and rank ordering becomes unstable across runs.
2. MMLU is known to contain a non-trivial number of incorrect or ambiguous reference answers. Independent audits published between 2023 and 2024 put the error rate in some subsets in the range of several per cent, with virology and a few professional subjects worst. Estimates vary by auditor and by criteria, so treat the exact figure as disputed.
3. The practical implication: a reported 92 per cent on MMLU may be close to the maximum achievable, and a difference between 92 and 93 may be noise plus disagreement about wrong keys.
4. This is exactly why MMLU-Pro, GPQA Diamond, Humanity's Last Exam and FrontierMath were built. New benchmarks appear when old ones saturate, and the appearance of a new hard benchmark is itself evidence that the previous one stopped working.
5. A documented case of the "which version did you test" problem. On 5 April 2025 Meta released Llama 4 in two configurations, Scout and Maverick. A variant identified as `Llama-4-Maverick-03-26-Experimental`, described as optimized for conversationality, scored very highly on the LMArena leaderboard. It was not the model released to the public.
6. On 8 April 2025 the arena operators stated that Meta's interpretation of their policy did not match what they expect from providers, that Meta should have made it clearer that the entry was a customized model, and that they would update their policies. Meta denied training on test sets.
7. Report that as what it is: a disclosure and naming failure with a policy response, not an established case of fraud. The transferable lesson is that a leaderboard entry identifies a specific artifact, and the artifact you can download may not be the artifact that was measured.
8. Statistical hygiene almost nobody applies. For a benchmark of N items and an observed accuracy p, the standard error is roughly sqrt(p(1-p)/N).

Benchmark	N	Approx 1 SE at p=0.9
HumanEval	164	2.3 points
GPQA Diamond	198	2.1 points
AIME (one paper)	15	7.7 points
SWE-bench Verified	500	1.3 points
MMLU	14,000	0.25 points

9. Read that table carefully. On GPQA Diamond, a 2-point difference is about one standard error, and one item is 0.5 points. On a single AIME paper, getting one more question right is 6.7 points.
10. So a headline of the form "scores 93.3 on AIME versus 86.7 for the competitor" describes one question out of fifteen.
11. That is before considering run-to-run sampling variance, which for reasoning models on small maths sets is frequently larger than the between-model gap.
12. Established fact: the standard-error arithmetic above. Marketing claim: any ranking asserted from a gap smaller than one standard error.

50.6.6 WORDS - remember these

1. Goodhart's law — a measure used as a target stops measuring — an observed regularity collapses when pressure is applied to it for control purposes.
2. Campbell's law — indicators used for decisions get corrupted — quantitative social indicators distort the processes they are meant to monitor.
3. Saturation — the test is too easy now — mean score near the ceiling, so between-model variance falls below label-noise variance.
4. Label noise — wrong answers in the answer key — the fraction of benchmark items whose reference answer is incorrect or ambiguous.
5. Self-reported — the maker measured it — a result produced by the model provider under an unpublished or partially published protocol.
6. Truncated axis — a chart that hides how small a gap is — a bar chart whose value axis does not begin at zero.
7. Standard error — how much the score would wobble by chance — approximately sqrt(p(1-p)/N) for accuracy p over N independent items.
8. Cherry-picking — showing the favourable comparison — selective choice of baselines, versions, subsets or runs after seeing results.

50.7 The alternatives, and their own weaknesses

50.7.1 PLAIN - in simple words

1. If fixed quizzes are weak, what else is there? Five things, each with its own flaw.
2. **Preference arenas.** Real people type a question, get two anonymous answers, and vote for the better one. Ratings are computed from the votes.
3. **Private held-out tests.** The same idea as a benchmark, but the questions are never published, so they cannot leak.
4. **Domain evaluations.** A test built for one job: medical coding, legal citation, chip design, customer support in Hindi.
5. **Agentic and end-to-end tests.** Give the model a real task in a real environment and check whether the task got done.
6. **Red-teaming.** People deliberately attack the model to find failures that no fixed test would catch.
7. Every one of these is better than MMLU for something, and worse for something else.
8. The arena measures whether people like the answer, which is not the same as whether the answer is correct.
9. A friendlier, longer, better-formatted wrong answer beats a curt correct one, quite often.
10. Private tests cannot be independently checked, because they are private.
11. Domain tests are only as good as the person who wrote them, and they do not transfer to other domains.
12. Agentic tests are expensive, slow, and measure the scaffold as much as the model.
13. Red-teaming finds problems but cannot tell you that no problems remain.
14. There is no method without a weakness. The working answer is to use several and to know what each one is blind to.

50.7.2 PLAIN - a picture in your head

1. Think about how you would judge a restaurant.
2. A hygiene inspection is a fixed checklist with a pass mark. Objective, narrow, and it says nothing about whether the food is good.
3. Customer star ratings are a preference arena. They capture something real that no inspection captures, and they reward large portions, friendly staff and good lighting as much as cooking.
4. A critic's blind visit is a private held-out evaluation. Harder to game, impossible for you to verify.

5. Asking a chef to cook one specific dish you care about is a domain evaluation. Most informative for you, useless for anyone who wants something else.
6. Watching the kitchen run a full Saturday service is the agentic test. Closest to reality, and by far the most expensive to arrange.

Where this comparison breaks: restaurant customers eat the food, so their ratings are grounded in the actual product. Arena voters usually cannot verify whether the answer they preferred was true. They are rating the presentation of a claim, often on a topic where they are not the expert.

50.7.3 PLAIN – a worked example

1. How a preference arena turns votes into a number.
2. Each vote is a pairwise comparison: model A beat model B on this prompt.
3. The rating system is the one from chess, invented by Arpad Elo, a Hungarian-American physicist, and adopted by the United States Chess Federation in 1960 and by FIDE in 1970.
4. The core formula gives the expected score of A against B from the gap in their ratings:

```
E(A) = 1 / (1 + 10 ^ ((R_B - R_A) / 400))

R = rating. A 400-point gap means the stronger side
is expected to win about 10 times out of 11.
```

5. Work three cases. A gap of 10 points gives an expected win rate of 51.4 per cent. A gap of 100 points gives 64.0 per cent. A gap of 200 points gives 76.0 per cent.
6. Read the first one again. A 10-point difference on a leaderboard means the higher model wins about 51 times in 100. That is close to a coin flip.
7. Published confidence intervals on such leaderboards are frequently plus or minus 3 to 10 points, so models separated by fewer than about 15 points are usually statistically tied even though they occupy different rank numbers.
8. As of August 2026, the largest public text arena listed a few hundred ranked models, with top scores clustered within a few points of each other and stated uncertainties of roughly plus or minus 3 to 10 points.
9. A ranking that shows positions 1 through 8 spanning about 15 points is, in statistical terms, showing one group of models, not eight ranks.
10. The honest version: an arena rating is not a quality score. It is a fitted parameter that predicts how often this model's answer is preferred over another model's answer, by these voters, on these prompts, in this period. Every clause in that sentence limits what the number can be used for.

50.7.4 PLAIN – what is really happening inside

1. What an arena measures well: how a broad population of users, asking the kinds of question they actually ask, reacts to an answer.
2. That is genuinely valuable and no fixed benchmark captures it. It picks up tone, refusal behaviour, formatting, and the ability to guess what the user meant.
3. What it measures badly, and the biases are documented, not theoretical.
4. **Length bias.** Longer answers win more often, controlling for content. Arena operators now publish a length-controlled ranking alongside the raw one specifically because of this.
5. **Style bias.** Headings, bullet lists and bold text raise win rates. The same content flattened into prose wins less.
6. **Verification gap.** A voter comparing two answers about a legal deadline usually does not know the deadline. They vote on confidence and clarity.
7. **Prompt distribution.** The votes come from whoever uses the site. That population skews technical and English-speaking and asks a lot of short questions, so the rating reflects that mixture and not yours.

8. **Provider self-selection.** Providers choose when to enter a model and which variant, and can test variants privately before public entry.
9. Red-teaming, briefly, because it is a different shape of thing. It is not scored on a scale. Its output is a list of found failures, and its value is in what it finds, not in a number.
10. The structural weakness of red-teaming is that it can only ever demonstrate presence, never absence. Not finding a jailbreak is not evidence that none exists.

50.7.5 TECHNICAL - the engineer's version

1. Modern arenas do not run a live Elo update loop, because Elo is order-dependent: the same set of games in a different order gives different ratings. They fit a **Bradley-Terry** model by maximum likelihood over the whole vote history, then bootstrap for confidence intervals, then present the result on an Elo-like scale.
2. That is why you should read the interval, not the rank. Ranks on such boards are usually published as a range, precisely because ties are common.
3. Style-controlled ranking works by fitting the preference model with additional covariates for response length and markdown features, so that the reported strength coefficient is adjusted for those factors.
4. Naming instability is real here too. The board widely known as LMSYS Chatbot Arena became LMArena; as of August 2026 the domain `lmarena.ai` redirects to `arena.ai`. Cite what a leaderboard is, not just where it lives.
5. Evaluation methods against what they are good and bad at.

Method	Good at	Bad at
Fixed benchmark	Cheap, comparable	Leaks, saturates
Preference arena	Real user taste	Truth, length bias
Private held-out	Leak resistance	Nobody can check
Domain eval	Your actual job	Generalizing
Agentic end-to-end	Realism	Cost, scaffold effects
Red-team	Finding failures	Proving absence
Model-as-judge	Scale, speed	Style and self bias

6. Cost figures matter and are rarely mentioned. A full SWE-bench Verified run with an agentic scaffold means 500 tasks, each with many model calls and repository builds. Runs costing hundreds to thousands of dollars and taking many hours are normal, which is exactly why the 300-item Lite subset exists.
7. This is also why the ARC Prize leaderboard reports cost per task and by default hides systems whose run cost exceeded 10,000 dollars. Any score can be bought with enough sampling; a score with a cost attached cannot.
8. Established fact: length and position biases in preference judging, and the order dependence of online Elo. Active research: how to control for style without controlling away real quality, and how to weight votes by voter expertise. Marketing claim: "ranked number one" when the interval overlaps five other entries.

50.7.6 WORDS - remember these

1. Preference arena — people vote between two anonymous answers — a pairwise human preference collection platform with a fitted rating model.
2. Elo — a chess rating scale reused for models — a rating system where the expected score is a logistic function of the rating difference over 400.
3. Bradley-Terry — the proper way to fit ratings from all votes at once — a maximum-likelihood pairwise comparison model, order independent.

4. Confidence interval — how much the rating could be wrong — the bootstrapped range around a fitted rating; overlapping intervals mean a tie.
5. Length bias — longer answers win — a systematic preference for verbose responses independent of content quality.
6. Style control — adjusting for formatting and length — fitting the preference model with covariates for length and markdown structure.
7. Red-teaming — attacking your own model on purpose — adversarial testing to discover failures; evidences presence, never absence.
8. Held-out evaluation — a private test set — evaluation items never published, used to detect overfitting to public benchmarks.

50.8 What "eval" means inside a company

50.8.1 PLAIN - in simple words

1. In public, "eval" means a famous benchmark with a leaderboard.
2. Inside a company that builds a product on a model, it means something much smaller and much more useful.
3. It means a file of examples of the job the product actually does, with the answer you want for each.
4. Thirty to a hundred examples is normal. Not fourteen thousand.
5. They are hand-picked, mostly from real cases that went wrong.
6. They are private, because they contain real customer text and because they are the company's actual measuring stick.
7. Every time anyone changes the prompt, the model, the retrieval setup or a parameter, the file is run again.
8. If the score drops, the change is rejected. That is the whole discipline.
9. In other words, it is a regression test suite, exactly like the automated tests you would write for ordinary software.
10. The score is not comparable to anyone else's score, and does not need to be.
11. It only needs to answer one question: did this change make my product better or worse at my job?
12. Nobody can build this for you, because only you know what your job is.
13. If you build software on a model and you do not have one of these, you are guessing every time you change anything.

50.8.2 PLAIN - a picture in your head

1. A bakery buys flour. The supplier changes.
2. The baker does not read the flour industry's national quality rankings.
3. The baker bakes their own three standard loaves with the new flour and compares them to the old ones.
4. Same recipe, same oven, same timings. Only the flour changed.
5. If the loaves are worse, the flour is rejected, whatever the rankings said.
6. The bakery's test is small, private, unscientific and completely decisive, because it measures the thing the bakery sells.
7. That is an internal eval.

Where this comparison breaks: bread quality is judged by a person who has baked for twenty years, in one taste. Model output quality often needs a written rule, because you will run the test hundreds of times and cannot taste every loaf yourself. Writing that rule down is the hard part of the job.

50.8.3 PLAIN - a worked example

1. A practical recipe for building one. Follow it in order.
2. Step 1. Write down the job in one sentence. "Turn a customer email into a support ticket with a category, a priority and a two-line summary."

3. Step 2. Collect 30 to 100 real inputs. Not invented ones. Real.
4. Step 3. Make sure the mix is deliberate, not random.

Slice of the set	Share	Why
Ordinary typical cases	50 per cent	The common path
Known hard cases	25 per cent	Where it broke before
Edge and malformed input	15 per cent	Robustness
Should refuse or escalate	10 per cent	Safety and limits

5. Step 4. For each input, write the answer you want. Where several answers are acceptable, write the rule instead of a single answer.
6. Step 5. Choose a scoring rule per item, from the cheapest kind that works.

```
exact       category must equal "billing"
contains    summary must mention the order number
regex       priority must match ^(P1|P2|P3)$
schema      output must parse as valid JSON
assert      code must pass these three tests
judge       a model grades 1-5 against a rubric
human       a person grades, for a 10-item subset
```

7. Step 6. Store it as a plain file. One row per example. Input, expected, rule, and a note about why it is in the set.
8. Step 7. Write a script that runs all items and prints one number plus the list of failures. The list of failures is the part you will actually read.
9. Step 8. Run it three times and note the spread before you trust any change smaller than that spread.
10. Step 9. Keep it in version control next to the code, and add a new row every time a real failure reaches a user. The set grows by itself.

50.8.4 PLAIN - what is really happening inside

1. Why 30 to 100 and not 10 or 10,000.
2. Below about 30, one item is more than 3 per cent, and the whole set can be passed by luck or by a prompt change that happens to suit those items.
3. Above a few hundred, hand-writing expected answers becomes a project, and people stop maintaining it. An unmaintained eval set is worse than none, because it produces confident wrong signals.
4. The trade-off is honest: 30 to 100 items cannot tell you a 2 per cent difference. It can reliably tell you a 15 per cent difference, which is the size of change that usually matters when swapping models or prompts.
5. Where model-as-judge belongs. Use it when the output is free text and no mechanical rule works, but pin it down: a fixed rubric with numbered criteria, a fixed judge model with a fixed dated identifier, and a fixed temperature.
6. And validate the judge once: have a person grade 20 items, compare with the judge, and record the agreement rate. If the judge agrees with your human less than about 80 per cent of the time, fix the rubric before trusting it.
7. Also record cost and latency per item on every run. A change that improves quality by 3 per cent and triples cost is a decision, not an improvement, and you cannot make that decision without both numbers.

50.8.5 TECHNICAL - the engineer's version

1. Treat the eval set as a first-class artifact: versioned in git, with a schema, a fixed identifier per item, and a changelog. Never edit an item silently; add a new one and deprecate the old one, or every historical number becomes uninterpretable.

2. A workable row schema:

```
id            stable, never reused
input         verbatim, including whitespace
expected      answer or acceptance rule
rule          exact | contains | regex | schema
              | assert | judge | human
slice         typical | hard | edge | refusal
added_on      date
reason        one line, why this exists
```

3. Report three numbers, not one: pass rate, mean latency, mean cost per item. Report pass rate per slice as well as overall, because a model that fixes typical cases and breaks refusals is a regression, not a gain.
4. Guard against a subtle trap: if you iterate on prompts against the eval set many times, you overfit to it exactly as labs overfit to public benchmarks. Hold back 20 per cent of items as a set you run only at decision time.
5. Available tooling as of 2026 includes provider-hosted eval products, open frameworks such as OpenAI's `evals` repository, `promptfoo`, `deepeval`, `Inspect` from the UK government's AI institute (named the AI Safety Institute at launch in 2024 and renamed the AI Security Institute in February 2025), and `LangSmith`-style tracing platforms. Note that provider-hosted eval products get deprecated too: as of mid-2026 OpenAI had announced shutdown dates for several platform features. Prefer a format you own.
6. Statistical note. With 50 items and an observed pass rate of 0.80, one standard error is sqrt(0.8 * 0.2 / 50) = 0.0566, about 5.7 points. So a move from 80 to 84 per cent on 50 items is well inside noise. Do not ship on it.
7. Established fact: internal regression evals are standard practice at companies shipping model-backed products. Marketing claim: any vendor assertion that their public benchmark results predict your task performance.

50.8.6 WORDS - remember these

1. Internal eval — your own small private test — a curated regression suite of task-representative items with per-item acceptance rules.
2. Regression test — checking a change did not break things — re-running a fixed suite after every modification and comparing to the last result.
3. Slice — a named group of items — a labelled subset such as typical, hard, edge or refusal, reported separately.
4. Rubric — the written marking scheme for a judge — the numbered criteria a judge model or human applies to score free text.
5. Golden set — the agreed correct answers — the reference outputs, versioned and never silently edited.
6. Holdout — items you do not tune against — the reserved fraction run only at decision time to detect overfitting to your own set.
7. Acceptance rule — how an item is marked — the per-item comparison, from exact match through to human judgement.

50.9 Reading a model announcement critically

50.9.1 PLAIN - in simple words

1. A model announcement is a marketing document that contains real data.
2. Both halves of that sentence are true and you need both.
3. The data is usually accurate as far as it goes. The selection is not neutral.
4. So read it the way you read a job application: assume everything stated is true, and pay most attention to what is missing.

5. Nine questions do almost all the work, and section 50.9.3 lists them.
6. If the announcement answers all nine, that is a good sign about the team, independent of the scores.

50.9.2 PLAIN - a picture in your head

1. Think of a used-car advertisement listing the good features honestly.
2. New tyres, full service history, one owner. All true.
3. It does not mention the mileage, and you notice that it does not.
4. You do not conclude the car is bad. You conclude you must ask.
5. The absence of a number is itself information, because the seller chose the list.

Where this comparison breaks: a car has one mileage figure and the seller knows it. A model has no single benchmark score to withhold, because the score depends on the protocol. So the missing item is usually the protocol, not a number, and the protocol is much easier to omit without anyone noticing.

50.9.3 PLAIN - a worked example

1. The nine questions, in order, applied to any announcement.
2. Which benchmarks are shown, and which of the usual ones are absent? An absent standard benchmark is the loudest signal in the document.
3. Which comparison models, at which exact versions and dates? "GPT-4" is not a version. "Claude" is not a version.
4. Who produced the competitor numbers: this lab, or the competitor's own published figures, or an independent runner?
5. What was the prompting setup: shots, chain-of-thought, system prompt?
6. Did the model have tools? Search, code execution, retrieval, file access? And did the comparison models have the same ones?
7. Is the score pass@1 and a single run, or is it best-of-n, majority voting, or an average over many samples?
8. Does the chart's axis start at zero, and is the gap bigger than the benchmark's known prompt sensitivity?
9. Is the benchmark new, and who built or funded it?
10. Has an independent third party reproduced any of it, and when?

50.9.4 PLAIN - what is really happening inside

1. Two phrasings that should always slow you down.
2. "State of the art on X" with no date. State of the art is a claim about a moment, and the moment is usually the week before publication.
3. "Comparable to human experts". Ask which humans, doing what, under what time limit, and with what resources. GPQA is the model case: its expert baseline was 65 per cent, measured under stated conditions, and quoting a model above that as "beyond PhD level" ignores that the humans were answering a four-option quiz outside a lab.
4. Also watch for the shift between a **model** claim and a **system** claim. A product with retrieval, tools and a scaffold is a system. Its score belongs to the system.
5. And watch the switch of subset: a headline referencing SWE-bench and a footnote saying Verified, or GPQA in the headline and Diamond in the footnote, are not errors, but they change what is being compared.

50.9.5 TECHNICAL - the engineer's version

1. Where the answers usually live, in decreasing order of reliability: the system card or technical report appendix, the model card, the announcement footnotes, the blog post body, the chart itself.
2. Reading order should therefore be reversed from the page order: appendix first, chart last.
3. Independent reproduction sources worth checking before believing a headline: the benchmark's own leaderboard, third-party evaluation organizations that run their own harnesses and publish protocols, and the open harness community's issue trackers, where discrepancies get argued out publicly.

4. A note on speed. Reproductions typically appear days to weeks after a release. The gap between announcement and reproduction is precisely the window in which the announcement is unchecked.
5. Established fact: benchmark numbers in major technical reports are generally reproducible within a few points when the protocol is published. Marketing claim: leaderboard positions asserted from gaps inside the reproduction spread.

50.9.6 WORDS - remember these

1. Model card — a short standard description of a model — a document stating intended use, training data summary, evaluations and limitations.
2. System card — the same for a whole product — documentation covering the model plus scaffolding, tools, safety mitigations and evaluations.
3. Self-reported — measured by the maker — a result produced by the provider, pending independent reproduction.
4. Reproduction — someone else got the same number — an independent run under a published protocol.
5. State of the art — best known result so far — a time-indexed claim, valid only at a stated date on a stated protocol.
6. System versus model — the product versus the weights — the distinction between an end-to-end deployed pipeline and the underlying network.

50.10 Training cutoff versus release date

50.10.1 PLAIN - in simple words

1. There are two dates and people constantly confuse them.
2. The **training data cutoff** is the last date of text the model learned from.
3. The **release date** is the day you could first use it.
4. The gap between them is typically several months, and has often been a year or more.
5. During that gap the world kept happening and the model did not see any of it.
6. So a model released in one year can be confidently describing the world of a year or more earlier.
7. It does not know it is out of date. It has no clock and no calendar.
8. Unless you tell it today's date, it does not know today's date.
9. And it is often wrong about its own cutoff, which surprises people most.

50.10.2 PLAIN - a picture in your head

1. Imagine someone who read every newspaper up to a certain Tuesday, then went into a sealed room, and is now answering your questions.
2. They are extremely well read up to that Tuesday.
3. They do not know today is a different day. Nobody told them.
4. Ask what happened last week and they will describe the last week they remember, with total confidence.
5. Worse, they read fewer papers in the final month than in earlier months, because the papers about that month had not been written yet.
6. So their sense of "recent" lands earlier than the actual Tuesday.

Where this comparison breaks: a person in a sealed room knows they are in a sealed room, and will say so. A model has no representation of its own situation, so it cannot volunteer the warning. That is the whole practical danger of the gap.

50.10.3 PLAIN - a worked example

1. Real published gaps between stated cutoff and release.

Model	Stated cutoff	Released
GPT-4	September 2021	14 March 2023
Claude 3 family	August 2023	4 March 2024
Llama 3 8B	March 2023	18 April 2024
Llama 3 70B	December 2023	18 April 2024
GPT-4o	October 2023	13 May 2024

2. Read the first row. GPT-4 shipped about 18 months after its data ended.
3. Read rows three and four. Two models released on the same day, from the same family, with cutoffs nine months apart.
4. That alone should end any belief that a family name implies a shared cutoff.
5. Since about 2024 the gaps have narrowed, often to a few months, because pipelines got faster and freshness became a selling point. But the gap has never been zero and cannot be.

50.10.4 PLAIN - what is really happening inside

1. Why the gap exists at all, in order.

```
data collection and filtering      weeks to months
pretraining run                    weeks to months
post-training: SFT, RL, tuning     weeks to months
internal + external evaluation     weeks
safety testing and red-teaming     weeks to months
staged rollout and capacity build  weeks
```

2. Every stage needs the previous one finished, so the delays add rather than overlap.
3. Now why models are unreliable about their own cutoff. Three separate causes.
4. First, the cutoff is not stored anywhere in the weights. If the model states one, it is because someone wrote it into the post-training data or the system prompt, or because the model is guessing.
5. Second, the internet writes about events with a lag. An event in the final two months before cutoff has far fewer documents about it than an event two years earlier. So the model's felt sense of "the present" sits earlier than its real cutoff. Researchers call this the effective cutoff, and it is commonly earlier than the stated one.
6. Third, training corpora contain many documents about earlier models with earlier cutoffs. A model asked its own cutoff can echo a date it read.
7. The practical rule that follows: never ask a model for its cutoff and act on the answer. Look it up in the provider's documentation.
8. And always put the current date in the system prompt if the date matters.

50.10.5 TECHNICAL - the engineer's version

1. Terminology precision. "Knowledge cutoff" usually refers to the pretraining corpus cutoff. Post-training data is frequently newer, and retrieval or tool results are newer still. A deployed system can therefore have three different effective dates at once.
2. The paper "Dated Data: Tracing Knowledge Cutoffs in Large Language Models", published in 2024, documented that measured effective cutoffs for many open models are earlier than the reported ones, and attributed this partly to corpus construction: deduplication and re-crawling mean older versions of pages persist while newer content is thinner.
3. Mitigations, in the order you should reach for them:

Mitigation	What it fixes
Date in system prompt	Model thinks it is the past

Mitigation	What it fixes
Web search tool	Missing recent facts
Retrieval over your docs	Missing private facts
Explicit "you may not know"	Confident staleness

4. Established fact: the cutoff-to-release gap, and its documented sizes. Established fact: models frequently misreport their own cutoff. Active research: measuring effective cutoffs per topic. Marketing claim: "always up to date", which describes a retrieval pipeline, not a model.
5. Practical detection test you can run in one minute: ask the model about three events with known dates spread across the suspected boundary, and see where its knowledge stops. That measures the effective cutoff for that topic, which is what you actually care about.

50.10.6 WORDS - remember these

1. Training cutoff — the last date of learned text — the corpus collection boundary for pretraining data.
2. Effective cutoff — where the knowledge really stops — the empirically measured boundary, usually earlier than the stated cutoff.
3. Release date — when you could first use it — general availability date of a specific model version.
4. Staleness — the model living in the past — degradation of factual accuracy for events after the cutoff.
5. Retrieval augmentation — giving it fresh documents at question time — supplying externally fetched context so the answer is not limited to weights.
6. Post-training — the tuning after the big training run — supervised fine-tuning and reinforcement learning stages, often using newer data than pretraining.

50.11 Versions, snapshots and the moving-name problem

50.11.1 PLAIN - in simple words

1. A product name and a model identifier are two different things.
2. The product name is what the marketing says. It is a brand.
3. The model identifier is the exact string your program sends to the API.
4. A product name can point at different weights over time, without notice.
5. A dated identifier points at one fixed set of weights that never changes.
6. So the same product name can behave differently next month, and your carefully tuned prompt can quietly stop working.
7. Nobody broke anything on purpose. The name moved.
8. The rule that follows is short: in production, use a dated identifier, and re-run your own eval set whenever you change it.

50.11.2 PLAIN - a picture in your head

1. Think of a coffee shop's "house blend".
2. It is on the menu every year under the same name.
3. The beans change with the harvest, the roast is adjusted, the supplier changes. The name does not.
4. If you love it, you cannot order last year's. It no longer exists.
5. A dated identifier is like buying a specific labelled lot, roasted on a stated day, from a stated farm.
6. You can reorder exactly that, until the shop stops stocking it.

Where this comparison breaks: coffee changes gradually and you taste it immediately. A moving model name can change one narrow behaviour, such as how it formats JSON or when it refuses, while everything else stays identical. You will not notice by tasting. You notice because a downstream parser starts failing at two in the morning.

50.11.3 PLAIN - a worked example

1. Real naming conventions from the three largest providers, as observed in August 2026. These change, so treat the shapes as the lesson and the examples as dated illustrations.

Provider	Shape	Example
OpenAI	name-YYYY-MM-DD	gpt-5-2025-08-07
Anthropic	family-version-YYYYMMDD	claude-opus-4-1-20250805
Anthropic	undated alias	claude-opus-5
Google	name plus channel	gemini-3.6-flash
Google	dated preview	model-preview-MM-YYYY

2. Read the OpenAI row. `gpt-5-2025-08-07` is a snapshot: those exact weights, fixed. `gpt-5` on its own is an alias that may be repointed.
3. Read the Anthropic rows. Some identifiers carry an eight-digit date, some do not. The dated ones are the ones to pin.
4. Google's documentation as of August 2026 describes four channels: stable, preview, latest and experimental. The `latest` alias is explicitly defined as pointing at the newest release for a model variation, with a stated notice period of about two weeks before a breaking change.
5. That is the clearest possible statement of the trade-off. An alias gives you automatic upgrades and about two weeks of warning. A pinned snapshot gives you stability and the obligation to migrate yourself.
6. Note also the older Google style using numeric suffixes such as `-001` and `-002`, where the trailing number is the revision of the same model generation. Different shape, same purpose.

50.11.4 PLAIN - what is really happening inside

1. Why aliases exist at all, honestly stated from both sides.
2. From the provider's side: most users want improvements without doing work, safety fixes must be able to ship immediately, and serving many pinned versions costs real hardware.
3. From your side: any change to weights is a change to behaviour, and you have tested against the old behaviour.
4. Those two positions cannot both be fully satisfied, which is why both options exist.
5. What actually changes when a name is repointed: refusal boundaries, verbosity and formatting habits, tool-calling and JSON reliability, tokenizer or context limits in larger changes, and latency and price.
6. What almost never changes silently: the API shape. Providers version that separately, precisely because it would break everyone at once.
7. The distinction is worth naming. The API contract is close to a standard, documented and versioned. Model behaviour under an alias is a convention, and one that assumes you will cope.
8. There is also a middle case people forget: a provider may change the serving stack, quantization or routing behind a fixed identifier. The weights are the same; the outputs may not be bit-identical. Ask about this if determinism matters to you.

50.11.5 TECHNICAL - the engineer's version

1. The production rule, in three lines.

```
1. Pin the dated snapshot in config, never an alias.
2. Store the exact id with every logged output.
3. Re-run your eval set before changing the pin.
```

2. Logging the identifier with each output is the step people skip and later regret. Without it, you cannot answer "did quality drop when we moved?" because you cannot tell which requests used which weights.

3. Keep the identifier in configuration, not in code, so a migration is a config change plus an eval run, not a code release.
4. Where aliases are still right: prototypes, internal tools, and anything where a two-week notice period is acceptable and nobody has built a parser around the exact output shape.
5. Where snapshots are essential: anything with a downstream schema, anything regulated, anything with a tuned prompt, and anything you cannot re-test quickly.
6. Watch for parameter-level changes too, not just weight changes. As one 2026 example, Anthropic's documentation lists `temperature`, `top_p` and `top_k` as deprecated for its newest models, with non-default values returning an error. Code that hard-codes a temperature will fail on migration even though the API version did not change.
7. Established fact: aliases are repointed and dated snapshots exist at all three major providers. Marketing claim: "drop-in upgrade". Every upgrade is a behaviour change until your own eval set says otherwise.

50.11.6 WORDS - remember these

1. Product name — the brand on the website — a marketing label that may map to different artifacts over time.
2. Model identifier — the exact string you send — the API-level model name that selects a served artifact.
3. Snapshot — a frozen dated version — an immutable identifier bound to one set of weights and serving configuration.
4. Alias — a name that follows the newest release — a mutable pointer, such as a bare family name or a `latest` channel.
5. Pinning — fixing the version in config — specifying an immutable snapshot identifier so behaviour does not change under you.
6. Channel — the stability tier — stable, preview, latest or experimental, each with different change and notice policies.

50.12 Deprecation and retirement

50.12.1 PLAIN - in simple words

1. Every model you build on will eventually be switched off.
2. Not "might". Will. Plan for it from the first day.
3. There are three stages and they mean different things.
4. **Legacy** means still working, no longer recommended, no further updates.
5. **Deprecated** means a shutdown date has been announced.
6. **Retired** means the identifier no longer works and your requests fail.
7. Providers give notice, but the notice period varies a great deal by provider and by model tier.
8. The single thing that makes a forced migration safe is your own eval set, because it is the only way to know whether the replacement is good enough for your job.

50.12.2 PLAIN - a picture in your head

1. Think of a bus route being withdrawn.
2. First, notices appear at the stops. That is deprecation.
3. Then a date is given, and an alternative route is suggested.
4. On the date, the bus stops coming. That is retirement.
5. If you never tried the alternative route before the date, you find out on the morning you needed it whether it gets you to work on time.
6. Trying the alternative early, with your actual journey, is the whole of migration strategy.

Where this comparison breaks: a bus route is either running or not, and you find out at the stop. A retired model identifier can fail in a way your code does not handle, at the worst possible moment, in production, returning an error your retry logic loops on. Test the failure path too.

50.12.3 PLAIN - a worked example

1. Real, checkable policies and dates, as published in August 2026.

Provider	Class	Notice given
OpenAI	Generally available	At least 6 months
OpenAI	Specialized variants	At least 3 months
OpenAI	Preview models	As short as 2 weeks
Anthropic	Public models	At least 60 days
Google	Preview models	At least 2 weeks

2. And real retirement events, which show the policies being applied.

Identifier	Retired
claude-3-7-sonnet-20250219	19 February 2026
claude-sonnet-4-20250514	15 June 2026
claude-opus-4-1-20250805	5 August 2026
gpt-5-2025-08-07 (announced)	11 December 2026

3. Read the third row against the first table. That snapshot was released in August 2025 and retired in August 2026. One year of life.
4. Read the fourth row. Announced on 11 June 2026 for shutdown on 11 December 2026: exactly the stated six months.
5. That is the realistic planning number. A pinned snapshot at a major provider has historically lasted roughly one to two years, with a few months of warning at the end.
6. Note also that whole product features get retired, not only models. In 2026 OpenAI announced shutdown dates for several platform features including its hosted evaluation product. Do not build your only eval harness inside a vendor's product.

50.12.4 PLAIN - what is really happening inside

1. Why retirement happens: serving old weights occupies accelerators that could run new ones, old models lack current safety mitigations, and every supported version multiplies the testing burden.
2. Where to find the schedule: every major provider publishes a deprecations page listing model identifier, deprecation date, retirement date and the recommended replacement. Read it quarterly. Subscribe to the changelog.
3. The migration procedure, in order.

```
1. Read the deprecation notice; note both dates.
2. Identify the suggested replacement id.
3. Run your eval set on old and new, same prompts.
4. Read the failure list, not just the score.
5. Fix prompts for the new model's habits.
6. Re-run. Compare cost and latency too.
7. Ship behind a flag; keep the old id available.
8. Watch production metrics; then remove the flag.
9. Delete the old id from config before the date.
```

4. Step 3 is the one that makes the rest possible. Without an eval set, steps 4 to 8 are guesswork and step 9 is a leap of faith.
5. Expect prompt changes to be needed. A newer model in the same family often needs a shorter prompt, because instructions written to work around an older model's weaknesses can actively hurt a newer one.

6. Budget the work. For a small production system with a good eval set, a migration is typically days. Without one, it is weeks and the outcome is unknown.

50.12.5 TECHNICAL - the engineer's version

1. Operational controls worth having in place before you need them: the model identifier in configuration, a per-request log of the identifier used, a feature flag to switch identifiers without deploying, and an alert on the specific API error returned for a retired model.
2. Consider a fallback chain in code: primary snapshot, then a named replacement, then a degraded path. Test the fallback deliberately, because an untested fallback is not a fallback.
3. If continuity matters more than anything else, the alternative is running open-weight models yourself. Then nobody can retire your model, and the cost is that you own the serving, the security patching and the hardware. That is a genuine trade, not a free option.
4. Established fact: the notice periods and retirement dates above, as published in August 2026. They are policies, not contracts, and they change; check the current pages rather than trusting this table.
5. The honest version of "the provider will look after you": the provider will give you a date and a suggested replacement. Whether the replacement does your job is entirely your problem, and only your eval set can answer it.

50.12.6 WORDS - remember these

1. Deprecated — announced for removal — still functional, with a published retirement date and a recommended replacement.
2. Retired — switched off — the identifier no longer serves requests and calls return an error.
3. Legacy — still working, no longer recommended — a supported but frozen version receiving no updates.
4. Notice period — how long you get to move — the interval between deprecation announcement and retirement.
5. Migration — moving to the replacement — re-pinning the identifier and re-validating behaviour against your own evaluation set.
6. Fallback chain — what to use when the first choice fails — an ordered list of model identifiers with a degraded final option.

50.13 What you should actually do

50.13.1 PLAIN - in simple words

1. Here is the whole procedure, in order, for choosing a model for a job.
2. One. Write down the job in one sentence, and what a good answer looks like.
3. Two. Build the eval set from section 50.8. Thirty to a hundred real examples with a marking rule.
4. Three. Shortlist two or three candidate models. Use public benchmarks only for this step, only to shortlist, and never to decide.
5. Four. Run your eval set on all of them, with identical prompts.
6. Five. Read every failure. The failures tell you more than the score.
7. Six. Measure cost and latency at the same time, on the same runs.
8. Seven. Decide using all three numbers together, not quality alone.
9. Eight. Pin the dated identifier you chose.
10. Nine. Put the failures you saw into the eval set as new items.
11. Ten. Re-check every three months, and whenever a deprecation notice arrives.
12. That is it. It is not sophisticated. It is just done consistently.

50.13.2 PLAIN - a picture in your head

1. This is how you would hire someone, and you already know how to do that.
2. You do not hire from a league table of universities.
3. You shortlist from credentials, then you set a work sample task that resembles the actual job.
4. You watch how they handle the hard parts, not just whether they finish.
5. You consider salary and availability alongside skill.
6. And you review after a few months, because circumstances change.

Where this comparison breaks: a person learns your job over time and asks questions when confused. A pinned model snapshot is exactly as good on day 300 as on day 1, and never tells you it is uncertain unless you build that in.

50.13.3 PLAIN - a worked example

1. What to measure, and what a reasonable target looks like.

What to measure	How	Why it matters
Pass rate overall	Your eval set	Does it do the job
Pass rate per slice	Eval set slices	Where it fails
Cost per task	Tokens times price	Unit economics
Time to first token	Client-side timer	Feels responsive
Tokens per second	Client-side timer	Long output speed
Run-to-run variance	3 repeat runs	Is a gain real
Refusal rate	Refusal slice	Over-caution
Format validity	Schema check	Downstream breakage

2. A worked cost calculation, using round illustrative prices rather than any provider's current ones, since prices change monthly.
3. Suppose a task uses 4,000 input tokens and 800 output tokens, and the price is 3 dollars per million input and 15 dollars per million output.
4. Input cost: 4,000 / 1,000,000 * 3 = 0.012 dollars.
5. Output cost: 800 / 1,000,000 * 15 = 0.012 dollars.
6. Total 0.024 dollars per task. At 50,000 tasks a month that is 1,200 dollars.
7. Now suppose a model scoring 4 points higher costs three times as much. That is 3,600 dollars a month for 4 points on a 50-item eval set, where one standard error is about 5.7 points.
8. Written that way the decision is obvious, and it was invisible while you were looking at a leaderboard.

50.13.4 PLAIN - what is really happening inside

1. Why "try two or three" and not one or ten.
2. One tells you nothing comparative. Ten is a research project you will not finish, and the differences between the middle candidates will be inside your noise floor anyway.
3. Two or three, chosen to be genuinely different, is the sweet spot: for example one frontier model, one cheaper model from the same provider, and one from a different provider or an open-weight model you host.
4. Include a cheaper model deliberately. The most common real finding is that the cheap model is good enough for 80 per cent of traffic, and a routing rule sends the rest to the expensive one.
5. Why re-check periodically: models change under aliases, new versions appear, prices fall, and your traffic mix drifts as users learn what your product can do.
6. Why the failure list beats the score: a 90 per cent pass rate where the 10 per cent are harmless is fine, and a 95 per cent pass rate where the 5 per cent are confident wrong answers about money is not.

50.13.5 TECHNICAL - the engineer's version

1. Measure latency properly and separately. Time to first token dominates perceived responsiveness in streaming interfaces; output tokens per second dominates total completion time for long outputs. Report both, at the median and at the 95th percentile, not as a mean.
2. Measure from where your users are. A model served in another region can add more latency in transit than the difference between two models.
3. Control the comparison. Same prompts, same temperature where settable, same context, same tools, same time of day. Provider load varies through the day and will otherwise contaminate your latency numbers.
4. Compute your noise floor first: run one model three times and record the spread. Any difference between models smaller than that spread is not a result.
5. Keep the artifacts: raw outputs, identifiers, timestamps and prices. When something changes in three months, the old raw outputs are the only way to prove what changed.
6. Established fact: the arithmetic and the method here. Active research: automated evaluation set generation and automatic prompt optimization, both of which are useful and neither of which removes the need for a human to define what a good answer is.

50.13.6 WORDS - remember these

1. Shortlist — the two or three you will actually test — candidate models chosen from public evidence before task-specific measurement.
2. Noise floor — the wobble between identical runs — the run-to-run standard deviation below which differences are meaningless.
3. Time to first token — how long before words appear — the latency from request to first streamed token, distinct from total completion time.
4. Unit economics — what one task costs — the token cost per completed task at your actual input and output lengths.
5. Routing — sending easy work to a cheap model — a policy that selects a model per request based on difficulty, cost or latency budget.
6. Regression — the new thing is worse at something — a drop on any slice of your evaluation set after a change.

50.98 Common wrong ideas

1. Wrong: benchmark scores measure intelligence. Right: they measure performance on a fixed list of items under one protocol. Intelligence is not defined well enough to be scored, and no benchmark claims to.
2. Wrong: a higher MMLU means it is better for my task. Right: MMLU is four-option factual recall and is saturated. It predicts almost nothing about summarizing your emails or fixing your code. Only your own eval set answers that.
3. Wrong: the leaderboard is objective. Right: every leaderboard embeds choices about prompts, extraction, subsets and sampling, and preference leaderboards additionally embed the taste and the question mix of whoever voted.
4. Wrong: the model knows its own training cutoff. Right: the cutoff is not stored in the weights. Any date it gives is repeated from post-training data or guessed, and the felt cutoff is usually earlier than the real one.
5. Wrong: the same model name always means the same model. Right: undated product names are aliases and get repointed. Only a dated snapshot identifier is fixed, and even that is eventually retired.
6. Wrong: if two labs disagree, one is cheating. Right: prompt template, few-shot count, extraction, subset, sampling, system prompt and tool access routinely explain gaps of several points, with nobody at fault.

7. Wrong: contamination means the scores are fake. Right: contamination inflates scores by an unknown and uneven amount. Matched fresh benchmarks show drops of up to about 8 points for some model families and near zero for others.
8. Wrong: a 2-point lead is a meaningful lead. Right: on GPQA Diamond one standard error is about 2 points and one item is 0.5 points. On a single AIME paper, one question is 6.7 points.
9. Wrong: pass@k is just the score. Right: pass@k rises with k by construction. pass@1 and pass@100 are different quantities and cannot be compared.
10. Wrong: benchmarks are useless, so ignore them. Right: they are useful for shortlisting, for tracking the field over years, and for spotting obvious weakness. They are not useful for choosing between two close models for your specific job.

50.99 Chapter summary in 20 lines

1. A benchmark is a fixed set of items with known answers plus a rule for scoring a response. Nothing more.
2. Every run has five parts: items, prompt template, generation settings, answer extraction, and aggregation.
3. Only the items are usually published, so a score belongs to a model plus a harness, not to a model.
4. MMLU (2020, 57 subjects, about 14,000 test items) and HellaSwag (2019) are saturated and now mostly sanity checks.
5. GSM8K (2021, 1,319 test items) is solved. MATH (2021), AIME and FrontierMath carry the maths signal now.
6. HumanEval (2021, 164 problems, pass@k) is tiny, isolated and leaked. MBPP (2021, 974 problems) has the same shape.
7. GPQA (2023, 448 items; Diamond 198) is hard four-option science recall, not the ability to do science.
8. SWE-bench (2023, 2,294 tasks; Verified 500) scores whether real repository tests pass, and measures the scaffold as much as the model.
9. ARC-AGI keeps a held-back set and reports cost per task, which more benchmarks should copy.
10. Few-shot examples mainly fix output format. Chain-of-thought uses the model's own output as working memory and transforms maths scores.
11. pass@k = 1 - C(n-c,k)/C(n,k). With n=10 and c=2, pass@1 is 0.20 and pass@5 is 0.78. The model did not improve; you bought more tries.
12. Two labs disagree because of prompts, shots, extraction, subsets, sampling, system prompts, tools, or bugs. Ask which harness and version.
13. Contamination is near-inevitable when training on the web. Detection by n-gram overlap, canaries, perplexity, old-versus-new splits and completion tricks can prove presence but never absence.
14. Saturation, Goodhart's law, self-reported numbers, cherry-picked baselines, best-of-n reporting, tool asymmetry and truncated axes are the seven ordinary ways a number misleads without anyone lying.
15. Preference arenas measure real user taste well and truth badly, with documented length and style bias; a 10-point gap is a 51 per cent win rate.
16. Inside a company, an eval is a private regression suite of 30 to 100 real examples with per-item marking rules, run on every change.
17. Training cutoff and release date are different, historically separated by months to over a year, and models are unreliable about their own cutoff.
18. Product names are mutable aliases; dated snapshots are fixed. Pin the snapshot in production and re-run your evals when you change it.
19. Every model is eventually retired. Typical notice in 2026 was at least six months at OpenAI, at least 60 days at Anthropic, about two weeks for preview channels.
20. Shortlist with public benchmarks, decide with your own eval set, measure cost and latency alongside quality, and re-check every few months.

51

CHAPTER

KB

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

51.0 What this chapter gives you

1. You will be able to say what changes when a model stops learning and starts answering, and give the memory numbers for both.
2. You will be able to read a real API request body field by field, and say what the system prompt is and what it is not.
3. You will be able to explain, with arithmetic, why the first token is slow for one reason and the rest are slow for a completely different reason.
4. You will be able to name the serving tricks that make a model fast, and say why the same model runs at different speeds at different providers.
5. You will be able to build a cost model for a real application and decide honestly whether to self-host.
6. You will be able to run a model on your own laptop with two named tools, and predict beforehand what speed you will get.
7. You will be able to download weights, read a model card, and pick a quantization level on purpose rather than by guessing.
8. You will be able to explain retrieval-augmented generation step by step, including how approximate nearest neighbour search works.
9. You will be able to explain tool calling and agents precisely, including why every tool is a security boundary and why long agent runs fail.
10. You will be able to state the real limits, including why hallucination is intrinsic and why prompt injection is still unsolved as of August 2026.

51.1 Inference versus training: what changes

51.1.1 PLAIN - in simple words

1. Training is the process that produced the numbers inside the model.
2. Inference is the process of using those numbers to answer a question.
3. During training, every number in the model can move.
4. During inference, not one of them moves. They are read and never written.
5. Training needs to remember, for each number, which way it was moving and how fast it has been moving lately. That is several extra copies.
6. Inference needs the numbers themselves, plus some working notes about the conversation so far.
7. So the same model needs roughly ten times less memory to use than to train.
8. Training is judged on work finished per week. Inference is judged on how long one person waits for one answer.
9. A training run that takes 30 days is normal. An answer that takes 30 seconds feels broken.

51.1.2 PLAIN - a picture in your head

1. Think of a very large printed cookbook that a team of chefs wrote over a year.
2. Writing it was training. Drafts, corrections, arguments, notes in the margins about which way each recipe was heading.
3. All of that scaffolding took far more desk space than the finished book.
4. Cooking from it is inference. You open the book, read, and cook.
5. You do need the book itself open in front of you, and a small notepad for what you have already done in this particular meal.
6. But if a customer waits 40 minutes for soup, the restaurant fails.

Where this comparison breaks: a chef can read one page and ignore the rest. A language model reads every page of the book for every single word it produces. That is the fact that governs the whole of this chapter, and no ordinary object behaves that way.

51.1.3 PLAIN - a worked example

1. Take a real model: Llama 3.1 8B, which has 8.03 billion parameters.
2. Chapter 47 gave the rule: memory equals parameter count times bytes per parameter.
3. At 2 bytes per parameter that is 16.1 billion bytes, which is 15.0 GiB.
4. To use the model, you need those 15.0 GiB plus working space.
5. To train the model, you need the same 15.0 GiB plus four more copies.
6. Here is the whole comparison, in GiB, for one 8B model:

Item	Inference	Training
Weights, 16-bit	15.0	15.0
Master copy, 32-bit	0	29.9
Gradients, 32-bit	0	29.9
Optimizer state (two)	0	59.8
KV cache, 8K tokens	1.0	0
Activations	0.5	about 28
Framework overhead	1.0	2.0
Total	about 17.5	about 165

7. So the same model is a 17.5 GiB problem to run and a 165 GiB problem to train fully. That is a factor of about 9.4.
8. 17.5 GiB fits on one consumer graphics card with 24 GB.
9. 165 GiB does not fit on any single card sold in 2026, including the 141 GB NVIDIA H200.

51.1.4 PLAIN - what is really happening inside

1. In training, each batch of data goes forward through the network, an error is measured, and the error is pushed backwards.
2. Pushing it backwards requires remembering every intermediate value from the forward pass. Those are the activations, and they are enormous.
3. In inference, none of that exists. There is no backward pass, so no activation needs to be kept after the layer that produced it is done.
4. What is kept instead is different: a per-token record called the KV cache, which we come to in section 51.4.
5. So inference memory is dominated by two things only: the frozen weights, and the KV cache that grows with the length of the conversation.
6. And the weights are read-only, which means many requests can share exactly one copy of them in memory. This is why batching works at all.

51.1.5 TECHNICAL - the engineer's version

1. Training memory per parameter under the standard mixed-precision recipe with Adam: 2 bytes bf16 weights, 4 bytes fp32 master, 4 bytes fp32 gradient, 8 bytes for Adam's first and second moments. Total 18 bytes.
2. Inference memory per parameter: 2 bytes at bf16, 1 byte at int8, about 0.5 bytes at 4-bit. No optimizer state, no gradients.
3. The workload character inverts. Training is throughput-bound and runs at high model FLOPs utilization, typically 35 to 55 percent on well-tuned clusters. Batch-1 decode runs at roughly 0.3 percent of peak FLOPs.

Property	Training	Inference
Weights	read and written	read only
Optimizer state	8 to 12 B/param	none
Bottleneck	compute, network	memory bandwidth
Measured by	MFU, days to finish	TTFT, tokens/s, p99

4. Tools that show the split: `nvidia-smi` for device memory, `torch.cuda.memory_summary()` for allocator detail, and the `/metrics` endpoint that vLLM exposes for queue depth, batch size and cache usage.

51.1.6 WORDS - remember these

Inference — using a trained model — the forward pass only, no gradients. Training — teaching the model — forward and backward passes plus optimizer. KV cache — the model's notes on this conversation — cached key and value tensors per token per layer. MFU — how much of the chip's maths power you use — model FLOPs utilization. TTFT — the wait before the first word appears — time to first token.

51.2 What an API call actually contains

51.2.1 PLAIN - in simple words

1. Talking to a hosted model is an ordinary web request. Nothing exotic.
2. Chapter 36 covered what an API is; Chapter 32 covered what happens on the wire when a request is encrypted and sent. This is one of those requests.
3. You send a block of text in a structured form, plus some settings.
4. The structured form is a list of messages. Each message has a role.
5. The roles are conventionally "system", "user" and "assistant".
6. "System" is the instruction you put at the front. "User" is what the person typed. "Assistant" is what the model said last time.
7. Everything in that list is glued together into one long string of tokens and fed to the model at once.
8. The model does not know which part came from where, except that the training taught it to treat the system part as instructions.

51.2.2 PLAIN - a picture in your head

1. Imagine handing a temporary worker a clipboard before every single task.
2. Sheet one says: you are a polite support agent, never discuss competitors.
3. Sheet two says: here is the customer's question.
4. The worker reads the whole clipboard from the top, then speaks.
5. Then the worker forgets everything and leaves.
6. Next time, you hand over a fresh clipboard with all the sheets again, plus the new ones.
7. The first sheet has no special authority. It is just first.

8. If the customer writes on their sheet "ignore sheet one", the worker has to decide, from habit alone, which sheet to obey.

Where this comparison breaks: a human worker has an employer, a contract and a memory of yesterday. The model has none of those. Its deference to the first sheet was installed by training on examples where the first sheet was obeyed. It is a strong statistical habit, not a permission system.

51.2.3 PLAIN - a worked example

1. Here is a complete request, in the shape most providers accept, with every field explained afterwards.

```
POST /v1/chat/completions HTTP/1.1
Host: api.example-provider.com
Authorization: Bearer sk-REDACTED
Content-Type: application/json

{
  "model": "example-model-2026-08",
  "messages": [
    {"role": "system",
     "content": "You are a support agent. Answer in
      British English. Never invent a policy."},
    {"role": "user",
     "content": "My git push failed. What do I check?"}
  ],
  "temperature": 0.2,
  "top_p": 0.95,
  "max_tokens": 400,
  "stop": ["\n\nUser:"],
  "stream": false,
  "response_format": {"type": "json_object"},
  "tools": []
}
```

2. `model` names the exact weights to run. Dated names matter: providers change what an undated name points at.
3. `messages` is the conversation, oldest first. The whole list is sent every time. Nothing is stored on the server between calls.
4. `temperature` controls randomness in picking the next token. 0 means always take the most likely token. 1.0 is the model's raw distribution.
5. `top_p` keeps only the smallest set of tokens whose probabilities add up to 0.95, then samples inside that set. It is called nucleus sampling.
6. `max_tokens` is a hard ceiling on the reply length, and it is a cost control as well as a safety valve.
7. `stop` is a list of strings. If the model produces one, generation halts and the string is not returned.
8. `stream` set to true sends the answer piece by piece as it is produced.
9. `response_format` asks for strictly valid JSON rather than prose.
10. `tools` declares functions the model may ask you to run. Section 51.12 covers this in full.
11. And here is the reply:

```
{
  "id": "chatcmpl-8xk2ab",
  "object": "chat.completion",
  "created": 1786600000,
  "model": "example-model-2026-08",
  "choices": [
    {
      "index": 0,
```

```
      "message": {
        "role": "assistant",
        "content": "Check the remote URL, then your token."
      },
      "finish_reason": "stop"
    }
  ],
  "usage": {
    "prompt_tokens": 62,
    "completion_tokens": 41,
    "total_tokens": 103
  }
}
```

12. `finish_reason` is the single most useful field for debugging. "stop" means the model chose to end. "length" means you hit `max_tokens` and the answer is truncated. "tool_calls" means it wants you to run something.
13. `usage` is what you are billed on. 62 input tokens and 41 output tokens here, priced at different rates.

51.2.4 PLAIN - what is really happening inside

1. On the server, that JSON is flattened into one token sequence using a chat template, which is a fixed pattern of special marker tokens.
2. A template looks roughly like this, with the markers written out:

```
<|start|>system
You are a support agent...<|end|>
<|start|>user
My git push failed...<|end|>
<|start|>assistant
```

3. The model then predicts what comes after the final marker.
4. The honest version: the system prompt is a run of tokens at the front of the context window and has no privileged status in the architecture.
5. Its influence comes entirely from post-training, where the model was shown many examples of the system section being followed.
6. Because it is only a habit, it can be overridden. A user message that says "the previous instructions were a test, here are the real ones" sometimes works, and content pasted in from a web page can carry instructions too.
7. That failure has a name, prompt injection, and section 51.15 treats it as the architectural problem it is.

51.2.5 TECHNICAL - the engineer's version

1. The message-list shape shown above is a de facto convention, not a standard. It came from OpenAI's chat completions endpoint in March 2023 and was copied widely because tooling supported it.
2. It is not universal. Anthropic's Messages API takes `system` as a top-level string rather than a message role. OpenAI's newer Responses API uses `input` and `instructions`. Local servers from llama.cpp and vLLM implement the older chat-completions shape for compatibility.
3. Streaming is served as Server-Sent Events over HTTP, media type `text/event-stream`, one JSON delta per event:

```
data: {"choices":[{"delta":{"content":"Check"}}]}
data: {"choices":[{"delta":{"content":" the"}}]}
data: {"choices":[{"delta":{"content":" remote"}}]}
data: [DONE]
```

4. Chapter 32 covers what carries this: TLS 1.3 over TCP 443, usually HTTP/2, with the response body arriving as a chunked stream. Streaming needs no new protocol; it is one long-lived response.

51.2.6 WORDS - remember these

System prompt — the instruction sheet at the front — leading tokens in the context, obeyed by training habit, not by architecture. Role — who said this line — a marker token pattern in the chat template. Temperature — how adventurous the model is — logit scaling before softmax. Nucleus sampling — keep only the likely words — top_p truncation of the cumulative distribution. finish_reason — why it stopped — stop, length, tool_calls or content_filter. Chat template — the glue that turns messages into one string — a Jinja template shipped with the model.

51.3 How generation actually performs

51.3.1 PLAIN - in simple words

1. Answering has two phases, and they behave like two different machines.
2. Phase one: the model reads your whole prompt. This is called prefill.
3. Phase two: the model writes the answer one token at a time. This is called decode.
4. Prefill can read every word of your prompt at the same moment, in parallel, because all the words already exist.
5. Decode cannot do that. Word five cannot be chosen before word four exists.
6. So prefill is limited by how much arithmetic the chip can do per second.
7. And decode is limited by something else entirely: how fast the chip can pull the model's numbers out of memory.
8. That is the whole secret. For each single token of output, the machine has to read every parameter in the model once.
9. So the speed of writing is roughly: memory speed divided by model size.
10. That one division explains almost everything people find strange about model speed.

51.3.2 PLAIN - a picture in your head

1. Imagine a librarian who must consult every book in a library before saying one word.
2. To read your question, the librarian can put a hundred assistants on it at once, one per sentence. That is quick, and it scales with staff.
3. But to say each word of the answer, the librarian must personally walk the whole library again. Every shelf. Every book.
4. Adding assistants does not help with the walking. The corridor is the limit.
5. So the wait before the first word depends on how long your question was and how many assistants are free.
6. And the speed of the words after that depends only on how fast one person can walk the library once.
7. A bigger library means slower words, in direct proportion.

Where this comparison breaks: the librarian would remember the books. The chip does not. It genuinely re-reads the weights from memory for every token, because there is nowhere near enough fast on-chip storage to hold them. That re-reading is not waste from bad design; it is the fundamental shape of the computation.

51.3.3 PLAIN - a worked example

1. Take Llama 3.1 8B at 16-bit precision. Weights are 16.1 GB.
2. Run it on one NVIDIA H100 SXM. Its memory bandwidth is 3,350 GB per second. That figure is from NVIDIA's own specification sheet.
3. Decode ceiling equals 3,350 divided by 16.1, which is 208 tokens per second.

4. Real systems reach 60 to 80 percent of that, so expect 125 to 165 tokens per second for a single user. Measured single-stream figures for 8B on an H100 sit in that range.
5. Now the same model on a MacBook Air with the M5 chip. Its memory bandwidth is 153 GB per second, from Apple's technical specifications.
6. At 4-bit the model is about 4.7 GB. 153 divided by 4.7 is 32 tokens per second as a ceiling; expect 20 to 25 in practice.
7. Now prefill on the same H100. Prefill work is about 2 times parameters times prompt tokens.
8. For a 1,000-token prompt: 2 times 8.03 billion times 1,000 equals 16.1 trillion floating-point operations.
9. An H100 SXM does about 990 trillion dense bf16 operations per second. At 40 percent utilization that prompt takes about 41 milliseconds.
10. Compare the phases directly:

Phase	1,000-token prompt	400-token answer
Work	16.1 TFLOP	400 x 16.1 GFLOP
Bytes moved	16.1 GB once	16.1 GB x 400
Time on H100	about 41 ms	about 2.9 s

11. So the answer takes seventy times longer than reading the question, even though the question was longer.
12. One more real-world correction. The reader of this book is in India. A round trip to a server in the United States is typically 220 to 280 milliseconds.
13. So for short prompts, the network wait is larger than the prefill. Time to first token is network plus queueing plus prefill, and often the network wins.

51.3.4 PLAIN - what is really happening inside

1. Here is the timeline of one request, drawn out:

```
client         network      queue     prefill        decode
  |--------------->|
  |     250 ms     |---------->|
  |                | wait 0-100 ms |
  |                            |---->|  41 ms
  |                                  |-> tok 1  (TTFT)
  |                                  |-> tok 2
  |                                  |-> ...
  |                                  |-> tok 400
  |<-----------------------------------|  ~3.2 s total
```

2. Time to first token is everything up to the first arrow out: network, queue, prefill, and one decode step.
3. This is why a long prompt hurts your first token but not your typing speed, and why a long answer hurts total time but not the first token.

51.3.5 TECHNICAL - the engineer's version

1. The controlling ratio is arithmetic intensity: floating-point operations per byte moved from memory.
2. An H100 SXM has about 990 TFLOP/s dense bf16 against 3.35 TB/s, a balance point of roughly 295 FLOP per byte. Below that ratio you are bandwidth-bound.
3. Batch-1 decode does 2 FLOP per parameter and reads 2 bytes per parameter: 1 FLOP per byte. You are using about 0.3 percent of the chip's arithmetic.
4. The metric for the decode side is memory bandwidth utilization, or MBU, popularized by Databricks in 2023: achieved bytes per second divided by peak. Well-tuned stacks reach 60 to 85 percent.

5. Note that the vendor's headline FLOPS figure often includes 2:1 structured sparsity. NVIDIA's 1,979 TFLOPS figure for H100 bf16 is the sparse number; the dense number is 989.5. Use the dense one for language models.

Hardware	Bandwidth GB/s	8B fp16 ceiling
H100 SXM	3,350	208 tok/s
H200 SXM	4,800	298 tok/s
RTX 5090	1,792	111 tok/s
RTX 4090	1,008	63 tok/s
M5 Max, 40-core	614	38 tok/s
MacBook Air M5	153	9.5 tok/s

6. Measure it yourself with `llama-bench` from llama.cpp, which reports prompt processing and token generation separately, or with vLLM's `benchmark_serving.py`, which reports TTFT and inter-token latency percentiles under load.

51.3.6 WORDS - remember these

Prefill — reading your question — parallel forward pass over all prompt tokens, compute-bound. Decode — writing the answer — sequential single-token forward passes, memory-bandwidth-bound. Arithmetic intensity — sums done per byte fetched — FLOP per byte, compared against the hardware's balance point. MBU — how much of the memory pipe you are using — memory bandwidth utilization. Inter-token latency — the gap between words — the reciprocal of decode tokens per second.

51.4 The serving optimizations, each explained

51.4.1 PLAIN - in simple words

1. Every trick in serving exists to fight one of two enemies: reading the weights too often, or leaving the machine idle.
2. The KV cache stops the model re-reading your whole conversation for every new word.
3. Continuous batching stops the machine idling while one user thinks.
4. Speculative decoding lets a small fast model guess ahead so the big model can check several words at once.
5. Quantization shrinks the weights so there is less to read per token.
6. Tensor parallelism splits one model across several chips so their memory pipes add up.
7. Prefix caching stores the processed form of a shared opening so it is not re-processed for every user.
8. Streaming does not make anything faster, but it makes the wait feel far shorter, which is often what people actually want.

51.4.2 PLAIN - a picture in your head

1. Think of a busy kitchen with one very slow oven.
2. The KV cache is keeping the prepared ingredients on the counter instead of re-chopping everything for each course.
3. Static batching is waiting until six orders arrive, cooking all six, and serving nobody until the slowest one finishes.
4. Continuous batching is putting each new order into the oven the moment a space appears, and taking each dish out as soon as it is done.
5. PagedAttention is using many small identical trays instead of reserving a whole shelf per order in case it turns out to be large.
6. Speculative decoding is a junior cook who prepares the next three likely steps; the chef glances at them and either accepts all three or redoes them.

Where this comparison breaks: in a kitchen, more staff means more parallel cooking. On a GPU, the constraint is a single memory bus that all the work shares, so most of these tricks are about using one bus better, not about adding more hands.

51.4.3 PLAIN - a worked example

1. Start with the KV cache, because it is the one that changes the order of the arithmetic.
2. Without a cache, to produce token number t the model must process all t-1 previous tokens again from scratch.
3. Generating 500 tokens after a 1,000-token prompt means processing, on average, 1,250 tokens, 500 times: 625,000 token-passes.
4. With a cache, each new token processes exactly one new token, because the previous keys and values are already stored: 500 token-passes.
5. That is a saving of 1,250 times for this case. The work goes from growing with the square of the length to growing with the length itself.
6. Now the price. The cache is real memory, and it is calculated exactly:

```
KV bytes per token
  = 2 (keys and values)
  x layers
  x key/value heads
  x head dimension
  x bytes per number

Llama 3.1 8B, 16-bit:
  = 2 x 32 x 8 x 128 x 2  = 131,072 bytes = 128 KiB

Llama 3.1 70B, 16-bit:
  = 2 x 80 x 8 x 128 x 2  = 327,680 bytes = 320 KiB
```

7. So an 8B model with a 128,000-token context holds 16 GiB of cache, which is more than the 15 GiB of weights.
8. This is why serving systems spend so much effort on cache memory, and why grouped-query attention, which cuts the number of key/value heads from 64 to 8, was such an important change. It divided this table by eight.

51.4.4 PLAIN - what is really happening inside

1. Static batching groups requests, runs them together, and returns them together. Every request waits for the longest one in the group.
2. Continuous batching, first described in the 2022 Orca paper, works at the level of single decode steps instead.
3. At each step the scheduler looks at every active sequence, runs one token for all of them in one fused operation, then evicts finished ones and admits waiting ones immediately.
4. The result is that the batch is never allowed to shrink, so the memory pipe stays busy. Reported throughput gains over static batching are large, often several times, at the same latency.
5. PagedAttention, from the vLLM paper at SOSP 2023, borrows the operating system idea of paging from Chapter 12.
6. Instead of reserving a contiguous block of cache for each sequence's maximum possible length, it allocates fixed-size blocks, typically 16 tokens each, and keeps a block table per sequence.
7. This removes internal fragmentation and lets several sequences share blocks, which matters when many requests begin with the same system prompt.
8. Speculative decoding uses a small draft model to propose the next k tokens cheaply, then runs the large model once over all k proposals in parallel.

9. Because the large model's forward pass over k tokens costs almost the same as over one token, accepted guesses are nearly free. Rejected ones cost only the draft work.
10. The mathematics were published independently by two groups in 2023, and the method is exact: the output distribution is identical to running the large model alone. It is a speed trick, not a quality trade.

51.4.5 TECHNICAL - the engineer's version

1. The optimizations, with what each actually buys:

Technique	What it saves	Typical effect
KV cache	recompute	quadratic to linear
Continuous batching	idle GPU	2 to 10x throughput
PagedAttention	cache memory	19 to 27% memory
Speculative decoding	decode steps	1.5 to 3x latency
4-bit weights	bytes per token	about 3x decode
Tensor parallel, 2 GPUs	bandwidth	1.6 to 1.9x
Prefix caching	prefill	up to 90% TTFT

2. Tensor parallelism splits each weight matrix across GPUs, so each device reads only its shard. Two GPUs give roughly twice the aggregate bandwidth, less an all-reduce over NVLink after every layer. Scaling is sub-linear and degrades sharply without a fast interconnect.
3. Quantized serving matters twice over: 4-bit weights cut both memory and bytes read per token. Activation quantization to FP8 additionally raises prefill throughput on Hopper and Blackwell hardware.
4. Why the same open model is faster at provider A than provider B, in order of how much it usually explains: the hardware generation, the quantization they silently chose, the batch size policy they run, the current queue depth, and the geographic distance to you.
5. That list is the reason a published benchmark of "model X speed" is close to meaningless without naming the deployment.

51.4.6 WORDS - remember these

KV cache — saved notes so the model need not re-read — cached key and value projections per token per layer. Continuous batching — slot new work in as space frees — iteration-level scheduling, from the Orca paper, 2022. PagedAttention — memory in small blocks, not big reservations — block-table KV allocation, vLLM, 2023. Speculative decoding — a fast guesser the big model checks — draft-and-verify with an exact acceptance rule. Tensor parallelism — one model split across chips — sharded matrix multiply with all-reduce per layer. Prefix caching — reuse the shared opening — KV reuse keyed on a common token prefix.

51.5 Cost and capacity

51.5.1 PLAIN - in simple words

1. Hosted models are sold by the token, and tokens in cost less than tokens out.
2. Chapter 47 explained what a token is: roughly three quarters of an English word, so 1,000 tokens is about 750 words.
3. The reason input is cheaper is exactly section 51.3. Your prompt is read in parallel in one pass, so it is cheap per token.
4. The answer is produced one token at a time, and each token costs a full read of the model. That is the expensive part.
5. Two discounts exist almost everywhere. Batch work that can wait up to a day is half price. Repeated openings you have marked as cacheable are about a tenth of the input price.

6. Building your own server instead is not obviously cheaper. You rent the machine by the hour whether anybody uses it or not.

51.5.2 PLAIN - a picture in your head

1. Think of a taxi where the meter runs differently for the two halves of the journey.
2. Listening to your address is quick and charged at a low rate per word.
3. Driving you there is slow and charged at a high rate per street.
4. Batch pricing is agreeing to be picked up sometime today rather than now, for half fare.
5. Cache pricing is a regular route the driver already knows, so the first part of the journey needs no navigation.
6. Buying your own car instead is the self-hosting decision. The car costs the same whether you drive it or not.
7. It only beats taxis if you drive a great deal, every day.

Where this comparison breaks: taxi fares scale with distance, which you cannot change. Token costs scale with how much text you send, which you control completely. Trimming a system prompt from 2,000 tokens to 400 is a real 80 percent cut on that part of every request, forever.

51.5.3 PLAIN - a worked example

1. Take a support assistant. Real shape, real numbers.
2. 10,000 conversations a day. Six turns each. A 800-token system prompt. 2,000 tokens of retrieved documents at the start. 100-token questions. 300-token answers.
3. Because the whole history is resent every turn, the input grows each turn: 2,900, then 3,300, 3,700, 4,100, 4,500, 4,900.
4. That totals 23,400 input tokens and 1,800 output tokens per conversation.
5. Now price it against three real published tiers from August 2026, in dollars per million tokens: a mid tier at 2 in and 10 out, a small tier at 1 and 5, and a very small tier at 0.10 and 0.60.

Choice	Per conversation	Per 30 days
Mid, 2 and 10	$0.0648	$19,440
Mid with caching	$0.0466	$13,980
Small, 1 and 5	$0.0324	$9,720
Very small, 0.10, 0.60	$0.0034	$1,026

6. Read that table twice. The same application costs nineteen thousand dollars a month or one thousand, depending on one decision.
7. The caching row assumes the 2,800-token opening is marked cacheable and hit on five of six turns, charged at one tenth, with a write surcharge on the first turn.

51.5.4 PLAIN - what is really happening inside

1. Renting one datacentre GPU in 2026 costs roughly 2 to 4 US dollars an hour on the open market, varying by provider, region and commitment. Take 2.50.
2. That is about 1,825 dollars a month, per GPU, running or idle.
3. One such GPU serving an 8-billion-parameter model with good batching produces on the order of 2,000 to 4,000 output tokens per second in aggregate across all users.
4. At 2,500 tokens per second held constantly, that is 6.6 billion output tokens a month, which works out at about 28 cents per million.
5. That looks unbeatable until you notice the words “held constantly”.
6. Compare the two cost shapes across real volumes:

Output tokens/month	Hosted at $0.60/M	Own GPU
10 million	$6	$1,825
100 million	$60	$1,825
1 billion	$600	$1,825
3 billion	$1,800	$1,825
10 billion	$6,000	$3,650

7. Break-even against a cheap hosted model is around 3 billion output tokens a month, and that is before you add a second GPU for redundancy, an engineer to run it, or the load spikes you must size for.
8. The honest version: self-hosting is rarely cheaper below very high sustained volume. The good reasons to self-host are usually not cost.
9. They are: data must not leave your network, you need a model that no provider hosts, you need the exact same weights for years, or you need a latency floor you control.

51.5.5 TECHNICAL - the engineer's version

1. Published list prices, checked August 2026. Prices move; treat these as a snapshot, not a constant.

Model, August 2026	Input $/M	Output $/M
Claude Opus 5	5.00	25.00
Claude Sonnet 5	2.00	10.00
Claude Haiku 4.5	1.00	5.00
OpenAI gpt-5.6-sol	2.50	15.00
OpenAI gpt-5.6-terra	1.00	6.00
OpenAI gpt-5.6-luna	0.10	0.60

2. Discount structure, also checked August 2026: batch processing at 50 percent of both input and output on both providers; cache reads at 10 percent of the input rate; cache writes at 125 percent for a short time-to-live.
3. Long-context surcharges are now common. OpenAI applies a higher rate above a published input threshold for the whole request, which means one oversized prompt reprices the entire call.
4. Reasoning models bill their internal thinking tokens as output. A model that thinks for 4,000 tokens before writing 200 costs twenty-one times more than the visible answer implies. This is the single most common billing surprise of 2025 and 2026.

51.5.6 WORDS - remember these

Input token — text you send — prompt tokens, billed at the lower rate. Output token — text produced — completion tokens, billed several times higher. Batch API — cheaper if you can wait — asynchronous queue, typically 50 percent off with a 24-hour window. Prompt caching — pay less for a repeated opening — server-side KV reuse keyed on an exact prefix, billed at about 10 percent. Duty cycle — how much of the time your machine is busy — utilization, the term that decides every self-hosting argument.

51.6 Running one yourself, practically

51.6.1 PLAIN - in simple words

1. You can run a real language model on your own laptop today, offline, for free, in about ten minutes.
2. llama.cpp is the engine. It is a C++ program that loads a model file and runs it on your processor, your graphics chip, or both.
3. Ollama wraps that engine in something friendly: one command to fetch a model, one to talk to it, and a small local server.

4. LM Studio is a desktop application with a window and buttons, for people who would rather not use a terminal.
5. vLLM, SGLang and TGI are for the other end: serving many users at once on datacentre graphics cards.
6. MLX is Apple's own numerical framework, built for Apple silicon, and it is noticeably faster than the alternatives on those machines.
7. Start with Ollama. Move to llama.cpp when you want control. Move to vLLM when you have users.

51.6.2 PLAIN - a picture in your head

1. Think of playing recorded music.
2. llama.cpp is the amplifier and speaker: raw, capable, many knobs, no case.
3. Ollama is the all-in-one hi-fi with one power button and a catalogue of albums you can fetch by name.
4. vLLM is the sound system for a concert hall, designed for two thousand listeners rather than one.
5. MLX is a speaker built specifically for the shape of your room.

Where this comparison breaks: with music, the recording is identical whatever plays it. Here the tools change the result. Different engines pick different default quantizations, context lengths and sampling settings, so the same named model can give different answers and very different speeds.

51.6.3 PLAIN - a worked example

1. Here is Ollama, start to finish, on a Mac or Linux machine.

```
# install with a package manager, then check it runs
ollama --version

# fetch a small model (a few gigabytes) and chat
ollama pull qwen3:8b
ollama run qwen3:8b

# see what you have, and what is loaded right now
ollama list
ollama ps

# use it as a local HTTP service on port 11434
curl localhost:11434/api/chat -d '{
  "model": "qwen3:8b",
  "messages": [{"role":"user","content":"Explain TCP in 3 lines"}],
  "stream": false
}'
```

2. That last call has the same shape as the hosted API in section 51.2. Most local servers deliberately imitate it so your code does not change.
3. Now llama.cpp directly, which is one level down.

```
# build once, with Apple GPU support turned on
cmake -B build -DGGML_METAL=ON
cmake --build build --config Release -j

# run a model pulled straight from Hugging Face
./build/bin/llama-cli -hf ggml-org/gemma-3-1b-it-GGUF \
  -p "Write one sentence about routers."

# serve it on port 8080, 8k context, all layers on the GPU
./build/bin/llama-server -m model-Q4_K_M.gguf \
  -c 8192 -ngl 99 --port 8080

# measure prompt speed and generation speed separately
./build/bin/llama-bench -m model-Q4_K_M.gguf
```

4. The flag that matters most is `-ngl`, the number of layers to put on the GPU. Setting it to 99 means "all of them, if they fit".
5. If the model does not fit, layers stay on the processor and speed collapses, often by a factor of five or more. That is the single most common cause of "why is my local model so slow".

51.6.4 PLAIN - what is really happening inside

1. llama.cpp was started by Georgi Gerganov in March 2023 as an experiment in running Llama on a MacBook with no graphics card at all.
2. It defined a file format, now called GGUF, which packs the weights, the tokenizer and the metadata into one file that can be memory-mapped.
3. On the serving side the design goal is the opposite. vLLM keeps the GPU saturated with continuous batching and PagedAttention from section 51.4.
4. SGLang adds RadixAttention, a shared prefix tree, which pays off when thousands of requests begin with the same long system prompt.
5. Hugging Face's Text Generation Inference was the early default, but Hugging Face put it into maintenance mode on 11 December 2025 and now points new users at vLLM or SGLang. Treat it as a migration topic, not a choice.
6. MLX is Apple's array framework, released in December 2023, designed around unified memory so tensors are not copied between processor and graphics chip. Ollama added an MLX backend in version 0.19 on 27 March 2026.

51.6.5 TECHNICAL - the engineer's version

Tool	What it is	Use it when
llama.cpp	C/C++ engine, GGUF	any hardware, full control
Ollama	wrapper plus registry	fastest local start
LM Studio	desktop application	no terminal wanted
vLLM	Python serving engine	many users, GPUs
SGLang	serving, RadixAttention	shared prefixes
TGI	older HF server	existing systems only
MLX	Apple array framework	Apple silicon speed

1. macOS caps what the GPU may wire down, historically around 75 percent of unified memory. The `iogpu.wired_limit_mb` sysctl raises it. That is an implementation detail of one operating system version and it has been renamed before; check before relying on it.
2. Ollama's published benchmark for its MLX backend, March 2026, on a mixture-of-experts model with 35 billion total and 3 billion active parameters: decode from 58 to 112 tokens per second, prefill from 1,154 to 1,810 tokens per second.
3. Vendor figures, so read them as a direction, not a promise.
4. For serving, the vLLM invocation you will actually type is `vllm serve <model> --tensor-parallel-size 2 --max-model-len 32768`, and it exposes an OpenAI-compatible endpoint plus a Prometheus `/metrics` path.
5. Watch three numbers in production: running batch size, waiting queue length, and KV cache utilization. When cache utilization hits 100 percent the scheduler starts preempting sequences and latency becomes erratic.

51.6.6 WORDS - remember these

GGUF — the one-file local model format — successor to GGML, memory-mappable, carries weights, tokenizer and metadata. Offload — putting model layers on the graphics chip — `-ngl` or `n_gpu_layers`; partial offload is the usual cause of slow local models. Ollama — the friendly local runner — a daemon

plus registry over llama.cpp. vLLM — the standard GPU serving engine — PagedAttention, continuous batching, OpenAI-compatible server. MLX — Apple's numerical framework — unified-memory array library for Apple silicon, released December 2023.

51.7 What hardware you actually need

51.7.1 PLAIN - in simple words

1. Two numbers decide everything: how much memory you have, and how fast that memory is.
2. Memory decides whether the model runs at all. Bandwidth decides how fast it talks.
3. The memory rule from Chapter 47: parameters times bytes per parameter, plus a bit for the conversation and the software.
4. At 4 bits per parameter, a rough working rule is that the model needs about 0.6 GB per billion parameters, plus 2 to 3 GB of headroom.
5. So an 8-billion model at 4 bits wants about 7 GB free, and a 32-billion model wants about 22 GB free.
6. If it does not fit, it does not simply run slower. Part of it spills to the processor or to disk and the speed falls off a cliff.
7. Apple silicon is unusually good here, because the processor and graphics chip share one pool of memory. A laptop with 32 GB can load models that need a 3,000 dollar graphics card on a PC.
8. But Apple laptops have much narrower memory pipes than datacentre cards, so they load big models and then run them slowly.

51.7.2 PLAIN - a picture in your head

1. Think of memory as the size of your desk and bandwidth as how fast you can turn pages.
2. A PC graphics card is a small desk with an extremely fast hand.
3. An Apple laptop is a large desk with a moderate hand.
4. A datacentre card is a large desk with an extraordinarily fast hand, in somebody else's building, rented by the hour.
5. If the book does not fit on the desk, you keep fetching pages from the floor, and everything stops.

Where this comparison breaks: a person can skim. The model reads every page for every word. There is no skimming, so the "fast hand" is used at full stretch on every single token.

51.7.3 PLAIN - a worked example

1. Here are real machines, with real figures from the manufacturers, and what each can hold.

Machine	Usable memory	Bandwidth
MacBook Air M1/M2, 8 GB	about 5 GB	about 100 GB/s
MacBook Air M5, 16 GB	about 11 GB	153 GB/s
MacBook Air M5, 24 GB	about 18 GB	153 GB/s
MacBook Pro M5 Max, 128 GB	about 110 GB	614 GB/s
Desktop, RTX 4090	24 GB	1,008 GB/s
Desktop, RTX 5090	32 GB	1,792 GB/s
Server, H100 SXM	80 GB	3,350 GB/s

2. Now the speeds. These are the bandwidth ceiling from section 51.3 with a realistic 70 percent efficiency applied, for 4-bit models.

Model at 4-bit	Air M5 16 GB	M5 Max	RTX 5090
4B, 2.5 GB	43 tok/s	170 tok/s	500 tok/s
8B, 4.7 GB	23 tok/s	91 tok/s	267 tok/s
14B, 8.5 GB	12 tok/s	51 tok/s	148 tok/s
32B, 19 GB	will not fit	23 tok/s	66 tok/s
70B, 40 GB	will not fit	11 tok/s	will not fit

3. Being specific, as promised. On an 8 GB MacBook Air: models up to about 4 billion parameters at 4 bits run fine. An 8B at 4 bits technically loads and then swaps to disk, giving a few tokens per second and a hot machine. Anything larger will not run at all.
4. On a 16 GB Air: 8B is comfortable, 14B works, 32B does not fit.
5. On a 24 GB Air: 32B at 4 bits fits with a short context and delivers about 8 tokens per second, which is slower than reading speed and unpleasant.
6. A useful threshold: below about 10 tokens per second, most people stop using a model interactively. Above 25 it feels fine for chat.

51.7.4 PLAIN - what is really happening inside

1. Why unified memory helps so much: on a normal PC, the graphics card has its own memory, and anything not in it must cross the PCI Express bus, which moves about 64 GB per second on a sixteen-lane fourth-generation link.
2. That is twenty to fifty times slower than the card's own memory, so a model that half fits runs at roughly the speed of the slow path.
3. On Apple silicon there is one pool. There is no crossing, so a 70-billion model at 4 bits genuinely runs on a laptop, which no 3,000 dollar consumer graphics card can do.
4. Where Apple silicon is still limited, in two places.
5. First, bandwidth. Even the fastest laptop chip at 614 GB per second is under a fifth of an H100. Big models load and then crawl.
6. Second, prompt processing. Prefill is compute-bound, and Apple's graphics cores have far less matrix throughput than a datacentre card. Pasting in a long document can take many seconds before the first word appears, even though generation afterwards looks reasonable.
7. This asymmetry surprises people constantly. The machine feels fine in chat and terrible the moment you paste a file into it.

51.7.5 TECHNICAL - the engineer's version

1. Sizing formula for a local deployment, in bytes:

```
total = params x bytes_per_param
      + kv_bytes_per_token x context_length x concurrency
      + activations (0.5 to 1 GiB)
      + runtime overhead (0.5 to 2 GiB)
```

2. Mixture-of-experts models decouple the two limits: capacity is set by total parameters, speed by active parameters. A 26-billion-total, 4-billion-active model needs about 16 GB at 4 bits but decodes like a 4B model.
3. Community measurements in 2026 report roughly 150 tokens per second for such a model on an RTX 4090 or 5090 and roughly 80 on an M5 Max.
4. Apple's published bandwidths, August 2026: M5 in the MacBook Air, 153 GB/s; M5 Pro up to 307 GB/s with up to 64 GB; M5 Max 460 GB/s in the 32-core configuration and 614 GB/s in the 40-core, up to 128 GB.

5. NVIDIA's published bandwidths: RTX 4090 1,008 GB/s over 24 GB GDDR6X; RTX 5090 1,792 GB/s over 32 GB GDDR7; H100 SXM 3,350 GB/s over 80 GB HBM3; H200 4,800 GB/s over 141 GB HBM3e; B200 about 8,000 GB/s.
6. Observe the reality rather than the ceiling with `llama-bench`, which prints prompt-processing tokens per second and generation tokens per second as two separate rows. They will differ by one or two orders of magnitude, exactly as section 51.3 predicts.

51.7.6 WORDS - remember these

Unified memory — one pool shared by processor and graphics — Apple silicon's architecture; capacity advantage, bandwidth disadvantage. VRAM — the graphics card's own memory — dedicated GDDR or HBM, fast but small on consumer cards. Offload spill — part of the model living on the slow side — layers left on the CPU or paged to disk; the usual cause of a tenfold slowdown. Prompt processing speed — how fast it reads what you paste — prefill tokens per second, compute-bound, weakest on laptops. Active parameters — the part of a sparse model actually used — the figure that sets decode speed in a mixture-of-experts model.

51.8 Getting weights and reading a model card

51.8.1 PLAIN - in simple words

1. Hugging Face is the place where models live. It is a website and a set of tools, and it works much like a code-hosting site.
2. Each model has a repository with an owner and a name, files inside it, and a page of documentation called a model card.
3. As of 2026 the site hosts more than two million public models. Most are variations, fine-tunes and re-uploads of a much smaller number of originals.
4. You download either the original weights, which are large files in a format for research frameworks, or a converted single-file version for local tools.
5. The model card is the part people skip and should not. A good one tells you what the model was built from, what it may legally be used for, and where it fails.
6. A bad one tells you only that the model is excellent.
7. Then you choose a quantization level, which is the trade from Chapter 47: fewer bits per number means less memory and more speed, and a little less accuracy.

51.8.2 PLAIN - a picture in your head

1. Think of buying a second-hand car from an online listing.
2. The files are the car. The model card is the seller's description.
3. A good listing states the year, the mileage, the service history, what is worn, and what the car is not suitable for.
4. A bad listing says "runs great, must sell" and shows one photograph taken from a flattering angle.
5. The licence is the logbook. Without it you may not legally use the car for the purpose you had in mind, however well it runs.

Where this comparison breaks: a car's condition can be inspected by a mechanic in an hour. A model's real behaviour on your task can only be found by running your own evaluation on your own data, which is why Chapter 50's material on evaluation matters more than any card.

51.8.3 PLAIN - a worked example

1. A typical repository contains these files. Knowing what each is stops a lot of confusion.

```
config.json                 architecture: layers, heads, sizes
model-00001-of-00004.safetensors   weight shard 1 of 4
```

```
model.safetensors.index.json       which tensor is in which shard
tokenizer.json              the vocabulary and merge rules
tokenizer_config.json       special tokens, chat template
generation_config.json      default temperature, stop tokens
README.md                   the model card
LICENSE                     the terms you are agreeing to
```

2. To download, use the official command-line tool, which is `hf` in current versions and was named `huggingface-cli` before 2025:

```
hf download meta-llama/Llama-3.1-8B-Instruct \
  --local-dir ./llama-3.1-8b

# or fetch only one quantized file for local use
hf download bartowski/Meta-Llama-3.1-8B-Instruct-GGUF \
  Meta-Llama-3.1-8B-Instruct-Q4_K_M.gguf --local-dir .
```

3. Prefer files ending in `.safetensors` over `.bin`. The older `.bin` format is Python pickle, which can execute code when loaded. Safetensors, introduced in 2022, is a plain data format that cannot.
4. Now the quantization decision. These are real measured figures from llama.cpp's own tables for an 8-billion model, showing file size and the increase in perplexity, which is the confusion measure from Chapter 50. Lower is better.

Level	File size	Extra perplexity
F16	14.00 GB	baseline
Q8_0	7.96 GB	+0.0026
Q6_K	6.14 GB	+0.0217
Q5_K_M	5.33 GB	+0.0569
Q4_K_M	4.58 GB	+0.1754
Q3_K_M	3.74 GB	+0.6569
Q2_K	2.96 GB	+3.5199

5. Read the last row. Two-bit quantization on a small model does not degrade it slightly; it breaks it.
6. The decision, as a table:

Situation	Choose	Reason
It fits at 8-bit	Q8_0	effectively lossless
Normal case	Q4_K_M	the standard default
Just short of fitting	IQ4_XS	smaller, small loss
Far short	smaller model at Q4	better than Q2

51.8.4 PLAIN - what is really happening inside

1. What a good model card contains, in the order you should read it.
2. The base model and how this one relates to it: is it an original pre-trained model, an instruction-tuned version, or somebody's fine-tune of a fine-tune.
3. A description of the training data. Not the data itself, usually, but its sources, size, languages and cut-off date.
4. The licence, stated plainly, including whether commercial use is allowed and whether outputs may be used to train other models.
5. Intended use and out-of-scope use. Real cards say what the model is not for.
6. Limitations and known biases, in specifics rather than boilerplate.

7. Evaluation results with the benchmark names, the exact settings, and ideally the harness used, so the numbers can be reproduced.
8. What a bad card omits, in order of how much it should worry you: the licence, the training data description, the evaluation settings, and any statement of weakness at all.
9. A card that reports only wins on benchmarks, with no method and no limitations section, is marketing. Treat the numbers as unverified.
10. The idea is not new or informal. Model cards were proposed in a 2019 paper by Margaret Mitchell and colleagues, and datasheets for datasets in a 2018 paper by Timnit Gebru and colleagues. Both are conventions, widely adopted and nowhere enforced.

51.8.5 TECHNICAL - the engineer's version

1. Licences differ far more than people assume, and "open" is not one thing. Apache 2.0 and MIT are permissive. Llama's community licence adds an acceptable-use policy and a threshold on monthly active users. Some research-only licences forbid commercial use entirely.
2. "Open weights" means you can download the parameters. It does not mean open source: the training data and training code are usually not released. Say open-weights when that is what you mean.
3. Gated repositories require accepting terms with your account before the files resolve, so automated pipelines need a token with the acceptance already recorded.

51.8.6 WORDS - remember these

Model card — the honest description page — a structured README covering data, licence, intended use, limitations and evaluations. Safetensors — a weight file that cannot run code — a simple tensor container format, preferred over pickle-based `.bin`. Open weights — you can download the numbers — distinct from open source, which would include data and training code. Perplexity — how confused the model is by real text — exponentiated average negative log-likelihood; used here to price quantization damage. Gated repository — you must accept terms first — access controlled by account agreement on the hub.

51.9 Building something small from scratch, honestly

51.9.1 PLAIN - in simple words

1. You can train your own language model on a laptop. It is a genuinely good afternoon's work, and it will teach you more than ten articles.
2. Realistically, on a laptop, you are training something between one and fifty million parameters on a few tens of millions of characters of text.
3. That is between one thousandth and one ten-thousandth of the size of the models you use every day.
4. What it will do: produce text with the correct shape. Sentences that look like sentences. Correct punctuation. Local coherence for a line or two.
5. If you train it on Shakespeare, it produces convincing Shakespeare-shaped nonsense with character names and stage directions in the right places.
6. What it will not do: know any facts, follow an instruction, answer a question, hold a conversation, or reason.
7. Those abilities appear only with vastly more data and a separate instruction-tuning stage afterwards.
8. Do it exactly once, as an education. Then never ship it, because a downloadable open model is thousands of times better and free.

51.9.2 PLAIN - a picture in your head

1. Think of a child copying the appearance of adult handwriting before they can spell.
2. The loops are right. The spacing is right. The line has the rhythm of writing. From across the room it looks like a letter.
3. Your tiny model is at exactly that stage, and getting there takes an hour.

4. Learning to spell is the next thousand hours, and learning to write something worth reading is the thousand after that.

Where this comparison breaks: the child is learning towards meaning the whole time. The model has no such trajectory. Scale does not turn imitation into understanding by itself; it turns imitation into much better imitation, which turns out to be enough for a surprising number of tasks, and this distinction is exactly where thoughtful people still disagree.

51.9.3 PLAIN - a worked example

1. Concrete plan, with real numbers.
2. Data: the collected works of Shakespeare, about 1.1 million characters, which is a standard toy dataset.
3. Model: a transformer with 6 layers, 6 attention heads, 384 hidden dimensions, character-level vocabulary. That is roughly 10 million parameters.
4. Training: a few thousand steps. On an Apple silicon laptop this takes on the order of 10 to 30 minutes. On a rented graphics card, a couple of minutes.
5. Result: text like this in shape, though the words will be invented.

```
DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.
```

6. Every convention is right. Speaker names, colons, line breaks, capitalized verse. No sentence means anything.
7. Now the honest scaling. To get factual knowledge you need billions of tokens, not one million characters.
8. The compute-optimal ratio from the 2022 Chinchilla paper is roughly 20 training tokens per parameter. A useful 1-billion-parameter model implies about 20 billion tokens, which is not a laptop job.

51.9.4 PLAIN - what is really happening inside

1. The canonical implementations to read, by name, so you can find them.
2. minGPT, by Andrej Karpathy, 2020: a few hundred lines, written to be read rather than run fast.
3. nanoGPT, by the same author, January 2023: the practical version, which reproduces GPT-2 at 124 million parameters.
4. The video "Let's build GPT: from scratch, in code, spelled out", 2023, which builds the whole thing live in about two hours.
5. llm.c, 2024: the same training in plain C and CUDA, with no framework at all, useful for seeing where the time actually goes.
6. nanochat, October 2025: the full pipeline including tokenizer, pre-training, fine-tuning, evaluation and a chat interface, in one repository.
7. The nanochat numbers are the useful ones for calibration. Its own documentation reports training a model of GPT-2 capability for about 48 dollars, roughly two to four hours on a node of eight H100 cards, against the roughly 43,000 dollars that training GPT-2 cost in 2019.
8. That fall of nearly a thousand times in six years is one of the clearest single facts about this field, and it is about hardware, software and recipes together, not about any one breakthrough.

51.9.5 TECHNICAL - the engineer's version

1. What a laptop-scale run looks like against the real thing:

Property	Laptop toy	Small open model
Parameters	10 million	8 billion
Training tokens	1 million chars	15 trillion
Hardware	1 laptop chip	thousands of GPUs
Wall-clock	20 minutes	weeks
Can follow orders	no	yes

2. The alternative, and usually the right one: fine-tune an existing open model instead of training one.
3. Full fine-tuning of an 8B model needs the 165 GiB from section 51.1, so it is a multi-GPU job.
4. Parameter-efficient methods avoid that. LoRA, from a 2021 Microsoft paper, freezes the model and trains small low-rank matrices alongside it, typically 0.1 to 1 percent of the parameters.
5. QLoRA, 2023, adds 4-bit quantization of the frozen base, which brought fine-tuning a 7B model onto a single 24 GB card, and a 70B model onto a single 80 GB card.
6. When fine-tuning is the right answer: you need a fixed output format, a house style, a domain vocabulary, or you want to make a small cheap model imitate a large expensive one on one narrow task.
7. When it is the wrong answer: you want the model to know new facts. Facts change, fine-tuning bakes them in, and the model will still state the old ones confidently. Use retrieval, which section 51.11 covers.
8. The practical threshold: below roughly 1,000 good examples, prompt engineering and few-shot examples usually beat fine-tuning. Above roughly 10,000, fine-tuning usually wins.
9. Evaluate before and after on a held-out set you wrote yourself. Fine-tuning frequently improves the target task and quietly degrades everything else, a phenomenon called catastrophic forgetting.

51.9.6 WORDS - remember these

Pre-training — the long expensive stage — next-token prediction over a very large corpus, producing a base model. Base model — a text continuer, not an assistant — the pre-trained model before instruction tuning. Chinchilla ratio — how much text per parameter — about 20 training tokens per parameter for compute-optimal training, from the 2022 paper. LoRA — training a small patch instead of the whole model — low-rank adapter matrices added to frozen weights. Catastrophic forgetting — getting better at one thing and worse at everything else — degradation of unrelated capabilities after narrow fine-tuning.

51.10 Prompting as an engineering practice, not a trick

51.10.1 PLAIN - in simple words

1. A prompt is just the text you send. There is no magic word.
2. What works is the same thing that works when briefing a competent new colleague who cannot ask you questions.
3. Be specific about the task, the audience, the length and the format.
4. Show an example of what you want, because one example beats a paragraph of description.
5. Say what the output must look like, exactly, especially if a program will read it.
6. For multi-step problems, let the model work through the steps rather than demanding the answer immediately.
7. Break big tasks into several calls instead of one enormous one.
8. And now the honest part: the same request phrased two ways can give noticeably different quality. That is a real weakness of the technology, not a skill you lack.

51.10.2 PLAIN - a picture in your head

1. Think of writing a work order for a contractor who will do exactly what the paper says and will never phone you back.
2. "Fix the bathroom" gets you something. Possibly not the thing you wanted.

3. "Replace the tap with this model, keep the existing pipework, finish by Friday, leave the old tap in the box" gets you the thing you wanted.
4. Adding a photograph of a finished job you liked works better still.
5. The contractor is skilled. The failure was in the brief.

Where this comparison breaks: a real contractor who received an ambiguous brief would either ask or use judgement informed by having seen your house. The model has no judgement about your situation and cannot ask, so it fills gaps with the statistically ordinary choice, which may be nothing like your case.

51.10.3 PLAIN - a worked example

1. Five real improvements. In each pair, the second version is not longer for the sake of it; each addition removes one specific ambiguity.

```
1 BEFORE: Summarize this.
  AFTER:  Summarize the text below in 5 bullet points for a
          non-technical manager. Each bullet under 20 words.
          Do not mention anything not in the text.

2 BEFORE: Write a regex for emails.
  AFTER:  Write a Python regex matching the addresses in the
          list below and rejecting the ones after "BAD:".
          Return only the pattern, no explanation.

3 BEFORE: Is this contract clause risky?
  AFTER:  Read the clause. List each obligation it places on
          the buyer, then rate each Low/Medium/High risk with
          one reason. If the clause is ambiguous, say so.

4 BEFORE: Extract the data.
  AFTER:  Return JSON only, matching exactly this shape:
          {"invoice_no": string, "total": number,
           "currency": string, "date": "YYYY-MM-DD"}
          Use null for any field not present in the text.

5 BEFORE: What is 17% of 4,382 plus the VAT?
  AFTER:  Work step by step. State each arithmetic step on its
          own line, then give the final number. Use the
          calculator tool for each multiplication.
```

2. Improvement 2 supplies examples of both classes, which is what few-shot prompting means.
3. Improvement 4 pins the exact schema, which is the difference between code that parses and code that crashes at 3 a.m.
4. Improvement 5 does two things: asks for steps rather than a single leap, and moves the arithmetic to a tool, for reasons section 51.15 explains.

51.10.4 PLAIN - what is really happening inside

1. Why examples work: the model is completing a pattern. Two worked examples define the pattern far more sharply than a description of it.
2. Zero-shot means no examples, just an instruction. Few-shot means two to five worked examples in the prompt. Few-shot costs tokens on every call and is usually worth it for format compliance.
3. Why step-by-step working helps: each token the model writes becomes input for the next token. Written reasoning is literally extra computation, and it gives the model somewhere to put intermediate results, since it has no scratchpad of its own.
4. This was formalized as chain-of-thought prompting in 2022, and the effect is large on multi-step arithmetic and logic.

5. When it does not help: simple lookups, classification, and formatting tasks, where it adds cost and latency for nothing.
6. There is also a 2026 research finding worth knowing: more reasoning is not monotonically better. Accuracy against reasoning length often follows an inverted U, and past a point extra thinking flips correct answers to wrong ones. That result is active research, not settled.
7. Recent reasoning models do this internally and are trained for it, so instructing them to "think step by step" can be redundant or harmful. Read the provider's guidance for the specific model.

51.10.5 TECHNICAL - the engineer's version

1. Treat prompts as versioned artifacts. They belong in the repository, with a test suite, not pasted into a dashboard.
2. Prompt sensitivity is measurable and is a genuine limitation. Reported effects in the literature include changes in accuracy from reordering few-shot examples, from formatting choices such as separators, and from the position of the question relative to a long document.
3. The position effect has a name: models attend most reliably to the beginning and end of a long context and least reliably to the middle, described as "lost in the middle" in a 2023 paper.
4. Put the question after the documents, and put the most important document last.
5. Techniques with reasonable evidence: few-shot examples, explicit output schemas, decomposition into separate calls, self-consistency by sampling several answers and taking the majority, and constrained decoding when the output must parse.
6. Techniques that are mostly folklore: politeness, offering tips, threatening, role-play preambles such as "you are a world-class expert", and long lists of adjectives. Some show small effects in some studies and vanish in others.
7. The honest summary: prompt engineering is real but shallow. Its ceiling is low compared with fixing your retrieval, your data or your task decomposition.

51.10.6 WORDS - remember these

Zero-shot — no examples given — instruction-only prompting. Few-shot — a handful of worked examples — in-context learning from demonstrations in the prompt. Chain of thought — asking it to show working — eliciting intermediate reasoning tokens before the answer. Self-consistency — ask several times and take the majority — sampling multiple reasoning paths and voting. Prompt sensitivity — small wording changes, different quality — measured variance in output quality under semantically equivalent rephrasing.

51.11 RAG: giving a model your documents

51.11.1 PLAIN - in simple words

1. A model knows only what was in its training data, which stopped at some date, and which never included your company's files.
2. You cannot fix that by asking nicely. The information is genuinely not inside the weights.
3. But you can put the relevant text into the question. The model reads its context, so text you paste in is text it can use.
4. Doing that automatically is called retrieval-augmented generation, or RAG.
5. The pipeline is: cut the documents into pieces, turn each piece into a vector of numbers, store them, turn the question into a vector too, find the closest pieces, paste them into the prompt, and generate.
6. The vector is the embedding from Chapter 47: a list of numbers where similar meanings sit close together.
7. Every part of that pipeline can fail, and when the answer is wrong it is usually retrieval that failed, not the model.

51.11.2 PLAIN - a picture in your head

1. Think of an exam where the candidate may bring notes, but only one page.
2. Someone else chooses which page, from a filing cabinet, in one second, based on the question.
3. If they bring the right page, the candidate looks brilliant.
4. If they bring a page about the right topic but the wrong year, the candidate confidently gives last year's answer.
5. If they bring six pages of near-misses, the candidate gets distracted and the good page gets lost among them.
6. The candidate's ability is not what varies between these cases. The filing clerk's is.

Where this comparison breaks: a real candidate would say "these notes do not cover this question". A model usually will not. It will answer from the notes if it can, and from habit if it cannot, and both answers look equally confident.

51.11.3 PLAIN - a worked example

1. The pipeline, drawn out:

```
DOCUMENTS                          QUESTION
    |                                 |
 [ chunk ]                            |
    |                                 |
 [ embed ] -> vectors             [ embed ]
    |                                 |
 [ store  ] <--- nearest search ------+
    |                                 |
 top-k chunks -----> [ prompt ] <-----+
                          |
                     [ model ] -> answer
```

2. Real sizes for a small system: 500 documents, average 8 pages each, cut into chunks of about 500 tokens with 50 tokens of overlap.
3. That gives roughly 30,000 chunks. Embedded at 1,536 dimensions in 4-byte floats, the vectors occupy 30,000 x 1,536 x 4 bytes, which is 184 MB.
4. Embedding cost, at a published rate of about 0.02 dollars per million tokens for a small embedding model: 15 million tokens costs 30 cents. Embedding is cheap; that is not where the money goes.
5. At query time you embed one question, find the 5 nearest chunks, and paste about 2,500 tokens into the prompt.
6. That 2,500 tokens is charged on every single question, which is why section 51.5's cost table had retrieved context in it.

51.11.4 PLAIN - what is really happening inside

1. Chunking first, because it decides more than any other choice.
2. Chunk too small and each piece loses the context that made it meaningful. A sentence saying "this does not apply in India" is useless alone.
3. Chunk too large and the embedding becomes an average of several topics, which is close to nothing in particular, so it matches nothing well.
4. Strategies, in increasing order of effort: fixed size with overlap; splitting on structure such as headings and paragraphs; splitting on semantic shifts; and adding a short summary of the parent document to every chunk so it carries its own context.
5. Overlap exists so that a fact sitting on a boundary appears whole in at least one chunk. Ten percent is a common default.

6. Then embedding. An embedding model is a separate, smaller model that maps text to a fixed-length vector. Both the documents and the query must use the same one, and changing it means re-embedding everything.
7. Then storage and search. With 30,000 chunks you could compare the query to every vector, which is exact and fast enough. With 30 million you cannot.
8. So vector databases use approximate nearest neighbour search, and the dominant method is called HNSW.
9. Here is HNSW in plain words. Build a graph where each vector is a point joined to a few of its neighbours. Then build a second, sparser graph above it containing a random subset, and a third above that, and so on.
10. To search, start at the top layer, walk greedily to the closest point you can reach, then drop a layer and repeat from there.
11. The top layers move you across the whole space in a few hops. The bottom layer does the fine work. It is an express train, then a local train, then walking.
12. It is approximate: it can miss the true nearest neighbour. You trade recall against speed with a parameter, usually called `ef_search`.

51.11.5 TECHNICAL - the engineer's version

1. RAG was named in a 2020 paper by Patrick Lewis and colleagues at Facebook AI Research. HNSW is from a 2016 paper by Yury Malkov and Dmitry Yashunin. FAISS, the library that made billion-scale search routine, came from Facebook AI Research in 2017.
2. Vector store choices in 2026, honestly ranked by how often they are the right answer: pgvector inside an existing Postgres for anything up to tens of millions of vectors; Qdrant, Milvus or Weaviate when you need dedicated scale or filtering; Pinecone when you want it fully managed; Chroma or FAISS for prototypes and local work.
3. Pure vector search fails on exact tokens: part numbers, error codes, proper nouns and negations. Keyword search fails on paraphrase. Hybrid search runs both and fuses the results.
4. The standard fusion method is reciprocal rank fusion, from a 2009 paper by Cormack, Clarke and Buettcher: score each document by the sum of one over (a constant plus its rank) in each list.
5. It needs no score calibration, which is why it is used.
6. Re-ranking is a second, more expensive model that scores query-document pairs jointly, called a cross-encoder. Retrieve 50 candidates cheaply, re-rank to the best 5, pass those on. This is usually the single highest return improvement in a RAG system.
7. Evaluate retrieval separately from generation, always. Retrieval metrics: recall at k, mean reciprocal rank, normalized discounted cumulative gain. Generation metrics: groundedness, meaning every claim is supported by a retrieved chunk, and answer relevance.
8. The three honest failure modes:

Failure	Symptom	Usual fix
Retrieval miss	confident wrong answer	hybrid search, chunking
Crowded context	vague, hedged answer	fewer chunks, re-rank
Ignored context	contradicts the source	cite-and-quote prompt

9. The third one deserves a note. Models do sometimes override retrieved text with training-time knowledge, especially when the retrieved text is surprising. Requiring the model to quote the supporting sentence before answering reduces it measurably and makes the failure visible when it happens.

51.11.6 WORDS - remember these

RAG — fetch the right text and paste it in — retrieval-augmented generation, named in a 2020 paper. Chunk — one searchable piece of a document — a token-bounded span, usually with overlap, that is embedded as a unit. Embedding model — turns text into a vector of meaning — a bi-encoder producing a fixed-length dense representation. HNSW — express train, local train, then walking — hierarchical navigable small world graph for approximate nearest neighbour search. Hybrid search — keywords and meaning together — dense plus sparse retrieval fused, commonly by reciprocal rank fusion. Re-ranker — a second opinion on the shortlist — a cross-encoder scoring query and document jointly.

51.12 Tool use and function calling

51.12.1 PLAIN - in simple words

1. A model produces text. Only text. It cannot press a button or call a web service.
2. It writes down a request, in a structured form you specified, saying which function it wants and with what arguments.
3. Your program reads that request, decides whether to run it, runs it, and puts the result back into the conversation as a new message.
4. The model then continues, now with the answer in front of it.
5. That is the whole mechanism. The model never executes anything. Your code does.
6. Which means every tool you offer is a door in your system that a piece of text can ask to open. That is a security boundary, and it must be treated like one.

51.12.2 PLAIN - a picture in your head

1. Think of a consultant behind a glass wall with no phone and no keyboard.
2. They can write notes and slide them under the glass.
3. A note might say: look up order 4471 and tell me the status.
4. You read it, decide whether that is a reasonable thing to ask, look it up, and slide the answer back.
5. At no point did the consultant touch your systems. They asked. You acted.
6. If you slide back whatever any note asks for, without checking, the glass wall is decorative.

Where this comparison breaks: a human consultant has intentions of their own. A model has none, but it can be steered by any text that reaches its context, including text inside the very documents it is reading for you. So the note under the glass may have been dictated by a stranger.

51.12.3 PLAIN - a worked example

1. A complete exchange, in four steps. First you declare the tool.

```
"tools": [{
  "type": "function",
  "function": {
    "name": "get_order_status",
    "description": "Look up the status of one order.",
    "parameters": {
      "type": "object",
      "properties": {
        "order_id": {"type": "string",
                     "description": "Order number"}
      },
      "required": ["order_id"]
    }
  }
}]
```

2. Second, the user asks "where is order 4471", and the model replies with a request rather than an answer:

```
"message": {
  "role": "assistant",
  "content": null,
  "tool_calls": [{
    "id": "call_a1",
    "type": "function",
    "function": {
      "name": "get_order_status",
      "arguments": "{\"order_id\": \"4471\"}"
    }
  }]
},
"finish_reason": "tool_calls"
```

3. Third, your code validates the arguments, runs the real lookup, and sends the whole conversation back with one extra message:

```
{"role": "tool",
 "tool_call_id": "call_a1",
 "content": "{\"status\":\"shipped\",
              \"eta\":\"2026-08-15\"}"}
```

4. Fourth, the model produces the final answer: "Order 4471 shipped and should arrive on 15 August 2026."

51.12.4 PLAIN – what is really happening inside

1. The tool declarations are inserted into the model's context as text, usually as a schema in the system section. They cost tokens on every call.
2. The model was trained to emit a particular token pattern when it wants a tool. The serving layer detects that pattern and returns it as structured JSON instead of prose.
3. Many providers use constrained decoding here, so the arguments are guaranteed to match the schema. Others do not, and you must handle malformed JSON.
4. The model chooses the tool by matching your description text against the user's request. So the description is a prompt, and vague descriptions cause wrong tool choices.
5. Now the security part, stated plainly. The model decides which tool to call based on text in its context. Some of that text came from your user. Some may have come from a document, a web page, or an email.
6. Therefore any text the model can read can attempt to trigger any tool the model can call. There is no separation between instructions and data.
7. So: validate every argument in your own code as if it came from an anonymous internet user, because effectively it did. Enforce permissions on the caller, not on the model.
8. Never pass model output into a shell, a SQL string or a file path without the same checks you would use on a web form.
9. Prefer narrow tools over general ones. `get_order_status(order_id)` is safe to expose. `run_sql(query)` is not, however convenient it looks.

51.12.5 TECHNICAL – the engineer's version

1. History: OpenAI shipped function calling in June 2023. The research precursor was Toolformer, published in February 2023. Anthropic released the Model Context Protocol in November 2024 to standardize how tools are described and discovered; OpenAI adopted it in March 2025, and its specification is dated by release, with 2026-07-28 current in August 2026.

2. Cost note: every tool round trip re-sends the entire conversation. A five-tool-call task sends the context six times. This is the main reason agent workloads are expensive, and the main reason prompt caching matters.
3. Threat model, in the vocabulary that has settled: the dangerous combination is private data access, exposure to untrusted content, and an ability to communicate outward. Simon Willison named this the "lethal trifecta" in June

2025. Remove any one of the three and exfiltration becomes much harder.

4. Practical controls: allowlist tools per session, require human confirmation for writes, run tools with least privilege, log every call with arguments, and put outbound network egress behind a proxy you control.

51.12.6 WORDS - remember these

Function calling — the model asks, you do it — structured tool-request output matching a declared JSON schema. Tool result — the answer handed back — a message with role "tool" carrying the execution output into the context. Constrained decoding — only valid output can be produced — logit masking against a grammar or schema. MCP — a common plug for tools — Model Context Protocol, from Anthropic, November 2024, adopted broadly in 2025 and 2026. Lethal trifecta — the dangerous combination — private data, untrusted content and outbound communication in one agent.

51.13 Agents: what the word actually means

51.13.1 PLAIN - in simple words

1. An agent is a loop, not a product and not a personality.
2. The loop is: the model decides what to do next, calls a tool, sees the result, and decides again, until it says it is finished or you stop it.
3. A single API call is one question and one answer. An agent is many calls in a row, where the model chooses what the next call contains.
4. RAG is different again: retrieval happens once, before the model runs, and the model does not choose it.
5. So the ladder is: one call, then one call with tools, then a loop with tools, which is an agent.
6. The problem is arithmetic. Steps multiply. If each step is 95 percent reliable, twenty steps together are 36 percent reliable.
7. That is not a detail to be tuned away. It is the central engineering fact about agents.

51.13.2 PLAIN - a picture in your head

1. Think of a relay race where each runner hands over a written message rather than a baton.
2. Each runner is fast and mostly accurate, but occasionally mishears a word.
3. Over forty, it arrives as something else entirely, and nobody along the way noticed anything wrong.
4. Now add this: every runner is completely confident that what they passed on was correct.
5. The fix is not faster runners. It is checkpoints where the written message is compared against the original.

Where this comparison breaks: a garbled message is obviously garbled at the end. An agent's wrong result usually looks exactly like a right one, because the same fluency that makes the output readable makes the error invisible.

51.13.3 PLAIN - a worked example

1. The compounding arithmetic, spelled out. This is just multiplication.

Per-step success	10 steps	30 steps
90%	34.9%	4.2%
95%	59.9%	21.5%
99%	90.4%	74.0%
99.9%	99.0%	97.0%

2. Read the 95 percent row. A step that fails once in twenty sounds excellent and produces a task that fails four times in five at thirty steps.
3. Now read the 99.9 percent row. To run thirty steps reliably you need per-step reliability that no language model currently offers unaided.
4. So the practical designs all do the same thing: reduce the number of steps, or make each step verifiable so failures are caught rather than compounded.
5. A verifiable step is one where something other than the model can check the result. Code that compiles. Tests that pass. A schema that validates. A number that reconciles.
6. This is exactly why coding agents work better than most other kinds. The compiler and the test suite are an independent judge on every step.

51.13.4 PLAIN - what is really happening inside

1. The common patterns, named, so you recognize them in documentation.
2. ReAct, from a 2022 paper, interleaves reasoning and acting: think, act, observe, repeat. It is the default shape of most agent frameworks.

```
Thought: I need the order status.
Action: get_order_status("4471")
Observation: {"status":"shipped","eta":"2026-08-15"}
Thought: I now have what I need.
Answer: It shipped and arrives on 15 August.
```

3. Plan-and-execute makes a full plan first, then carries out the steps. It is cheaper and more predictable, and worse at recovering when reality differs from the plan.
4. Reflection adds a criticism step: the model reviews its own output against the goal and revises. It helps when there is an objective standard to check against, and mostly produces confident agreement when there is not.
5. Multi-agent designs give different roles to different model calls, such as a planner, a coder and a reviewer, with messages between them.
6. The honest position on multi-agent systems as of 2026: they demonstrate well, they are expensive, and independent evidence that they beat one well-prompted model with good tools on ordinary tasks is thin. Some published analyses argue that errors compound faster across agents, not slower.
7. What actually makes agents work in production, in order of importance: narrow scope, verifiable steps, a hard step limit, a budget limit, retries with different phrasing, and a human checkpoint before anything irreversible.

51.13.5 TECHNICAL - the engineer's version

1. Separate what is established from what is sold.
2. Established: tool-calling loops work well for bounded tasks with checkable outputs; coding assistants that run tests are genuinely useful; retrieval plus a small number of tool calls is a solid production pattern.
3. Active research: long-horizon planning, self-correction without an external verifier, reliable multi-agent coordination, and memory that survives across sessions in a useful way.
4. Marketing claim: that an agent can be given a business objective and left alone. No published evidence supports this for open-ended tasks, and the compounding arithmetic above argues against it directly.

5. Engineering controls that belong in any agent you deploy:

Control	What it prevents
Max steps and max tokens	runaway loops and bills
Tool allowlist per task	unexpected capability
Idempotency keys	duplicate side effects
Human approval on writes	irreversible mistakes
Full trace logging	undebuggable failures

51.13.6 WORDS - remember these

Agent — a loop where the model chooses the next action — iterative plan-act-observe with tools and a stopping condition. ReAct — think, act, observe, repeat — the interleaved reasoning and acting pattern from a 2022 paper. Reflection — the model criticizing its own draft — a self-review step, useful mainly with an external standard to check against. Compounding error — small failures multiplying — end-to-end success equals per-step reliability raised to the number of steps. Human in the loop — a person approves the risky bit — a mandatory confirmation gate before irreversible actions.

51.14 Memory, and why a model with none appears to have some

51.14.1 PLAIN - in simple words

1. A language model has no memory between calls. None at all.
2. The only state it has is the text currently in its context window.
3. When a chat application seems to remember yesterday, an ordinary database remembered it and pasted it back in.
4. Every turn of a conversation resends the whole conversation so far.
5. That has a cost consequence people miss: turn twenty is not the same price as turn one, because turn twenty carries nineteen turns of history.
6. Everything sold as "memory" is an application feature: store some text, retrieve the relevant bits later, put them in the prompt.
7. So memory is just retrieval, from section 51.11, pointed at your own past conversations instead of your documents.

51.14.2 PLAIN - a picture in your head

1. Imagine a brilliant advisor with complete amnesia every night.
2. Each morning you hand them a folder containing everything relevant from before, and they work perfectly.
3. They are not remembering. You are.
4. If your folder is wrong, they are confidently wrong, and they have no way to notice, because the folder is their entire world.

Where this comparison breaks: the advisor would know they had amnesia. The model has no sense of a gap. Text it was never given simply does not exist for it, and it will fill the space with something plausible rather than reporting the absence.

51.14.3 PLAIN - a worked example

1. A ten-turn conversation with 500 tokens per turn, no caching:

Turn	Input tokens	Cumulative input
1	500	500
5	2,500	7,500
10	5,000	27,500

2. Ten turns of 500 tokens each is 5,000 tokens of conversation, but you paid for 27,500 input tokens, which is five and a half times more.
3. The growth is quadratic in the number of turns. Doubling the turns quadruples the input bill.
4. Three fixes, all standard: cache the stable prefix, summarize old turns into a short note, or drop old turns entirely past a window.

51.14.4 PLAIN - what is really happening inside

1. The common designs, and what each is really doing.
2. Full history: send everything. Simple, exact, and quadratic in cost. Fine for short sessions.
3. Sliding window: keep the last N turns. Cheap and predictable; loses anything older, silently.
4. Running summary: after every few turns, ask the model to compress the older part into a paragraph, and send the summary plus recent turns. Cheap, and lossy in ways you cannot predict.
5. Retrieved memory: store every turn in a vector database and retrieve the few most relevant to the current message. Scales, and inherits every failure mode from section 51.11.
6. Structured facts: extract specific fields, such as the user's name, preferences and account, into an ordinary database, and inject them as text. This is the most reliable and the least fashionable.
7. The limits are the same in every case: whatever is not retrieved does not exist, and nothing about this is the model remembering.

51.14.5 TECHNICAL - the engineer's version

1. Context is a hard bound, not a soft one. Exceeding it produces an error or silent truncation, and truncation usually removes the oldest tokens, which is where your system prompt might be.
2. Put the system prompt where truncation cannot reach it, and validate token counts before sending, using the model's own tokenizer.
3. Structured extraction beats free-text memory for anything you will filter or act on. "The user's plan is enterprise" belongs in a column, not in prose inside a vector store.

51.14.6 WORDS - remember these

Context window — everything the model can see right now — the maximum token sequence, its only state. Conversation history — the transcript resent every turn — the messages array, billed in full on each call. Summarization memory — squash the old part — periodic lossy compression of earlier turns into a short note. Structured memory — facts in a proper database — extracted fields injected as text, the most reliable design.

51.15 The limits, stated plainly

51.15.1 PLAIN - in simple words

1. Hallucination is when a model states something false with the same confidence it states something true.
2. It is not a bug that was left in. It follows from what the model is.
3. The model was trained to produce likely continuations of text. A confident, fluent, plausible answer is a likely continuation whether or not it is true.
4. There is also no internal flag for "I do not know". The model always produces a next token, and the machinery for producing a true one and a false one is identical.

5. Asking it to be accurate does not help, because it is not choosing to be inaccurate.
6. What helps is putting truth in front of it, or letting a tool check.
7. And this is the important sentence: confident wrongness is the default failure mode. Errors do not arrive marked as errors.

51.15.2 PLAIN - a picture in your head

1. Think of an extremely well-read person answering at a dinner party, at speed, who considers not answering to be a failure.
2. Ninety percent of the time they are right and it is delightful.
3. Ten percent of the time they produce the same tone, the same fluency, the same specific detail, and it is invented.
4. You cannot tell the two apart from the outside, and neither can they.
5. Now imagine the party is a legal filing, a medical note or a payment.

Where this comparison breaks: the dinner guest could stop and say "I am not sure". The model can be trained to say that more often, and modern ones do, but it is still producing the phrase because such phrases followed such contexts in training, not because it inspected its own uncertainty and found it high.

51.15.3 PLAIN - a worked example

1. What actually reduces hallucination, and what does not:

Intervention	Effect
Retrieval with quoted source	large reduction
Tool for the checkable part	removes that class
Constrained output schema	removes format errors
Second pass verification	moderate reduction
Saying "be accurate"	negligible
Saying "do not hallucinate"	negligible

2. A concrete pair. Ask for a citation for a claim and you may get a real- looking reference that does not exist, with plausible authors, journal and year.
3. Give the model five real abstracts and require it to quote from one, and the invented citation disappears, because the correct behaviour is now the easy continuation.
4. Now three things the model cannot know, whatever you do.
5. Anything after its training cut-off. It will still answer, using the world as it was.
6. Anything private: your files, your database, this morning's ticket. Unless you put it in the context, it is not there.
7. Its own weights and internals. A model's account of why it answered is a plausible story generated after the fact, not a readout. This is established: self-reported reasoning does not reliably match the computation.

51.15.4 PLAIN - what is really happening inside

1. Why counting and arithmetic fail, connecting to Chapter 47.
2. The model never sees letters. It sees tokens, which are chunks of several characters, and the chunks were chosen by compression statistics.
3. So "strawberry" may arrive as two or three tokens, and the letter r is not individually visible in any of them. Counting them is like counting bricks in a photograph of a wall taken from a mile away.
4. Long arithmetic fails for a second, separate reason: the model does a fixed amount of computation per token and has no scratchpad.

5. Multiplying two six-digit numbers requires carrying intermediate results through many steps. With no working memory, the only place to put them is the output itself.
6. That is exactly why writing out the steps helps, and why handing the sum to a calculator tool helps far more.
7. Now prompt injection, which is the unsolved one. The model has a single channel. Instructions and data arrive as the same tokens.
8. So text inside a document, a web page, an email or a code comment can address the model directly, and the model has no reliable way to tell that this text was not from you.
9. The term was coined by Simon Willison in September 2022. As of August 2026 it remains unsolved in the general case, and security guidance from national agencies now assumes it rather than promising a fix.
10. Jailbreaking is a related but different thing: the user of a model trying to get past its own trained refusals. Injection is a third party attacking your application through content it processes.
11. Sycophancy is the tendency to agree with the user, to soften a correct answer under pushback, and to praise a bad idea. It comes from training on human preference, and humans prefer agreement.

51.15.5 TECHNICAL - the engineer's version

1. Hallucination is best understood as a calibration problem. A well-calibrated model would assign confidence matching its accuracy and abstain below a threshold.
2. A 2025 paper from OpenAI, "Why Language Models Hallucinate", argues the incentive is structural: both pre-training objectives and benchmark scoring reward a guess over an admission of ignorance, exactly as a multiple-choice exam with no penalty for wrong answers does.
3. That implies the fix is partly in evaluation design: score abstention as better than a wrong answer, and models trained against that scoring abstain more. This is active work, not a solved problem.
4. Uncertainty signals you can actually use: token log-probabilities where the provider exposes them, agreement across several samples at nonzero temperature, and agreement between two different models. All are weak but better than nothing.
5. Prompt injection defenses that reduce risk without solving it: strict separation of trusted and untrusted content with explicit markers, least-privilege tools, breaking the lethal trifecta by removing outbound communication or private data access from the same agent, output filtering, and human confirmation on consequential actions.
6. Defenses that do not work: instructing the model to ignore instructions in documents, and detecting injections with another language model, which is itself injectable.
7. What all of this means if you are building:

Rule	Because
Never auto-execute output	injection is unsolved
Verify facts with a source	fluency is not accuracy
Design for wrong answers	they are unmarked
Log inputs and outputs	you will need the trace
Keep a human on writes	reversal is expensive

51.15.6 WORDS - remember these

Hallucination — confident invention — generation of unsupported content, a calibration failure rather than a discrete bug. Calibration — confidence matching accuracy — the property a model would need to know when to abstain. Prompt injection — instructions smuggled in through data — an architectural consequence of one shared channel; unsolved as of August 2026. Jailbreak — getting past the model's own rules — user-side circumvention of trained refusals, distinct from injection. Sycophancy — agreeing

because you pushed — preference-trained bias toward agreement and flattery. Knowledge cut-off — the date the world stopped for it — the end of the training data, after which nothing is known.

51.16 The gaps you did not ask about but need

51.16.1 PLAIN - in simple words

1. Six things you will meet in week one that nobody puts in the introduction.
2. Multimodality: how a picture becomes something a text model can read.
3. Reasoning models: models trained to think at length before answering, and billed for the thinking.
4. Long context: how a model that started with 2,000 tokens now takes a million, and why that is less useful than it sounds.
5. Cards: a model card describes one model; a system card describes a whole deployed product including its safety testing.
6. The safety vocabulary: alignment, red-teaming, refusal, guardrails, evals.
7. Distillation: making a small model imitate a big one.

51.16.2 PLAIN - a picture in your head

1. For images, think of a photograph cut into a grid of small squares.
2. Each square is described as a list of numbers, in the same kind of language the model already uses for words.
3. The squares are then laid in a row alongside the text tokens, and the model reads the whole row as one sequence.

Where this comparison breaks: the translation is learned, not defined, and it is lossy in ways nobody chose. Fine print, precise counts and exact spatial relations survive badly, which is why models misread charts and miscount objects far more often than their fluent descriptions suggest.

51.16.3 PLAIN - a worked example

1. Rough token costs of non-text inputs, which is what you will be billed on:

Input	Becomes roughly
1024 x 1024 image	750 to 1,600 tokens
One page of a PDF	1,000 to 2,000 tokens
One minute of speech	150 words, plus overhead

2. So ten screenshots can cost more than a long document, and a reasoning model may bill 4,000 hidden thinking tokens behind a 200-token answer.

51.16.4 PLAIN - what is really happening inside

1. Images: a vision encoder cuts the picture into patches, embeds each patch, and a small projection layer maps those embeddings into the same space as token embeddings. The transformer then treats them as ordinary positions.
2. Audio: either a spectrogram is patched the same way, or a separate codec turns sound into discrete tokens from a learned audio vocabulary.
3. Reasoning models: trained with reinforcement learning to produce long internal chains of thought before answering. Established fact: this raises accuracy substantially on mathematics, competitive programming and multi-step logic.
4. Active research: how far it generalizes, and the 2026 finding that past a point extra thinking reduces accuracy.

5. Long context is achieved by changing how positions are encoded, most commonly by scaling rotary position embeddings, plus attention variants that avoid materializing a full attention matrix.
6. The honest note: a large advertised context is a capacity, not a competence. Retrieval quality inside a long context degrades with distance from the ends, and simple needle-finding benchmarks overstate real performance on tasks needing several facts combined.
7. Distillation, from a 2015 paper by Hinton and colleagues: train a small student model on the outputs of a large teacher. Most small fast models available today were made this way.

51.16.5 TECHNICAL - the engineer's version

1. Alignment, defined concretely rather than as a slogan: the engineering work of making a model's outputs conform to a written specification of intended behaviour, implemented through preference training and rule-based methods, and measured by evaluations. It is a practice with artifacts, not an aspiration.
2. Red-teaming: adversarial testing by humans or automated attackers, aimed at producing the failures the model is meant not to produce. Findings feed back into training and into guardrails.
3. Refusal: the trained behaviour of declining a request. Both over-refusal and under-refusal are measured, and the trade between them is a product decision, not a technical constant.
4. Guardrails: checks outside the model, on input and output, such as classifiers, regular expressions, schema validation and allowlists. They are the only part of the safety stack that is deterministic.
5. Evals: automated test suites over fixed inputs with scoring, covering both capability and safety. Chapter 50 covered how to build them; the essential discipline is that a private eval set you wrote beats any public leaderboard.
6. System card: a document about a deployed system rather than a bare model, covering intended use, safety evaluations, red-team results, known failure modes and mitigations. The convention was established by the GPT-4 system card in March 2023 and is now normal for frontier releases.
7. Where the field appears to be heading, explicitly labelled speculation and not fact: more inference-time compute rather than only larger models; sparse mixture-of-experts as the default architecture; agentic use as the main product surface; standardized tool protocols; and small models on devices for the majority of routine work.
8. Any of these could be wrong within a year, and this paragraph should be read with its date attached.

51.16.6 WORDS - remember these

Multimodal — text plus images or audio — inputs encoded into the same embedding space and consumed as one sequence. Reasoning model — thinks before it answers — trained to emit long internal chains of thought, billed as output tokens. System card — the document about the product — deployment-level description of intended use, evaluations and mitigations. Alignment — making behaviour match the specification — preference training and rules, measured by evaluations. Guardrail — a check outside the model — deterministic filtering or validation on inputs and outputs. Distillation — a small model taught by a big one — student trained on teacher outputs, from a 2015 paper.

51.17 A practical decision guide

51.17.1 PLAIN - in simple words

1. Most projects reach for the most complicated option first. Work the other way.
2. Ask whether the task needs a language model at all. Regular expressions, database queries and existing libraries are cheaper, faster and exact.
3. If it does, start with one API call and a good prompt.
4. Add retrieval when the answer depends on documents the model has not seen.
5. Add tools when the answer depends on live data or on an action.
6. Add a loop, making it an agent, only when the number of steps genuinely cannot be known in advance.

7. Fine-tune when the format or style must be fixed, or when you want a small model to do one narrow job cheaply.
8. Self-host when policy, volume or control demands it, not because it sounds cheaper.

51.17.2 PLAIN - a picture in your head

1. Think of choosing transport for a journey.
2. A taxi is a hosted API: instant, no maintenance, pay per trip.
3. Bringing a map is retrieval.
4. Hiring a driver who chooses the route as they go is an agent.
5. Buying a fleet is self-hosting, and it only pays if you drive constantly.

Where this comparison breaks: transport choices are reversible in an afternoon. Architecture choices lock in data formats, evaluation sets and vendor contracts, so the cost of starting too complicated is higher than the comparison suggests.

51.17.3 PLAIN - a worked example

```
Does it need judgement about language?
  no  -> write ordinary code. Stop.
  yes -> Is the needed knowledge in the model already?
           yes -> one API call with a good prompt
           no  -> Is it in documents you hold?
                    yes -> RAG
                    no  -> Is it live or an action?
                             yes -> tools
                             no  -> reconsider the task
Is the number of steps known in advance?
  yes -> a fixed chain of calls
  no  -> an agent loop, with step and cost limits
Is the output format or style unstable?
  yes and you have 1,000+ examples -> fine-tune
  yes and you do not               -> few-shot examples
Must data stay inside your network, or is volume
above roughly 3 billion output tokens a month?
  yes -> self-host
  no  -> hosted API
```

51.17.4 PLAIN - what is really happening inside

1. The same guidance as a table, with the cost and latency notes that decide arguments.

Task	Approach	Note
Classify text	small hosted model	cheap, fast, few-shot
Extract fields	model plus schema	constrained decoding
Answer from your docs	RAG	retrieval is the risk
Answer about live data	tools	validate arguments
Summarize a document	one call	watch input cost
Fixed house style	fine-tune	needs 1,000+ examples
Multi-step research	agent	cap steps and budget
Regulated private data	self-host	cost is not the reason
Bulk offline work	batch API	half price, slower

2. Latency notes worth having in mind. A short hosted call is roughly 0.5 to 3 seconds. Adding retrieval adds 50 to 300 milliseconds.

3. Each tool round trip adds a full model call. A reasoning model can take 10 to 60 seconds. An agent is measured in minutes.
4. Cost notes. A simple call is fractions of a cent. RAG multiplies input tokens. An agent multiplies whole calls. Reasoning multiplies output tokens. The multipliers stack.

51.17.5 TECHNICAL - the engineer's version

1. Build the evaluation set before choosing an approach. Thirty to two hundred real examples with known good answers, held privately, decides every question in this section empirically rather than by argument.
2. Choose the cheapest model that passes your evaluation, then move up only if it fails. Most production traffic does not need a frontier model.
3. Design for model substitution from the first day: keep prompts in one place, keep an adapter layer, and re-run the evaluation set when you change models, because a provider changing an undated model name is a real event.
4. Route by difficulty. A small model handles the common case and escalates on low confidence or explicit triggers. This routinely cuts costs by an order of magnitude while leaving quality on hard cases intact.
5. Instrument tokens per request, cost per task, cache hit rate, p50 and p99 latency, and end-to-end task success. The last one is the only one your users experience.

51.17.6 WORDS - remember these

Model routing — send easy work to the cheap model — confidence or rule-based dispatch across a tier of models, with escalation. Escalation — try harder when the cheap answer looks weak — a second call to a stronger model triggered by a confidence signal. Eval set — your own private exam for the system — a fixed collection of real inputs with expected outputs, versioned with the code. Golden path — the case you optimize for — the dominant traffic pattern that should decide model and architecture choice.

51.98 Common wrong ideas

1. Wrong: it looked that up when I asked. Right: unless a retrieval or search tool ran, nothing was looked up. The answer came from frozen weights.
2. Wrong: it remembers me. Right: it has no memory between calls. An application stored your text and pasted it back into the context.
3. Wrong: hallucination will be fixed in the next version. Right: it is a property of predicting likely text without verification. Versions reduce the rate; retrieval, tools and checking reduce it much more.
4. Wrong: the system prompt is secure and hidden. Right: it is ordinary tokens at the front of the context, obeyed by training habit, and it can be overridden or revealed by clever input.
5. Wrong: an agent can be trusted to finish the job. Right: per-step errors compound, so a 95 percent reliable step is 21 percent reliable over thirty steps. Cap steps, verify each one, keep a human on writes.
6. Wrong: self-hosting is cheaper. Right: you rent the machine whether you use it or not. Break-even against cheap hosted models is around billions of output tokens a month.
7. Wrong: a bigger context window means it uses all of it well. Right: capacity is not competence. Attention to the middle of a long context is measurably weaker than to the ends.
8. Wrong: more parameters always means faster answers if the hardware is good. Right: decode speed is memory bandwidth divided by model bytes. A bigger model is slower on the same machine, in direct proportion.
9. Wrong: prompt engineering is the main skill. Right: it is real but shallow. Retrieval quality, task decomposition, evaluation and tool design matter more.
10. Wrong: the model runs my code when it calls a function. Right: it emits a structured request; your program decides whether to run it. Every tool is a security boundary you own.

51.99 Chapter summary in 20 lines

1. Inference is the frozen model answering; training is what produced it, and it needs roughly ten times the memory.
2. An API call is ordinary HTTPS carrying a list of messages plus sampling settings, and the whole conversation is resent every time.
3. The system prompt is only leading tokens. Its authority is a training habit, not a protection.
4. Generation has two phases: prefill, which is parallel and compute-bound, and decode, which is sequential and bandwidth-bound.
5. Decode speed is approximately memory bandwidth divided by model bytes, which explains almost every speed question people ask.
6. Time to first token is network plus queue plus prefill; tokens per second afterwards is a different limit entirely.
7. The KV cache turns quadratic recomputation into linear work and costs 128 KiB per token for an 8B model at 16-bit.
8. Continuous batching, PagedAttention, speculative decoding, quantization, tensor parallelism and prefix caching are the serving toolkit.
9. The same model runs at different speeds at different providers because of hardware, quantization, batching policy, load and distance.
10. Output tokens cost five to six times input tokens because decode is sequential and prefill is parallel.
11. Self-hosting is rarely cheaper below very high sustained volume; the good reasons are privacy, control and stability.
12. llama.cpp is the engine, Ollama and LM Studio the friendly wrappers, vLLM and SGLang the serving engines, MLX the Apple silicon path.
13. Memory decides whether a model runs; bandwidth decides how fast. Apple's unified memory wins on capacity and loses on bandwidth and prefill.
14. Read the model card for base model, data, licence, limitations and evaluations, and default to Q4_K_M unless you have a reason.
15. Training a tiny model on a laptop teaches shape without knowledge; do it once as education, then fine-tune or use an open model instead.
16. RAG is chunk, embed, store, retrieve, inject, generate; chunking and retrieval quality dominate the outcome.
17. Tool calling means the model emits a request and your code executes it, so every tool is a security boundary.
18. An agent is a loop, and per-step errors compound multiplicatively, so narrow scope, verifiable steps and human checkpoints are what make it work.
19. Hallucination is intrinsic to predicting likely text without verification; retrieval, tools and constrained output reduce it, exhortation does not.
20. Prompt injection remains unsolved as of August 2026 because instructions and data share one channel, so build as though confident wrong output will happen, because it will.

KB

PART I

The Reference Desk

The numbers, commands, decision trees, timeline and glossary you will come back to.

52

CHAPTER

KB

The Engineer's Reference

52.0 How to use this chapter

1. Every other chapter in this book teaches. This one does not teach. It answers.
2. It is the chapter you open when you are already in the middle of something and you need one number, one flag, or one next step.
3. Nothing here is new. Every fact appeared earlier in the book, in context, with the explanation attached. Here it is stripped down to the lookup.
4. The chapter is arranged so you can find things without reading. The section numbers below are the map.
5. Sections 52.1 to 52.3 are numbers: latency, powers of two, units, bases, character codes. Use these when you are estimating.
6. Section 52.4 is the networking reference: layers, addresses, ports, status codes, record types, states. Use this when you are reading a capture or a log.
7. Section 52.5 is the networking command cheat sheet. Every command has two extra columns that most cheat sheets leave out: what a pass **proves**, and what a pass does **not** prove. That distinction is the whole diagnostic method of this book.
8. Section 52.6 is a decision tree for "I cannot reach a site". It is written as a branching procedure. Follow the numbers.
9. Sections 52.7 and 52.8 are the same two things for git: a full command cheat sheet, then a decision tree for a failed git operation.
10. Sections 52.9 to 52.16 are the rest of the desk reference: shell, file magic numbers, complexity, cloud, security, AI, regular expressions, editors.
11. Section 52.17 is what to do when none of the above worked.
12. Two conventions run through the chapter. First, macOS syntax is given first, because that is what the reader of this book is using, with the Linux difference noted on the next line. Second, anything that is a convention rather than a standard is labelled as such.
13. Anything marked "approximate" is an order of magnitude, not a measurement. Order of magnitude is usually what you need. A number that is right to within a factor of two will stop you making the wrong decision.
14. One warning before you start. A reference table tells you what a thing is. It does not tell you whether the thing is your problem. Keep the habit of asking what an observation rules out, not just what it looks like.

52.1 Numbers every engineer should know

52.1.1 Why this section exists

1. Most engineering decisions are decided by a factor of a thousand, not by a factor of two.
2. If you know that memory is roughly one hundred times slower than cache, and a network hop across the planet is roughly a million times slower than memory, you will make the right architectural call without measuring.
3. The list below descends from fastest to slowest. Learn the shape, not the digits.
4. The original list of this kind is credited to Jeff Dean of Google, who presented "Numbers Everyone Should Know" in a 2009 keynote. Peter Norvig published a similar table earlier, in his 2001 essay "Teach Yourself Programming in Ten Years".
5. The figures below are updated for typical 2025 to 2026 hardware. They are approximate.

52.1.2 The latency ladder, real figures

Operation	Typical time	Note
1 CPU cycle at 3 GHz	0.33 ns	one tick
L1 cache reference	1 ns	4 to 5 cycles
Branch mispredict	3 to 5 ns	pipeline refill
L2 cache reference	4 ns	about 14 cycles
L3 cache reference	15 to 40 ns	shared, slower
Mutex lock and unlock	17 ns	uncontended
Main memory reference	80 to 100 ns	DDR4 or DDR5
Read 1 MB from RAM	50 us	one core, 20 GB/s
NVMe SSD 4 KiB read	20 to 100 us	random
Read 1 MB from NVMe	150 to 300 us	sequential
Same-rack round trip	0.1 ms	switch to switch
Same-datacentre round trip	0.5 ms	500 us
HDD seek, 7200 rpm	8 ms	seek plus rotate
Read 1 MB from HDD	5 ms	sequential
Same-city round trip	5 to 10 ms	metro fibre
Same-continent round trip	20 to 50 ms	
Mumbai to Frankfurt	110 to 130 ms	approximate
Mumbai to Virginia	190 to 220 ms	approximate
Cross-continent worst case	250 to 300 ms	
Geostationary satellite	480 to 600 ms	35,786 km up
Low-earth-orbit satellite	25 to 60 ms	about 550 km up

1. The single most useful line in that table is the second-from-last group: light in glass fibre travels at about 200,000 kilometres per second, which is roughly two thirds of the speed of light in vacuum.
2. That gives a rule you can do in your head: about **1 millisecond of round trip per 100 kilometres of fibre**.
3. Check it. Mumbai to London is about 7,200 km in a straight line and about 10,000 km of real cable route. Ten thousand divided by one hundred is 100 milliseconds. Measured round trips on that path are 110 to 130 ms.
4. This rule is physics. No amount of money buys past it. It is why content delivery networks exist: you cannot make the light faster, so you move the copy closer.

52.1.3 The latency ladder, scaled to human time

1. Multiply every figure by one billion. One nanosecond becomes one second.
2. Now the ladder becomes something a person can feel.

Operation	Real	Human scale
L1 cache reference	1 ns	1 second
L2 cache reference	4 ns	4 seconds
L3 cache reference	15 ns	15 seconds
Main memory reference	100 ns	1 minute 40 seconds
NVMe SSD read, fast	20 us	5.6 hours
NVMe SSD read, typical	100 us	1.2 days
Same-datacentre round trip	0.5 ms	5.8 days
Read 1 MB from HDD	5 ms	2 months
HDD seek	8 ms	3 months
Mumbai to Frankfurt	120 ms	3.8 years
Mumbai to Virginia	200 ms	6.3 years
Cross-continent worst case	300 ms	9.5 years
Machine reboot	30 s	950 years
The reader's curl timeout	15 s	475 years

3. Read the last two lines again. When the reader's `curl` waited fifteen seconds and gave up, the processor had, in its own sense of time, waited several human centuries for an answer that never came.
4. That is why a timeout is such strong evidence. Nothing on a working network takes that long. Silence for fifteen seconds is not slowness. It is absence.

52.1.4 Powers of two, 2^0 to 2^64

1. Every size in computing is a power of two or a multiple of one.
2. Learn 2^10 = 1024, 2^20 = about one million, 2^30 = about one billion. The rest follows.

Power	Value	As a byte size
2^0	1	-
2^1	2	-
2^2	4	-
2^3	8	one byte in bits
2^4	16	one hex digit range
2^5	32	-
2^6	64	-
2^7	128	ASCII range
2^8	256	one byte of values
2^9	512	classic disk sector
2^10	1024	1 KiB
2^11	2048	2 KiB
2^12	4096	4 KiB, one page
2^13	8192	8 KiB
2^14	16384	16 KiB
2^15	32768	32 KiB, int16 max+1
2^16	65536	64 KiB, port count
2^17	131072	128 KiB
2^18	262144	256 KiB
2^19	524288	512 KiB
2^20	1048576	1 MiB
2^21	2097152	2 MiB
2^22	4194304	4 MiB

Power	Value	As a byte size
2^23	8388608	8 MiB
2^24	16777216	16 MiB, colours
2^25	33554432	32 MiB
2^26	67108864	64 MiB
2^27	134217728	128 MiB
2^28	268435456	256 MiB
2^29	536870912	512 MiB
2^30	1073741824	1 GiB
2^31	2147483648	2 GiB, int32 max+1
2^32	4294967296	4 GiB, IPv4 space
2^33	8589934592	8 GiB
2^34	17179869184	16 GiB
2^35	34359738368	32 GiB
2^36	68719476736	64 GiB
2^37	137438953472	128 GiB
2^38	274877906944	256 GiB
2^39	549755813888	512 GiB
2^40	1099511627776	1 TiB
2^41	2199023255552	2 TiB
2^42	4398046511104	4 TiB
2^43	8796093022208	8 TiB
2^44	17592186044416	16 TiB
2^45	35184372088832	32 TiB
2^46	70368744177664	64 TiB
2^47	140737488355328	128 TiB, x86 user VA
2^48	281474976710656	256 TiB, MAC space
2^49	562949953421312	512 TiB
2^50	1125899906842624	1 PiB
2^51	2251799813685248	2 PiB
2^52	4503599627370496	4 PiB, double mantissa
2^53	9007199254740992	8 PiB, exact int limit
2^54	18014398509481984	16 PiB
2^55	36028797018963968	32 PiB
2^56	72057594037927936	64 PiB
2^57	144115188075855872	128 PiB
2^58	288230376151711744	256 PiB
2^59	576460752303423488	512 PiB
2^60	1152921504606846976	1 EiB
2^61	2305843009213693952	2 EiB
2^62	4611686018427387904	4 EiB
2^63	9223372036854775808	8 EiB, int64 max+1
2^64	18446744073709551616	16 EiB, address space

3. Three lines in that table cause more real bugs than the rest combined.
4. 2^31 minus 1 is 2147483647, the largest signed 32-bit integer. This is the Year 2038 problem: a 32-bit Unix timestamp counting seconds from 1 January 1970 overflows at 03:14:07 UTC on 19 January 2038.
5. 2^53 is 9007199254740992. A double-precision floating point number, the only number type JavaScript had until BigInt was standardized in ES2020, can represent every integer up to 2^53 exactly and not one above it. Git commit hashes and large database IDs sent as JSON numbers are silently corrupted

here.

6. 2^32 is 4294967296, the total number of IPv4 addresses. That is fewer than the number of humans alive. This is why NAT exists.

52.1.5 Decimal versus binary prefixes

1. There are two systems and they disagree by a growing amount.
2. Decimal prefixes are powers of a thousand. Binary prefixes are powers of 1024. The binary ones were standardized by the International Electrotechnical Commission in IEC 60027-2, amendment 2, in January 1999, and now live in ISO/IEC 80000-13:2008.

Decimal	Value	Binary	Gap
kB	10^3	KiB = 2^10	2.4%
MB	10^6	MiB = 2^20	4.9%
GB	10^9	GiB = 2^30	7.4%
TB	10^12	TiB = 2^40	10.0%
PB	10^15	PiB = 2^50	12.6%
EB	10^18	EiB = 2^60	15.3%
ZB	10^21	ZiB = 2^70	18.1%
YB	10^24	YiB = 2^80	20.9%

3. Who uses which, in practice, is a convention and not a standard.
4. Disk and SSD makers use decimal. A "1 TB" drive holds 10^12 bytes.
5. Operating systems differ. Windows reports that same drive as 931 GB, because it divides by 2^30 but writes the label "GB". macOS since version 10.6, released in 2009, uses decimal, so it agrees with the box and says 1 TB. Linux tools vary; `df -h` uses 1024 by default, `df -H` uses 1000.
6. RAM is always binary. A "16 GB" memory module is 16 GiB, because memory is addressed by binary address lines and cannot be anything else.
7. Network speeds are always decimal, and always in bits. "1 Gbit/s" is 1,000,000,000 bits per second.
8. In November 2022 the 27th General Conference on Weights and Measures added four new SI prefixes: ronna for 10^27, quetta for 10^30, ronto for 10^-27 and quecto for 10^-30. There are no matching binary prefixes.

52.1.6 Common bandwidth figures

Link	Rate	Year introduced
10BASE-T Ethernet	10 Mbit/s	1990
100BASE-TX	100 Mbit/s	1995
1000BASE-T	1 Gbit/s	1999
10GBASE-T	10 Gbit/s	2006
400G Ethernet	400 Gbit/s	2017
800G Ethernet	800 Gbit/s	2024
Wi-Fi 802.11b	11 Mbit/s	1999
Wi-Fi 802.11g	54 Mbit/s	2003
Wi-Fi 4, 802.11n	up to 600 Mbit/s	2009
Wi-Fi 5, 802.11ac	up to 6.9 Gbit/s	2013
Wi-Fi 6, 802.11ax	up to 9.6 Gbit/s	2021
Wi-Fi 7, 802.11be	up to 46 Gbit/s	2024
USB 2.0	480 Mbit/s	2000
USB 3.0	5 Gbit/s	2008

Link	Rate	Year introduced
USB4	40 Gbit/s	2019
USB4 version 2	80 Gbit/s	2022
Thunderbolt 3 and 4	40 Gbit/s	2015, 2020
Thunderbolt 5	80 to 120 Gbit/s	2023
SATA III	6 Gbit/s	2009
PCIe 3.0 x16	15.75 GB/s	2010
PCIe 4.0 x16	31.5 GB/s	2017
PCIe 5.0 x16	63 GB/s	2019
DDR4-3200, one channel	25.6 GB/s	2014
DDR5-6400, one channel	51.2 GB/s	2020
HBM3 per stack	819 GB/s	2022
HBM3E per stack	about 1.2 TB/s	2024

1. Note the unit change halfway down. Network and bus speeds are quoted in bits per second. Memory and PCIe totals are usually quoted in bytes per second. Mixing them up gives you an eight-fold error.

52.1.7 Typical throughput of real systems

1. These are order-of-magnitude figures for a single well-configured node in 2025 to 2026. Your numbers will differ. The shape will not.

System	Typical throughput	Note
Redis, single core	100k ops/s	simple GET and SET
Redis, pipelined	1M ops/s	batched requests
PostgreSQL, simple reads	10k to 50k queries/s	indexed, cached
nginx, static file	50k to 500k req/s	small files
Kafka, one broker	100 MB/s and up	sequential writes
HDD, 7200 rpm	100 to 200 IOPS	random 4 KiB
SATA SSD	up to 90k IOPS	random 4 KiB
NVMe SSD	up to 1M IOPS	random 4 KiB
gzip compress	30 to 100 MB/s	one core
zstd compress, level 3	300 to 500 MB/s	one core
SHA-256 with hardware	1 to 2 GB/s	one core
AES-256-GCM with AES-NI	2 to 10 GB/s	one core
JSON parse	100 to 500 MB/s	one core

2. One more figure that catches people out. A single TCP connection is limited by its window divided by the round-trip time. With a 64 KiB window and a 100 ms round trip, that is 64 KiB divided by 0.1 seconds, which is about 655 kB/s, which is about 5.2 Mbit/s.
3. That is the bandwidth-delay product limit. It is why a single download from a distant server can crawl on a gigabit line, and why parallel connections or a larger window fix it. Modern systems scale the window automatically, up to 1 GiB, using the window scale option of RFC 7323.

52.2 Units and conversions

52.2.1 Bit and byte

1. A **bit** is one binary digit. It is either 0 or 1.
2. A **byte** is eight bits. This has been near-universal since the IBM System/360 in 1964. Older machines used 6, 7, 9 and 36-bit units.

3. The correct formal word for exactly eight bits is **octet**. Networking standards use it, because "byte" once meant other sizes. RFC documents say octet.
4. Lowercase b means bit. Uppercase B means byte. This is a convention, but a very strongly held one.
5. A **nibble** is four bits, which is exactly one hexadecimal digit.
6. A **word** has no fixed size. It means the natural unit of the processor: 16 bits on an 8086, 32 on an i386, 64 on modern x86-64 and ARM64.

52.2.2 The full prefix table

Prefix	Symbol	Factor
quetta	Q	10^30
ronna	R	10^27
yotta	Y	10^24
zetta	Z	10^21
exa	E	10^18
peta	P	10^15
tera	T	10^12
giga	G	10^9
mega	M	10^6
kilo	k	10^3
milli	m	10^-3
micro	u	10^-6
nano	n	10^-9
pico	p	10^-12
femto	f	10^-15
atto	a	10^-18
zepto	z	10^-21
yocto	y	10^-24
ronto	r	10^-27
quecto	q	10^-30

1. Note that kilo is lowercase k in SI. Uppercase K is not an SI prefix; it is the symbol for kelvin. Everyone writes KB anyway. That is a convention that contradicts the standard.
2. The micro symbol is properly the Greek letter mu. In plain text everyone writes u. This book writes us for microseconds.

52.2.3 MB/s versus Mb/s

1. To convert bits per second to bytes per second, divide by 8.
2. To convert bytes per second to bits per second, multiply by 8.
3. Some people add ten per cent for framing overhead when talking about Ethernet. That is a rough allowance, not a standard.

Line rate	Theoretical max	Realistic
10 Mbit/s	1.25 MB/s	1.1 MB/s
100 Mbit/s	12.5 MB/s	11 MB/s
1 Gbit/s	125 MB/s	110 to 118 MB/s
2.5 Gbit/s	312.5 MB/s	290 MB/s
10 Gbit/s	1.25 GB/s	1.15 GB/s
40 Gbit/s	5 GB/s	4.6 GB/s

Line rate	Theoretical max	Realistic
100 Gbit/s	12.5 GB/s	11.5 GB/s

4. Worked example. Your broadband plan says 300 Mbit/s. Divide by 8 to get 37.5 MB/s. A 4 GB file, at best, takes 4000 divided by 37.5, which is about 107 seconds. If your download manager shows 5 MB/s, you are getting about 40 Mbit/s, which is roughly one seventh of the plan.

52.2.4 Hertz to time period

1. Frequency and period are reciprocals. Period equals one divided by frequency.

Frequency	Period	Where you see it
1 Hz	1 s	a clock tick
50 Hz	20 ms	mains power, India
60 Hz	16.67 ms	mains power, USA
60 Hz refresh	16.67 ms	standard display
120 Hz refresh	8.33 ms	fast display
1 kHz	1 ms	audio low end
44.1 kHz	22.7 us	CD sample rate
1 MHz	1 us	early CPUs
1 GHz	1 ns	modern CPU order
3 GHz	0.333 ns	one CPU cycle
2.4 GHz	0.417 ns	Wi-Fi band centre
5 GHz	0.2 ns	Wi-Fi band centre

2. Wavelength equals the speed of light divided by frequency. At 2.4 GHz that is 300,000,000 divided by 2,400,000,000, which is 12.5 centimetres. That is why a 2.4 GHz antenna is a few centimetres long.

52.2.5 dBm to milliwatts

1. dBm is power measured in decibels relative to one milliwatt. It is a logarithmic scale, so it turns multiplication into addition.
2. The formula is: power in dBm equals 10 times the base-10 logarithm of the power in milliwatts.
3. Two shortcuts do almost all the work. Adding 3 dB doubles the power. Adding 10 dB multiplies it by ten.

dBm	Power	Meaning for Wi-Fi
+30 dBm	1000 mW = 1 W	legal max in some bands
+20 dBm	100 mW	typical router transmit
0 dBm	1 mW	reference point
-30 dBm	0.001 mW	excellent signal
-50 dBm	very small	very good
-60 dBm	very small	good
-67 dBm	very small	minimum for video
-70 dBm	very small	usable, weak
-80 dBm	very small	unreliable
-90 dBm	very small	unusable, near noise

4. Signal strength in Wi-Fi is always a negative dBm number, because received power is always far below one milliwatt. Closer to zero is stronger. Minus 50 is much better than minus 80.
5. A drop of 3 dBm means you are receiving half the power. A drop of 10 dBm means one tenth.

52.2.6 Temperature

1. To convert Celsius to Fahrenheit: multiply by 9, divide by 5, add 32.
2. To convert Fahrenheit to Celsius: subtract 32, multiply by 5, divide by 9.
3. To convert Celsius to Kelvin: add 273.15.

Celsius	Fahrenheit	What it is
-273.15	-459.67	absolute zero
0	32	water freezes
20	68	comfortable room
27	80.6	warm room, India
35	95	hot ambient
40	104	idle CPU, normal
70	158	CPU under load
90	194	CPU throttling starts
100	212	water boils, CPU limit
105	221	typical shutdown point

4. Silicon does not fail at 90 degrees Celsius. It slows itself down on purpose, which is called thermal throttling, to avoid getting hotter. A laptop that suddenly gets slow after ten minutes of heavy work is almost always throttling.

52.2.7 Storage sizes and what fits

Size	What fits
1 byte	one ASCII character
100 bytes	a short sentence
1 KiB	half a page of text
10 KiB	a small web page's HTML
100 KiB	a compressed photo, low
1 MiB	a typical JPEG photo
5 MiB	a 4-minute MP3 at 160 kbit/s
50 MiB	a phone photo burst
700 MiB	one CD-ROM
1 GiB	250 songs, or 1 hour SD video
4.7 GB	one single-layer DVD
8 GiB	a small language model, 4-bit
25 GB	one Blu-ray layer
50 GB	a large modern game
256 GB	a typical phone
1 TB	250,000 photos
4 TB	a big desktop drive
20 TB	one large hard disk, 2026
1 PB	a mid-size company's archive
1 EB	global scale, a few of these

1. Two facts to keep the scale honest. The English Wikipedia text dump, without images and without history, is roughly 25 GB uncompressed and about 22 GB compressed as of 2025. The complete works of Shakespeare in plain text are about 5.5 MB.

52.3 Number bases and encoding

52.3.1 Binary, decimal, hexadecimal and octal, 0 to 32

1. Hexadecimal is base 16. It uses 0 to 9 then A to F.
2. One hex digit is exactly four bits. Two hex digits are exactly one byte. That is the entire reason hex exists in computing.
3. Octal is base 8, three bits per digit. It survives in Unix file permissions and in C escape sequences.

Dec	Binary	Hex	Octal
0	000000	00	0
1	000001	01	1
2	000010	02	2
3	000011	03	3
4	000100	04	4
5	000101	05	5
6	000110	06	6
7	000111	07	7
8	001000	08	10
9	001001	09	11
10	001010	0A	12
11	001011	0B	13
12	001100	0C	14
13	001101	0D	15
14	001110	0E	16
15	001111	0F	17
16	010000	10	20
17	010001	11	21
18	010010	12	22
19	010011	13	23
20	010100	14	24
21	010101	15	25
22	010110	16	26
23	010111	17	27
24	011000	18	30
25	011001	19	31
26	011010	1A	32
27	011011	1B	33
28	011100	1C	34
29	011101	1D	35
30	011110	1E	36
31	011111	1F	37
32	100000	20	40

52.3.2 The powers of two in hexadecimal

1. In hex, every power of two is a 1, 2, 4 or 8 followed by zeros. This makes sizes instantly recognizable in a hex dump.

Power	Decimal	Hex
2^8	256	0x100
2^10	1024	0x400
2^12	4096	0x1000
2^16	65536	0x10000
2^20	1048576	0x100000
2^24	16777216	0x1000000
2^30	1073741824	0x40000000
2^31	2147483648	0x80000000
2^32	4294967296	0x100000000
2^40	1 TiB	0x10000000000
2^48	256 TiB	0x1000000000000
2^64	16 EiB	0x10000000000000000

2. When you see `0xFFFFFFFF` in a log, that is 4294967295, which is the largest unsigned 32-bit value, which usually means "all bits set" or "not found".
3. When you see `0xDEADBEEF` or `0xCAFEBABE`, that is a human-chosen marker value, put there so it stands out in memory. `0xCAFEBABE` is also the real first four bytes of a Java class file, chosen by James Gosling's team in 1991 for exactly that reason.

52.3.3 The printable ASCII table

1. ASCII was standardized in 1963 by the American Standards Association and revised in 1967 and 1986. It defines 128 codes, 0 to 127.
2. Codes 0 to 31 and 127 are control codes. Codes 32 to 126 are printable.

```
Dec  Hx Ch  Dec  Hx Ch  Dec  Hx Ch  Dec  Hx Ch
 32  20 SP   56  38 8    80  50 P   104  68 h
 33  21 !    57  39 9    81  51 Q   105  69 i
 34  22 "    58  3A :    82  52 R   106  6A j
 35  23 #    59  3B ;    83  53 S   107  6B k
 36  24 $    60  3C <    84  54 T   108  6C l
 37  25 %    61  3D =    85  55 U   109  6D m
 38  26 &    62  3E >    86  56 V   110  6E n
 39  27 '    63  3F ?    87  57 W   111  6F o
 40  28 (    64  40 @    88  58 X   112  70 p
 41  29 )    65  41 A    89  59 Y   113  71 q
 42  2A *    66  42 B    90  5A Z   114  72 r
 43  2B +    67  43 C    91  5B [   115  73 s
 44  2C ,    68  44 D    92  5C \   116  74 t
 45  2D -    69  45 E    93  5D ]   117  75 u
 46  2E .    70  46 F    94  5E ^   118  76 v
 47  2F /    71  47 G    95  5F _   119  77 w
 48  30 0    72  48 H    96  60 `   120  78 x
 49  31 1    73  49 I    97  61 a   121  79 y
 50  32 2    74  4A J    98  62 b   122  7A z
 51  33 3    75  4B K    99  63 c   123  7B {
 52  34 4    76  4C L   100  64 d   124  7C |
 53  35 5    77  4D M   101  65 e   125  7D }
 54  36 6    78  4E N   102  66 f   126  7E ~
 55  37 7    79  4F O   103  67 g
```

3. Four relationships in that table are worth memorizing, because they let you do character arithmetic in your head.
4. The digit `0` is 48, which is hex 0x30. So the numeric value of a digit character is the character minus 48.

5. Uppercase `A` is 65, hex 0x41. Lowercase `a` is 97, hex 0x61.
6. The gap between an uppercase letter and its lowercase form is exactly 32, which is a single bit: bit 5, value 0x20. Flipping that one bit changes case. That is not an accident; the 1963 committee arranged it deliberately.
7. Space is 32, hex 0x20. Delete is 127, hex 0x7F.

52.3.4 Control codes and escape sequences

Escape	Dec	Hex	Meaning
`\0`	0	00	null, string end in C
`\a`	7	07	bell, terminal beep
`\b`	8	08	backspace
`\t`	9	09	horizontal tab
`\n`	10	0A	line feed, newline
`\v`	11	0B	vertical tab
`\f`	12	0C	form feed, new page
`\r`	13	0D	carriage return
`\e`	27	1B	escape, starts ANSI
`\\`	92	5C	a literal backslash
`\"`	34	22	a literal double quote
`\'`	39	27	a literal single quote
`\x41`	65	41	hex escape, gives A
`\101`	65	41	octal escape, gives A
`\u00E9`	233	E9	Unicode escape, e-acute

1. The line ending difference is the single most common cause of a file looking corrupted when moved between systems.
2. Unix and macOS since version 10 end a line with LF alone, which is 0x0A.
3. Windows ends a line with CR then LF, which is 0x0D 0x0A.
4. Classic Mac OS, up to version 9 in 2001, used CR alone, 0x0D.
5. In a hex dump, `0D 0A` at the end of each line means the file came from Windows. In git, the setting `core.autocrlf` controls whether git converts between the two.

52.3.5 UTF-8 byte-length rules

1. UTF-8 was designed by Ken Thompson and Rob Pike in September 1992, famously sketched on a placemat in a diner in New Jersey. It is defined today by RFC 3629, from 2003.
2. It encodes every Unicode code point in one to four bytes, and it is backward compatible with ASCII: any ASCII file is already valid UTF-8.

Code point range	Bytes	First byte pattern
U+0000 to U+007F	1	0xxxxxxx
U+0080 to U+07FF	2	110xxxxx
U+0800 to U+FFFF	3	1110xxxx
U+10000 to U+10FFFF	4	11110xxx

3. Every byte after the first in a multi-byte sequence has the pattern `10xxxxxx`. That is the self-synchronizing property: from any byte you can find the start of the character by scanning backwards past bytes beginning with `10`.
4. A byte beginning with `0` is a lone ASCII character. A byte beginning with `11` starts a multi-byte character. A byte beginning with `10` is a continuation.

5. Practical consequences you will meet.
6. English text costs about 1 byte per character. Latin letters with accents, Greek, Cyrillic, Hebrew and Arabic cost 2. Devanagari, Chinese, Japanese and Korean cost 3. Emoji and rarer historic scripts cost 4.
7. So the same sentence stored in Hindi takes roughly three times the bytes of the English version, before you even count that Hindi may need more characters.
8. `len()` in Python 3 counts code points, not bytes. `strlen()` in C counts bytes. `.length` in JavaScript counts UTF-16 code units, so an emoji outside the basic plane counts as 2.
9. A byte order mark, U+FEFF, encoded as the bytes `EF BB BF`, sometimes appears at the start of a UTF-8 file written by Windows tools. It is legal but discouraged, and it breaks shell scripts, because the shell sees those three bytes before the `#!` line.

52.4 The networking reference

52.4.1 OSI and TCP/IP side by side

1. The OSI model has seven layers. It was published as ISO/IEC 7498-1, with the first version in 1984.
2. The TCP/IP model has four layers. It was written down in RFC 1122 in October 1989, by which time the protocols had already been running for years.
3. OSI is the vocabulary everyone uses. TCP/IP is the thing that actually runs. Both statements are true at the same time.

OSI	Name	TCP/IP	Examples
7	Application	Application	HTTP, DNS, SSH
6	Presentation	Application	TLS, JPEG, UTF-8
5	Session	Application	RPC, TLS sessions
4	Transport	Transport	TCP, UDP, QUIC
3	Network	Internet	IP, ICMP, IPsec
2	Data link	Link	Ethernet, Wi-Fi, ARP
1	Physical	Link	copper, fibre, radio

4. Three honest notes about that table.
5. TLS does not fit cleanly. It sits above TCP and below HTTP, which is layer 6-ish, but it has its own handshake and session resumption, which is layer 5-ish. People argue about this. Nobody wins.
6. ARP is called layer 2 by most people and layer 2.5 by pedants, because it carries layer 3 addresses inside layer 2 frames.
7. QUIC, standardized in RFC 9000 in May 2021, runs on top of UDP but does transport work: streams, ordering, congestion control, encryption. In OSI terms it is layer 4 built on top of layer 4.
8. Where the reader's own failure sat: DNS worked, so layer 7 name lookup was fine. The TCP handshake never completed, so the failure was at layer 4 or below, on the path.

52.4.2 The RFC 1918 private ranges

1. RFC 1918, published February 1996, reserves three blocks that anyone may use inside a private network and that no router on the public internet will forward.

Block	Range	Addresses
10.0.0.0/8	10.0.0.0 to 10.255.255.255	16,777,216
172.16.0.0/12	172.16.0.0 to 172.31.255.255	1,048,576
192.168.0.0/16	192.168.0.0 to 192.168.255.255	65,536

2. The middle block is the one people get wrong. `172.16.0.0/12` covers 172.16.x.x through 172.31.x.x, not 172.16 through 172.16.
3. That is exactly why the reader's traceroute looked odd. Hops showing `172.31.0.17`, `172.26.22.235`, `172.26.14.75` and `172.16.18.33` are all private addresses inside the same /12, used by the reader's ISP for its own core network.
4. Seeing private addresses in a traceroute is normal and proves nothing about a fault. It tells you the ISP numbers its own core privately, which is ordinary practice, especially with carrier-grade NAT.

52.4.3 Reserved IPv4 ranges with their RFCs

Block	Purpose	RFC
0.0.0.0/8	"This network"	RFC 791
0.0.0.0/32	This host, this network	RFC 1122
10.0.0.0/8	Private use	RFC 1918
100.64.0.0/10	Shared, carrier NAT	RFC 6598
127.0.0.0/8	Loopback	RFC 1122
169.254.0.0/16	Link local, no DHCP	RFC 3927
172.16.0.0/12	Private use	RFC 1918
192.0.0.0/24	IETF assignments	RFC 6890
192.0.0.0/29	DS-Lite	RFC 6333
192.0.2.0/24	Docs, TEST-NET-1	RFC 5737
192.31.196.0/24	AS112 redirection	RFC 7535
192.52.193.0/24	AMT relay	RFC 7450
192.88.99.0/24	Old 6to4, deprecated	RFC 7526
192.168.0.0/16	Private use	RFC 1918
192.175.48.0/24	AS112 delegation	RFC 7534
198.18.0.0/15	Benchmarking only	RFC 2544
198.51.100.0/24	Docs, TEST-NET-2	RFC 5737
203.0.113.0/24	Docs, TEST-NET-3	RFC 5737
224.0.0.0/4	Multicast	RFC 5771
240.0.0.0/4	Reserved, future use	RFC 1112
255.255.255.255/32	Limited broadcast	RFC 919

1. The full list lives in the IANA IPv4 Special-Purpose Address Registry. The rows above are the ones you meet.
2. `169.254.x.x` on your own machine means DHCP failed. The machine gave itself a link-local address because no server answered. If you see it, fix DHCP before you look at anything else.
3. `100.64.0.0/10` is the range your ISP uses when it puts many customers behind one public address. If your router's outside address is in this range, you are behind carrier-grade NAT and you cannot accept inbound connections.
4. Use `192.0.2.0/24`, `198.51.100.0/24` and `203.0.113.0/24` in documentation and examples. They exist for that. Never invent an example address that belongs to somebody.

52.4.4 CIDR from /8 to /32

1. The prefix length is how many leading bits are the network part. The rest is the host part.
2. Total addresses is 2 to the power of (32 minus the prefix). Usable hosts is that minus two, because the all-zeros address is the network itself and the all-ones address is the broadcast.

Prefix	Netmask	Total	Usable hosts
/8	255.0.0.0	16,777,216	16,777,214
/9	255.128.0.0	8,388,608	8,388,606
/10	255.192.0.0	4,194,304	4,194,302
/11	255.224.0.0	2,097,152	2,097,150
/12	255.240.0.0	1,048,576	1,048,574
/13	255.248.0.0	524,288	524,286
/14	255.252.0.0	262,144	262,142
/15	255.254.0.0	131,072	131,070
/16	255.255.0.0	65,536	65,534
/17	255.255.128.0	32,768	32,766
/18	255.255.192.0	16,384	16,382
/19	255.255.224.0	8,192	8,190
/20	255.255.240.0	4,096	4,094
/21	255.255.248.0	2,048	2,046
/22	255.255.252.0	1,024	1,022
/23	255.255.254.0	512	510
/24	255.255.255.0	256	254
/25	255.255.255.128	128	126
/26	255.255.255.192	64	62
/27	255.255.255.224	32	30
/28	255.255.255.240	16	14
/29	255.255.255.248	8	6
/30	255.255.255.252	4	2
/31	255.255.255.254	2	2
/32	255.255.255.255	1	1

3. Two special cases. A /31 has no network or broadcast address; RFC 3021, from December 2000, allows both addresses to be hosts, which is how router-to-router links are numbered. A /32 is one single address, used for loopbacks and for host routes.
4. Classless Inter-Domain Routing replaced the old class A, B and C scheme. It was defined in RFC 1518 and RFC 1519 in September 1993, and was a direct response to the routing table growing faster than router memory.

52.4.5 The ports worth memorizing

1. Port numbers are 16 bits, so 0 to 65535. RFC 6335 splits them into three ranges: 0 to 1023 system, 1024 to 49151 user or registered, and 49152 to 65535 dynamic or ephemeral.
2. On Linux the outgoing ephemeral range is usually 32768 to 60999. On macOS it is 49152 to 65535. Both are settings, not standards.

Port	Service	Note
20, 21	FTP data, control	plaintext, obsolete
22	SSH, SFTP, scp	git over SSH
23	Telnet	plaintext, do not use
25	SMTP relay	often blocked at home
53	DNS	UDP and TCP
67, 68	DHCP server, client	UDP
69	TFTP	UDP, no auth
80	HTTP	plaintext

Port	Service	Note
110	POP3	plaintext
123	NTP	UDP
143	IMAP	plaintext
161, 162	SNMP, SNMP trap	UDP
179	BGP	over TCP
389	LDAP	plaintext
443	HTTPS, and HTTP/3	TCP and UDP
445	SMB file sharing	never expose
465	SMTP over TLS	submission
514	syslog	UDP
587	SMTP submission	with STARTTLS
636	LDAPS	over TLS
993	IMAPS	over TLS
995	POP3S	over TLS
1080	SOCKS proxy	
1194	OpenVPN	UDP usually
1433	Microsoft SQL Server	
1521	Oracle database	
3306	MySQL, MariaDB	
3389	RDP, remote desktop	never expose
5432	PostgreSQL	
5672	AMQP, RabbitMQ	
5900	VNC	
6379	Redis	no auth by default
6443	Kubernetes API server	
8080, 8000	HTTP alternate	dev convention
8443	HTTPS alternate	dev convention
9042	Cassandra	
9090	Prometheus	
9200	Elasticsearch	
11211	Memcached	
27017	MongoDB	
51820	WireGuard	UDP

3. Note the two rows that mattered in the reader's session. Git over SSH uses port 22. Git over HTTPS uses port 443. In the reader's session one transport worked while the other did not, which is only possible because they are different ports to different endpoints.

52.4.6 ICMP types worth knowing

Type	Name	When you see it
0	Echo Reply	a successful ping
3	Destination Unreachable	see the code below
5	Redirect	router says use another
8	Echo Request	the ping you sent
11	Time Exceeded	every traceroute hop
12	Parameter Problem	malformed header

1. Type 3 carries a code that says exactly what was unreachable.

Code	Meaning
0	Network unreachable
1	Host unreachable
2	Protocol unreachable
3	Port unreachable
4	Fragmentation needed, DF set
13	Administratively prohibited

2. Code 4 is the one that causes mysterious hangs. It is how path MTU discovery works. If a firewall blocks it, large packets vanish and small ones get through, and a page loads its HTML but never its images.
3. Code 13 is a firewall openly telling you it refused. That is a polite refusal, and it is evidence. Silence is not.
4. Traceroute works entirely on type 11 code 0. Each probe goes out with a time-to-live one higher than the last, and each router that decrements it to zero sends back type 11.
5. Under IPv6, ICMPv6 is defined by RFC 4443 and the numbers change. Echo Request is 128, Echo Reply is 129, Packet Too Big is 2. ICMPv6 is not optional: blocking it breaks IPv6 entirely, because Neighbor Discovery rides on types 133 to 137.
6. The reader's trace showed * * * from hop 13 to hop 20. That means no type 11 replies came back from those hops. It does not mean the packets died there. Many routers rate-limit or refuse to generate ICMP.

52.4.7 HTTP methods

1. **Safe** means the method is not supposed to change anything on the server.
2. **Idempotent** means doing it twice has the same effect as doing it once.
3. Both properties are promises made by the specification, currently RFC 9110, published June 2022. A server can break them. A well-behaved one does not.

Method	Safe	Idempotent	Usual use
GET	yes	yes	fetch a resource
HEAD	yes	yes	headers only
OPTIONS	yes	yes	what is allowed
TRACE	yes	yes	echo the request
PUT	no	yes	replace at this URL
DELETE	no	yes	remove
POST	no	no	create or act
PATCH	no	no	partial update
CONNECT	no	no	tunnel, used by proxies

4. The practical consequence: a client may retry a GET, HEAD, PUT or DELETE automatically after a network failure. It must not retry a POST automatically, because it cannot know whether the first one landed.
5. That is exactly the reasoning behind the git failure in section 52.8. A disconnect while reading the response tells you nothing about whether the write was applied.

52.4.8 HTTP status codes

1. The first digit gives the family. One hundred series is informational, two hundred success, three hundred redirection, four hundred client error, five hundred server error.
2. Codes marked with a star are ones you should know without looking up.

Code	Name	Star
100	Continue	
101	Switching Protocols	star
102	Processing	
103	Early Hints	
200	OK	star
201	Created	star
202	Accepted	
203	Non-Authoritative Info	
204	No Content	star
205	Reset Content	
206	Partial Content	star
207	Multi-Status	
208	Already Reported	
226	IM Used	
300	Multiple Choices	
301	Moved Permanently	star
302	Found	star
303	See Other	star
304	Not Modified	star
305	Use Proxy, deprecated	
307	Temporary Redirect	star
308	Permanent Redirect	star
400	Bad Request	star
401	Unauthorized	star
402	Payment Required	
403	Forbidden	star
404	Not Found	star
405	Method Not Allowed	star
406	Not Acceptable	
407	Proxy Auth Required	
408	Request Timeout	
409	Conflict	star
410	Gone	
411	Length Required	
412	Precondition Failed	star
413	Content Too Large	
414	URI Too Long	
415	Unsupported Media Type	
416	Range Not Satisfiable	
417	Expectation Failed	
418	I am a teapot	
421	Misdirected Request	
422	Unprocessable Content	star
423	Locked	
424	Failed Dependency	
425	Too Early	
426	Upgrade Required	
428	Precondition Required	
429	Too Many Requests	star

Code	Name	Star
431	Headers Too Large	
451	Legal Reasons	star
500	Internal Server Error	star
501	Not Implemented	
502	Bad Gateway	star
503	Service Unavailable	star
504	Gateway Timeout	star
505	Version Not Supported	
506	Variant Also Negotiates	
507	Insufficient Storage	
508	Loop Detected	
510	Not Extended	
511	Network Auth Required	star

3. Four distinctions that people get wrong every day.
4. 401 versus 403. A 401 means "I do not know who you are; send credentials". A 403 means "I know who you are and you may not". The reader's missing `workflow` scope produced a 403-class refusal, not a 401.
5. 502 versus 503 versus 504. A 502 means the gateway got a broken answer from the thing behind it. A 503 means the service is deliberately unavailable, often overloaded or in maintenance. A 504 means the thing behind it never answered in time.
6. 301 versus 308, and 302 versus 307. The older pair, 301 and 302, historically allowed clients to change a POST into a GET when following. The newer pair, 307 and 308, forbid changing the method. Use 308 and 307 when you care.
7. 418 is real and is a joke. It comes from RFC 2324, the Hyper Text Coffee Pot Control Protocol, published on 1 April 1998. RFC 7168 extended it to teapots in 2014. It is not part of HTTP.
8. 451 is named for the novel Fahrenheit 451 and was standardized in RFC 7725 in February 2016. It means content is blocked for legal reasons.

52.4.9 Common DNS record types

Type	Number	What it holds
A	1	one IPv4 address
AAAA	28	one IPv6 address
CNAME	5	an alias to another name
MX	15	mail server plus priority
NS	2	authoritative name server
PTR	12	name for an address
SOA	6	zone parameters
TXT	16	free text, SPF, DKIM
SRV	33	host, port, priority
CAA	257	who may issue certificates
DS	43	DNSSEC delegation signer
DNSKEY	48	DNSSEC public key
RRSIG	46	DNSSEC signature
NSEC3	50	DNSSEC proof of absence
SVCB	64	service binding
HTTPS	65	HTTPS service binding
TLSA	52	certificate pinning, DANE

Type	Number	What it holds
SSHFP	44	SSH host key fingerprint

1. A CNAME cannot exist at the zone apex, that is at `example.com` itself, because the apex must also hold SOA and NS records and a CNAME forbids any other record at the same name. Providers work around this with a non-standard record they call ALIAS or ANAME. That is a vendor convention, not a standard.
2. SVCB and HTTPS records were standardized in RFC 9460 in November 2023. The HTTPS record can tell a browser to use HTTP/3, and can carry the encrypted client hello parameters, before any connection is made.
3. TTL is a number of seconds attached to every record. It is a request, not a guarantee. Resolvers may cache longer or shorter.

52.4.10 TCP flags and states

1. The TCP header has a flags field. The current specification is RFC 9293, from August 2022, which replaced RFC 793 from 1981.

Flag	Name	Meaning
SYN	Synchronize	start a connection
ACK	Acknowledge	I received up to here
FIN	Finish	I have no more to send
RST	Reset	tear down now, refused
PSH	Push	deliver to the app now
URG	Urgent	urgent pointer valid
ECE	ECN Echo	congestion signalled
CWR	Congestion Window Reduced	I slowed down

2. The three-way handshake is SYN, then SYN plus ACK, then ACK. The four-way close is FIN, ACK, FIN, ACK.
3. An RST in reply to a SYN means "nothing is listening on that port". That is a refusal, and it is information. It arrives in milliseconds.
4. No reply at all to a SYN means the packet was dropped somewhere and nobody told you. That is what the reader saw. It costs the full timeout.

State	Meaning
CLOSED	no connection
LISTEN	waiting for a SYN
SYN_SENT	sent SYN, waiting
SYN_RECEIVED	got SYN, sent SYN-ACK
ESTABLISHED	data can flow
FIN_WAIT_1	we sent FIN
FIN_WAIT_2	our FIN was acked
CLOSE_WAIT	they sent FIN, we have not
CLOSING	both sent FIN together
LAST_ACK	we sent our FIN last
TIME_WAIT	waiting out stray packets

5. Two states are diagnostic on their own.

6. Many sockets stuck in `CLOSE_WAIT` means your own application is not calling close. That is an application bug, not a network problem.
7. Many sockets in `TIME_WAIT` is normal on a busy client. The specification says wait twice the maximum segment lifetime, which by the letter of the standard is four minutes. Linux uses a fixed sixty seconds and does not let you change it without recompiling.
8. A connection that sits in `SYN_SENT` and never leaves it is the exact signature of the reader's fault: the SYN went out, nothing came back.

52.5 Networking command cheat sheet

52.5.1 How to read this section

1. Every entry below gives the macOS form first, because that is the machine in the running example, then the Linux difference on the next line.
2. Then two lines that most cheat sheets omit.
3. **Proves** is the set of things that must be true for the command to have succeeded. If it passed, all of these are true.
4. **Does not prove** is the trap. These are things people assume a pass means and that it does not mean.
5. Getting this second line right is the difference between diagnosing a problem and guessing at it.

52.5.2 The commands at a glance

Command	Layer it tests	Needs network
`ifconfig` or `ip addr`	your own interface	no
`arp -a`	local link, layer 2	local only
`netstat -rn` or `ip route`	your routing table	no
`ping`	ICMP reachability	yes
`traceroute`	path, hop by hop	yes
`mtr`	path plus loss over time	yes
`dig`, `host`, `nslookup`	DNS resolution	yes
`scutil --dns`	which resolver macOS uses	no
`nc -vz`	one TCP port	yes
`telnet`	one TCP port, old	yes
`nmap`	many ports at once	yes
`curl -v`	full request, TLS, HTTP	yes
`wget`	file download	yes
`openssl s_client`	TLS handshake only	yes
`tcpdump`	actual packets on the wire	no
`lsof -i`	which process holds a port	no
`ss` or `netstat -an`	socket states	no
`whois`	who owns a name or address	yes
`networksetup`	macOS network settings	no

52.5.3 ping

1. macOS: `ping -c 5 github.com`
2. Linux: same, plus `-w 5` sets a total deadline in seconds.
3. Useful flags, macOS: `-c count`, `-i interval`, `-s size`, `-t timeout in seconds for the whole run`, `-D` set the do-not-fragment bit.
4. Careful: on Linux `-t` means time-to-live, not timeout. Same letter, different meaning. Linux uses `-M do` for do-not-fragment.

5. Proves: your interface is up, your route exists, the target address is reachable by ICMP, and something at that address answers ICMP.
6. Does not prove: that any TCP port is open, that the web server is running, that DNS is correct, or that the service you want works. Ping tests ICMP and only ICMP.
7. A failed ping proves even less. Very many hosts and networks drop ICMP on purpose. Cloudflare, AWS load balancers and corporate firewalls commonly do. A ping that fails while `curl` succeeds is completely normal.
8. Path MTU test, macOS: `ping -D -s 1472 github.com`. If 1472 fails and 1400 works, something on the path has a smaller MTU. Add 28 bytes of headers to get the real figure: 1472 plus 28 is 1500.

52.5.4 traceroute

1. macOS: `traceroute -n github.com`
2. Linux: `traceroute -n github.com`, and `tracepath` is a simpler alternative that needs no privileges.
3. Useful flags: `-n` no reverse lookups so it runs fast, `-m 30` maximum hops, `-q 1` one probe per hop, `-w 2` wait two seconds, `-I` use ICMP probes instead of UDP.
4. macOS default probes are UDP to high port numbers. Linux default is the same. Windows `tracert` uses ICMP by default. This matters, because firewalls filter the three differently.
5. TCP mode is the most useful on a modern network. Linux: `traceroute -T -p 443 github.com`. macOS: `traceroute -P TCP -p 443 github.com`.
6. Proves: which routers along the path are willing to answer, in order, up to the last one that replied.
7. Does not prove: that the path is symmetric, that the reply came back the same way it went, that a hop with no reply is broken, or that the last replying hop is where the trouble is.
8. The reader's trace stopped replying after hop 12, `ae106-0.rwa04.pnq20.ntwk.msn.net (104.44.20.42)`. That does not mean hop 12 dropped the traffic. It means hops beyond it did not send back ICMP time-exceeded messages, which is extremely common.
9. Hops showing more than one address, as hops 5, 6, 8, 9 and 11 did in the reader's trace, mean the traffic is being spread over parallel links. That is load balancing working correctly, not a fault.

52.5.5 mtr

1. `mtr` combines ping and traceroute and keeps running, so it shows loss per hop over time.
2. macOS: not installed by default. Install with `brew install mtr`, then run with `sudo mtr github.com` because it needs raw sockets.
3. Linux: `mtr github.com`, often already installed.
4. Report mode, for pasting into a ticket: `mtr -rwc 100 github.com`. That means report, wide output, one hundred cycles.
5. TCP mode: `mtr -T -P 443 github.com`.
6. Proves: sustained loss or latency at and beyond a specific hop.
7. Does not prove: that loss shown at a middle hop is real. Routers deliberately rate-limit the ICMP replies they generate for themselves while forwarding everything else perfectly. Only loss that continues to the final hop and stays there is real loss.
8. That single rule saves more wasted hours than any other line in this chapter. Read the last row, not the middle rows.

52.5.6 dig, host and nslookup

1. `dig` is the precise one. Use it.
2. macOS: `dig github.com`. On recent macOS versions Apple has removed the BIND lookup tools from the base install. If `dig` is missing, get it with `brew install bind`.
3. Linux: `dig github.com`, from the `dnsutils` or `bind-utils` package.

```
dig +short github.com                # just the answer
dig +short A github.com              # only IPv4
dig +short AAAA github.com           # only IPv6
```

```
dig @1.1.1.1 github.com              # ask a specific resolver
dig @8.8.8.8 github.com              # ask a different one
dig +trace github.com                # walk from the root down
dig -x 20.207.73.82                  # reverse lookup
dig +noall +answer github.com        # answer section only
dig NS github.com                    # nameservers
dig MX github.com                    # mail servers
dig TXT github.com                   # text records
dig +tcp github.com                  # force TCP not UDP
dig +dnssec github.com               # ask for signatures
```

4. Proves: that a resolver answered, what it answered, and how long the answer may be cached.
5. Does not prove: that the address is reachable, that it is the right address for you, or that your operating system will use it. `dig` talks to the resolver directly and bypasses the system cache.
6. That last point matters on macOS, which has its own resolver daemon. To see what the system will actually do, use `dscacheutil -q host -a name github.com`, and to see which resolvers are configured use `scutil --dns`.
7. `host github.com` gives a one-line answer. `nslookup github.com 1.1.1.1` works everywhere including Windows. Both are fine for a quick check and worse than `dig` for anything else.
8. In the reader's session, name resolution worked: `github.com` resolved to `20.207.73.82` through the Cloudflare resolver at `1.1.1.1`. That single fact eliminated DNS as the cause.

52.5.7 curl

1. The single most useful network command on any machine.
2. macOS and Linux syntax are the same. macOS has shipped curl since 10.4.

```
curl -v https://github.com           # show the whole conversation
curl -I https://github.com           # headers only, HEAD request
curl -sS -o /dev/null -w '%{http_code}\n' https://github.com
curl -L https://example.com          # follow redirects
curl --max-time 10 https://github.com # give up after 10 seconds
curl --connect-timeout 3 https://x.io # give up connecting after 3
curl -4 https://github.com           # force IPv4
curl -6 https://github.com           # force IPv6
curl --resolve github.com:443:20.207.73.82 https://github.com
curl -H 'Accept: application/json' https://api.github.com
curl -X POST -d 'a=1' https://example.com/f
curl -k https://self-signed.example  # skip cert check
curl --http1.1 https://github.com    # force HTTP/1.1
curl -x http://proxy:3128 https://x.io # go via a proxy
```

3. The timing breakdown is the hidden gem. It tells you which stage was slow.

```
curl -sS -o /dev/null -w \
'dns %{time_namelookup} conn %{time_connect} tls
%{time_appconnect} first %{time_starttransfer} total
%{time_total}\n' https://github.com
```

4. Proves, when it returns a status code: DNS resolved, TCP connected, TLS completed if the URL was HTTPS, the request was sent, and a server answered.
5. Does not prove: that the answer is correct, that the site works in a browser, or that the certificate is trusted if you passed `-k`.
6. A hang with no output after `Trying <address>:443...` proves the TCP handshake never completed. That is a layer 4 or below problem and nothing above it can be at fault yet.

7. That is exactly what the reader saw. `curl -v https://github.com` printed `Trying 20.207.73.82:443...` and then produced nothing for fifteen seconds. No RST, no ICMP unreachable, no TLS error. Silence.
8. Silence is a specific finding. A server that is down sends RST. A firewall that refuses politely sends ICMP type 3 code 13. Nothing at all means a silent drop somewhere on the path.

52.5.8 wget

1. macOS: not installed by default. `brew install wget`.
2. Linux: usually present.

```
wget https://example.com/file.tar.gz   # save with its own name
wget -O out.bin https://example.com/f  # save under a chosen name
wget -O - https://example.com          # print to standard output
wget --spider https://example.com    # exists? do not save
wget -c https://example.com/big.iso    # continue a partial download
wget -T 10 -t 3 https://example.com    # 10s timeout, 3 tries
```

3. Proves and does not prove: the same as curl.
4. Use wget for downloading files and mirroring. Use curl for inspecting a request. That split is a convention, not a rule.

52.5.9 nc, the netcat tool

1. The cleanest way to test one TCP port and nothing else.
2. macOS: `nc -vz github.com 443`. Apple ships its own build of the OpenBSD version.
3. Linux: `nc -vz -w 3 github.com 443` with the OpenBSD build.

```
nc -vz github.com 443            # is port 443 open
nc -vz -w 3 github.com 22        # with a 3 second timeout
nc -vzu 1.1.1.1 53               # UDP, far less reliable
nc -l 8080                       # listen on 8080, a fake server
nc -vz github.com 20-25          # a small range
```

4. Proves, on success: the full TCP three-way handshake completed to that address on that port.
5. Does not prove: that the service behind the port speaks the protocol you expect, that TLS will succeed, or that your credentials are valid.
6. This is the correct tool for separating "the network is blocked" from "the application is refusing me". If `nc -vz` succeeds and `curl` fails, the problem is above layer 4.
7. UDP testing with `-u` is close to useless on its own, because a silent UDP port and an open UDP port look identical. Only the application reply proves anything.

52.5.10 telnet

1. Removed from macOS since High Sierra, version 10.13, in 2017. Install with `brew install telnet` or, better, use `nc` instead.
2. `telnet github.com 443` then type text. Historically used to speak HTTP by hand.
3. Proves: a TCP connection opened, and you can type raw bytes into it.
4. Does not prove: anything about encryption. Telnet is plaintext. Never use the telnet protocol itself, on port 23, for real work. It sends passwords in the clear. SSH replaced it in 1995.

52.5.11 nmap

1. Not installed by default anywhere. `brew install nmap` on macOS, `apt install nmap` on Debian and Ubuntu.

```
nmap -Pn -p 443 github.com         # one port, skip host discovery
nmap -p 22,80,443 192.168.0.1      # a few ports
```

```
nmap -p- 192.168.0.1              # all 65535 ports, slow
nmap -sV -p 443 example.com       # try to identify the service
nmap -sn 192.168.0.0/24           # who is on my network
nmap --top-ports 100 192.168.0.1  # the common ones only
sudo nmap -sS -p 443 example.com  # half-open scan, needs root
```

2. Proves: which of the scanned ports answered, and how.
3. Does not prove: that a filtered result means blocked by the target. It may be blocked anywhere along the path, including by your own network.
4. One legal and social rule, and it is not optional. Scan only machines you own or have written permission to scan. Port scanning other people's networks is, in many countries including India under the Information Technology Act 2000, potentially an offence, and at minimum it will get your connection blocked.

52.5.12 netstat, ss and lsof

1. These look at your own machine, not the network. None of them need connectivity.
2. macOS: `netstat -an -p tcp` shows all TCP sockets. `netstat -rn` shows the routing table. `netstat -i` shows interface counters. `netstat -s` shows protocol statistics.
3. Linux: `netstat` is deprecated in favour of `ss`, from the `iproute2` package.

```
ss -tulpn                     # Linux: listening TCP+UDP, pids
ss -tn state established      # Linux: established TCP only
ss -s                         # Linux: summary counts
netstat -an -p tcp            # macOS: all TCP sockets
netstat -rn                   # both: routing table
lsof -i -P -n                 # both: sockets with process names
lsof -nP -iTCP:443            # both: who is using port 443
lsof -nP -iTCP -sTCP:LISTEN   # both: what is listening
```

4. Proves: exactly which local process holds which port, and what state each connection is in.
5. Does not prove: anything about the remote end. A socket shown as `ESTABLISHED` locally can be dead, if the peer vanished without sending FIN or RST. Only traffic proves liveness.
6. On macOS, `lsof` needs `sudo` to see other users' processes.

52.5.13 arp

1. macOS and Linux: `arp -a` lists the address resolution cache, which maps IPv4 addresses to hardware addresses on your local network segment.
2. Linux modern form: `ip neigh show`.
3. `sudo arp -d 192.168.0.1` deletes one entry, which forces a fresh lookup.
4. Proves: which machines your computer has recently talked to on the same local segment, and their MAC addresses.
5. Does not prove: anything beyond your own network. ARP does not cross a router. If the gateway is in the table with a MAC address, your link to the router is working.
6. `arp -a` showing `192.168.0.1` with a valid MAC address would have been the first thing to check in the reader's case, and it would have passed. The local network was fine.

52.5.14 ifconfig, ip and route

1. macOS: `ifconfig` and `ifconfig en0`. Apple has not adopted the `ip` command.
2. Linux: `ip addr show`, or the short form `ip -br a`.

```
ifconfig -a                   # macOS: every interface
ifconfig en0                  # macOS: just Wi-Fi
ipconfig getifaddr en0        # macOS: just the IPv4 address
ipconfig getpacket en0        # macOS: the whole DHCP lease
```

```
netstat -rn                    # macOS: routing table
route -n get default           # macOS: which route wins
ip -br a                       # Linux: brief addresses
ip link show                   # Linux: layer 2 state
ip route show                  # Linux: routing table
ip route get 20.207.73.82      # Linux: which route wins
```

3. On macOS, `en0` is usually Wi-Fi on a laptop and Ethernet on a desktop. `lo0` is loopback. `utun0` and upwards are tunnel interfaces, created by VPNs, by iCloud Private Relay, and by some corporate agents.
4. The reader's machine showed several `utun` interfaces. That is worth noticing but does not prove a VPN was routing the traffic. Check with `route -n get default` and see which interface the default route uses.
5. Proves: your own addresses, your own routes, your own link state.
6. Does not prove: that the route leads anywhere.

52.5.15 tcpdump

1. The ground truth. Everything else is an interpretation; this is the packets.
2. macOS: `sudo tcpdump -i en0 -n host 20.207.73.82`
3. Linux: same, with the interface usually named `eth0`, `ens3` or `wlan0`.

```
sudo tcpdump -i en0 -n host 20.207.73.82
sudo tcpdump -i en0 -nn port 443
sudo tcpdump -i any -n icmp
sudo tcpdump -i en0 -n -c 20 tcp port 443
sudo tcpdump -i en0 -w capture.pcap        # save for Wireshark
sudo tcpdump -i en0 -n 'tcp[tcpflags] & tcp-syn != 0'
sudo tcpdump -i en0 -n -A port 80          # show payload as text
```

4. Flags that matter: `-n` no name lookups, `-nn` also no port names, `-i any` all interfaces on Linux, `-c N` stop after N packets, `-s 0` full packet, `-w file` write a capture file, `-A` print payload as ASCII, `-X` print hex and ASCII.
5. Proves: exactly what left your machine and exactly what came back.
6. Does not prove: what happened in the middle. You see your end of the wire and nothing else.
7. In the reader's case, tcpdump would have shown repeated SYN packets going out to `20.207.73.82` on port 443, with retransmissions at roughly one, three and seven seconds, and nothing at all coming back. That is the signature of a silent drop, and it is different from every other failure.

52.5.16 openssl s_client

1. Tests the TLS handshake by itself, with nothing above it.

```
openssl s_client -connect github.com:443 -servername github.com
openssl s_client -connect github.com:443 -showcerts
openssl s_client -connect github.com:443 -tls1_2
openssl s_client -connect github.com:443 -tls1_3
echo | openssl s_client -connect github.com:443 2>/dev/null \
  | openssl x509 -noout -subject -issuer -dates
```

2. `-servername` sets Server Name Indication, which almost every modern host requires. Without it you may get the wrong certificate or a failure.
3. Proves: TCP connected, a TLS handshake completed, which version and cipher were agreed, and which certificate chain was presented.
4. Does not prove: that the certificate is trusted by your operating system. The `openssl` command uses its own trust store, which on macOS is usually not the system keychain. A certificate that openssl rejects may work fine in Safari, and the reverse.

5. Check expiry dates with the last command above. Certificate expiry is one of the most common causes of a site that worked yesterday and not today.
6. Since 2024 the industry has been shortening maximum certificate lifetimes. The CA/Browser Forum voted in April 2025 to reduce the maximum from 398 days down in stages, reaching 47 days by March 2029. Expiry checks are getting more important, not less.

52.5.17 whois

1. `whois github.com` looks up who registered a domain name.
2. `whois 20.207.73.82` looks up who owns an address block.
3. Proves: what a registry says. Registry data is often stale, and since the European General Data Protection Regulation took effect in May 2018, most personal contact details in domain records are hidden.
4. Does not prove: who is operating the machine today.
5. In the reader's session, `whois` on `20.207.73.82` would show a Microsoft range. That is consistent with GitHub being owned by Microsoft, which acquired it in 2018 for 7.5 billion dollars in stock, and fronting traffic through Microsoft's network edge.
6. The traceroute agreed: from hop 7 onward the names all ended in `ntwk.msn.net`, which is Microsoft's backbone naming.

52.5.18 macOS-only tools

1. `scutil --dns` shows every resolver the system will use, in order, with the search domains and the interface each is bound to. This is the truth about macOS DNS; `/etc/resolv.conf` on macOS is a compatibility file and can be misleading.
2. `scutil --nwi` shows network information: which interfaces are usable, and whether IPv4 and IPv6 are available on each.
3. `scutil --proxy` shows the proxy configuration.
4. `networksetup -listallnetworkservices` lists the named services such as Wi-Fi and Ethernet.
5. `networksetup -getinfo "Wi-Fi"` shows address, mask, router and MAC.
6. `networksetup -getdnsservers "Wi-Fi"` shows configured resolvers.
7. `sudo networksetup -setdnsservers "Wi-Fi" 1.1.1.1 8.8.8.8` sets them.
8. `sudo networksetup -setdnsservers "Wi-Fi" Empty` puts it back to DHCP.
9. `sudo dscacheutil -flushcache; sudo killall -HUP mDNSResponder` clears the DNS cache. Both halves are needed on modern macOS.
10. `networkQuality` runs Apple's built-in throughput and responsiveness test. It has shipped since macOS 12 Monterey, released in 2021.
11. `ipconfig getpacket en0` prints the entire DHCP lease including the offered DNS servers, which is how you tell what the router wanted you to use.
12. The reader was using `1.1.1.1`, Cloudflare's public resolver, rather than the router. `scutil --dns` is what proves that.

52.5.19 Worked example: the reader's own five commands

1. Here is the exact sequence that settled the reader's case, with what each step eliminated.

```
Step 1  ping 192.168.0.1
        -> replies.  Local network and router: fine.

Step 2  dig +short github.com
        -> 20.207.73.82.  DNS: fine.

Step 3  curl -v --max-time 15 https://github.com
        -> "Trying 20.207.73.82:443..." then nothing.
        -> TCP handshake never completed.

Step 4  traceroute -n github.com
```

```
        -> reaches Microsoft's Pune edge, then * * *.
        -> path exists at least that far.

Step 5  same URL on mobile data, same phone
        -> loads instantly.
        -> the server is up; the destination is fine.
```

2. What that combination proves: the failure is specific to the path between this home connection and that destination, at the transport layer, and it is a silent drop rather than a refusal.
3. What it does not prove: who is dropping the packets, or why. The evidence does not name a party, and you should not either.

52.6 Decision tree: I cannot reach a site

52.6.1 The shape of the procedure

1. Work upward through the layers. Never start in the middle.
2. Each step either passes, which eliminates everything below it, or fails, which tells you exactly where to stop looking.
3. Do not skip a step because you are sure it is fine. The reader's case looked like a DNS problem and was not.

```
      START: "I cannot reach a site"
                  |
       1. Is any other site reachable?
          no -> your machine or link. Go to A.
          yes v
       2. Does the name resolve?
          no -> DNS. Go to B.
          yes v
       3. Does TCP connect on the port?
          no -> path or firewall. Go to C.
          yes v
       4. Does TLS complete?
          no -> certificate or version. Go to D.
          yes v
       5. Does HTTP return a status code?
          no -> the app is hung. Go to E.
          yes v
       6. Is the status what you expected?
          no -> application layer. Go to F.
          yes -> it works. The problem is elsewhere.
                 Go to G.
```

52.6.2 Step 1: is anything reachable at all

1. Run `ping -c 3 192.168.0.1`, using your own gateway address.
2. Find your gateway with `route -n get default` on macOS or `ip route show` on Linux.
3. If the gateway does not answer, stop. The problem is your machine, your cable, or your Wi-Fi. Check `ifconfig en0` for an address.
4. If your address begins `169.254.`, DHCP failed. Nothing else matters until that is fixed. Renew the lease or restart the interface.
5. If the gateway answers, run `ping -c 3 1.1.1.1`.
6. If a public address answers but names do not resolve, jump straight to step 2. That combination is the classic DNS-only failure.
7. If the gateway answers and `1.1.1.1` does not, the problem is between your router and the internet. Check the router's own status page and the physical line.

8. Branch A conclusion: local. Fix the interface, the lease, the cable or the router before going further.

52.6.3 Step 2: does the name resolve

1. Run `dig +short github.com`.
2. If you get an address, DNS works. Note the address and move on.
3. If you get nothing, ask a different resolver: `dig @1.1.1.1 +short github.com` and `dig @8.8.8.8 +short github.com`.
4. If a public resolver answers and your configured one does not, your resolver is the problem. On macOS, check with `scutil --dns` and change it with `networksetup -setdnsservers`.
5. If no resolver answers for that one name but others resolve, the domain itself may be broken. Check with `dig +trace github.com`, which walks from the root servers down and shows where the chain stops.
6. If the answer looks wrong, for example a private address for a public site, suspect a captive portal, a filtering resolver, or a modified hosts file. Check `/etc/hosts` on macOS and Linux.
7. Clear the cache and try again: on macOS, `sudo dscacheutil -flushcache; sudo killall -HUP mDNSResponder`.
8. Branch B conclusion: DNS. Nothing above this layer can work until a name turns into an address.

52.6.4 Step 3: does TCP connect

1. Run `nc -vz github.com 443`.
2. Success means the three-way handshake completed. Move to step 4.
3. Now read the failure carefully, because the three failure modes mean completely different things.
4. **Connection refused**, immediately. You received a TCP RST. Something is there and it said no. The host is up; the port is closed. Suspect a service that is not running, or the wrong port.
5. **No route to host** or **host unreachable**, quickly. You received an ICMP type 3. A router along the way openly told you it could not deliver. Suspect routing or a firewall that refuses politely.
6. **Timeout**, after many seconds, with no message at all. Nothing came back. This is a silent drop. It is the hardest case and it is the reader's case.
7. Confirm which of the three you have with `curl -v --max-time 15` and, if you can, with `sudo tcpdump -i en0 -n host <address> and port 443` in another window.
8. If it is a timeout, test whether it is that one destination or all destinations: try `nc -vz 1.1.1.1 443` and `nc -vz google.com 443`.
9. Test whether it is that one port: try `nc -vz github.com 22` and `nc -vz github.com 80`.
10. Test whether it is that one network path: use a different network. Mobile data on a phone, with tethering or with the same URL typed into the phone, is the cleanest test available to a home user.
11. If it works on a different network and not on yours, the destination is fine and the path is not.
12. Branch C conclusion: transport or path. See section 52.6.8 for what to do next, because this is where the reader ended up.

52.6.5 Step 4: does TLS complete

1. Run `openssl s_client -connect github.com:443 -servername github.com`.
2. Success prints a certificate chain and a line saying which protocol version and cipher were negotiated.
3. `certificate verify failed` means the chain did not validate. Look at the reason code printed on the same line.
4. `certificate has expired` means exactly that. Check the dates with `openssl x509 -noout -dates`.
5. `unable to get local issuer certificate` usually means an intermediate certificate is missing from the server's chain, or your trust store is old.
6. `wrong version number` or `no protocols available` means a version mismatch. Old servers that only speak TLS 1.0 are refused by modern clients. Test with `-tls1_2` and `-tls1_3` explicitly.
7. A hostname mismatch means the certificate does not cover the name you asked for. Check the subject alternative names with `openssl x509 -noout -text` and look for the SAN extension.

8. If TLS fails only in your browser but succeeds with openssl, suspect a corporate inspection proxy that is re-signing traffic, or a VPN.
9. Branch D conclusion: TLS. The network is fine; the trust is not.

52.6.6 Step 5 and 6: does HTTP answer, and correctly

1. Run `curl -sS -o /dev/null -w '%{http_code} %{time_total}\n'` against the URL.
2. No status code and a long time means the connection opened and the server never replied. That is an application hang, not a network fault. Branch E.
3. A 502, 503 or 504 means you reached a gateway and the thing behind it is broken or overloaded. That is the operator's problem, not yours. Branch F.
4. A 401 or 403 means you reached the right service and it refused you. Check your credentials, token, or scopes. Branch F.
5. A 404 means you reached the right service and asked for the wrong path.
6. A 429 means you are being rate limited. Read the `Retry-After` header.
7. A 200 with wrong content means caching, a stale CDN copy, or the wrong virtual host. Try `curl -H 'Cache-Control: no-cache'` and try the origin directly with `--resolve`.
8. Branch F conclusion: application layer. The whole network stack worked.

52.6.7 Everything passes and it still fails

1. This happens, and it has a small number of causes. Work through them in this order.
2. It works from the command line but not in the browser. Suspect a browser extension, a proxy setting inside the browser, or a service worker holding a cached broken response. Test in a private window with extensions off.
3. It works over IPv4 and not IPv6, or the reverse. Force each with `curl -4` and `curl -6`. Broken IPv6 with working IPv4 is common and produces long pauses because the browser tries IPv6 first.
4. The reader's machine reported `IPv6: (none)`, which removes this branch from the reader's own case entirely.
5. It works for small requests and fails for large ones. That is an MTU problem. Test with `ping -D -s 1472` and step the size down. If 1400 works and 1472 does not, something on the path has a smaller MTU and is not sending the ICMP message that would let path MTU discovery work.
6. It works sometimes. Run `mtr -rwc 200` and look at the loss on the final row only. Intermittent means loss, congestion, or a load-balanced path where one of several parallel links is broken.
7. That last case is worth naming: if a hop shows several addresses, as hops 5, 6, 8, 9 and 11 did in the reader's trace, then some flows take one link and some take another. A single bad link out of four gives you a fault that appears for one in four connections and looks random.
8. It works for other people on the same network. Compare `scutil --dns`, `route -n get default` and the proxy settings between the two machines.
9. It stopped working at a specific time. Ask what changed then: a certificate expiry, a DNS TTL running out, a scheduled firewall rule, a router reboot.
10. It works with a VPN on. That proves the destination and the server are fine and that the direct path is the problem, because the VPN carries your traffic through a different route. It does not prove who is blocking.

52.6.8 Exactly what to report to an ISP

1. Support staff cannot act on "the internet is not working". They can act on a reproducible measurement. Give them these seven things, in this order.
2. Your own public address and the time, including the time zone. Get the address with `curl -4 ifconfig.me` when you have any working path, or from the router's status page.
3. The destination name and the exact address it resolves to on your machine. For the reader: `github.com` resolving to `20.207.73.82`.

4. The exact symptom, in protocol terms. Not "it is slow". Say: "TCP SYN to 20.207.73.82 port 443 receives no response of any kind, no RST and no ICMP unreachable, and times out after 15 seconds."
5. A full traceroute, pasted verbatim, with `-n` so it is fast and unambiguous. Include the hop where replies stop.
6. The control test that proves it is not the destination. For the reader: "The same URL loads instantly over mobile data on the same phone, at the same moment."
7. The control test that proves it is not everything. For example: "Other HTTPS sites work normally from the same machine at the same time."
8. An `mtr` report over at least 100 cycles, in report mode, so they can see loss over time rather than one snapshot.
9. What you have already ruled out, listed plainly: local network, DNS, the destination server, IPv6.
10. One sentence of what you want: "Please check whether traffic from my connection to 20.207.73.82 on port 443 is being dropped in your network."
11. Do not accuse anyone. The evidence supports a silent drop somewhere on that path. It does not identify who is doing it, and claiming otherwise makes the ticket easier to dismiss.
12. Keep a copy of every command output with timestamps. If the fault is intermittent, that record is the only thing that will get it taken seriously.

52.7 Git command cheat sheet

52.7.1 How to read the two extra columns

1. **Net** says whether the command needs the network. If it says no, the command works with the cable unplugged. That is most of git.
2. **Risk** says whether the command can destroy work that is not recoverable from the reflog.
3. Low means safe. Medium means it rewrites history but the old commits stay reachable through `git reflog` for 90 days by default. High means it can delete uncommitted changes or untracked files, which nothing can recover.

52.7.2 Setup and configuration

Command	What it does	Net	Risk
`git init`	make a new repository	no	low
`git init --bare`	make a server-side repo	no	low
`git clone URL`	copy a repo and history	yes	low
`git clone --mirror`	exact copy of all refs	yes	low
`git clone --depth 1`	shallow, latest commit	yes	low
`git config --global x y`	set a user-wide setting	no	low
`git config --list`	show every setting	no	low
`git config --list --show-origin`	and which file set it	no	low
`git remote -v`	list remotes and URLs	no	low
`git remote add name URL`	add a remote	no	low
`git remote set-url name URL`	change a remote URL	no	low
`git help command`	full documentation	no	low

1. The three settings worth doing on a new machine, once:

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
git config --global init.defaultBranch main
git config --global pull.rebase false
git config --global core.editor "vim"
```

52.7.3 Local work

Command	What it does	Net	Risk
`git status`	what changed, what is staged	no	low
`git status -sb`	the same, one line each	no	low
`git add file`	stage a file	no	low
`git add -p`	stage chosen hunks	no	low
`git add -A`	stage everything	no	low
`git commit -m "msg"`	record the staged snapshot	no	low
`git commit --amend`	replace the last commit	no	med
`git diff`	working tree vs index	no	low
`git diff --staged`	index vs last commit	no	low
`git diff main..feature`	between two branches	no	low
`git rm file`	delete and stage the delete	no	high
`git mv old new`	rename and stage it	no	low
`git restore file`	discard working changes	no	high
`git restore --staged file`	unstage, keep changes	no	low
`git stash`	park changes, clean tree	no	low
`git stash pop`	bring them back	no	low
`git stash list`	see what is parked	no	low
`git clean -n`	show what would be deleted	no	low
`git clean -fd`	delete untracked files	no	high

1. `git restore file` and `git clean -fd` are the two most dangerous commands in daily use. They delete things git has never seen. The reflog cannot help you. Always run `git clean -n` first.

52.7.4 History and inspection

Command	What it does	Net	Risk
`git log`	full history	no	low
`git log --oneline`	one line per commit	no	low
`git log --oneline --graph --all`	the branch shape	no	low
`git log -p`	history with diffs	no	low
`git log -S "text"`	commits that changed text	no	low
`git log --since="2 weeks"`	by date	no	low
`git show SHA`	one commit in full	no	low
`git blame file`	who last touched each line	no	low
`git shortlog -sn`	commits per author	no	low
`git describe --tags`	nearest tag plus distance	no	low
`git rev-list --count HEAD`	how many commits	no	low
`git bisect start`	begin a binary search	no	low
`git bisect good SHA`	mark a known-good point	no	low
`git bisect bad`	mark the current as bad	no	low
`git bisect reset`	stop and go back	no	low
`git reflog`	every position HEAD held	no	low
`git range-diff a...b`	compare two versions of a series	no	low

1. `git reflog` is the undo button for almost everything. If a rebase or reset went wrong, find the old SHA in the reflog and `git reset --hard` to it. Entries expire after 90 days for reachable commits and 30 days for unreachable ones, by default.

52.7.5 Branching

Command	What it does	Net	Risk
`git branch`	list local branches	no	low
`git branch -a`	include remote-tracking	no	low
`git branch -vv`	show tracking and ahead/behind	no	low
`git switch name`	move to a branch	no	low
`git switch -c name`	create and move to it	no	low
`git switch -`	back to the previous branch	no	low
`git checkout name`	the older form of switch	no	low
`git branch -d name`	delete a merged branch	no	low
`git branch -D name`	delete regardless	no	med
`git merge feature`	join a branch in	no	low
`git merge --abort`	undo a conflicted merge	no	low
`git merge --no-ff`	always make a merge commit	no	low
`git rebase main`	replay commits on a new base	no	med
`git rebase -i HEAD~5`	edit the last five commits	no	med
`git rebase --abort`	undo an in-progress rebase	no	low
`git cherry-pick SHA`	copy one commit here	no	low
`git merge-base a b`	the common ancestor	no	low
`git tag -a v1.0 -m "msg"`	make an annotated tag	no	low
`git worktree add ../dir br`	second checkout, same repo	no	low

1. The one fact about rebase that matters. Rebase does not move commits. It creates new commit objects with the same changes and new parents, so every rebased commit gets a new SHA.
2. That is exactly what the reader saw: a local branch rebased onto a moved `main` came out with a different SHA, because it was a different object.
3. Merge preserves history and adds a commit. Rebase rewrites history and adds none. Neither is correct in general; teams pick one by convention.

52.7.6 Remotes and the network

Command	What it does	Net	Risk
`git fetch`	download objects and refs	yes	low
`git fetch --all --prune`	all remotes, drop dead refs	yes	low
`git pull`	fetch then merge	yes	low
`git pull --rebase`	fetch then rebase	yes	med
`git push`	send commits to a remote	yes	low
`git push -u origin br`	push and set upstream	yes	low
`git push --force`	overwrite the remote branch	yes	high
`git push --force-with-lease`	force, but refuse if it moved	yes	med
`git push --tags`	send tags too	yes	low
`git ls-remote origin`	list remote refs, live	yes	low
`git remote show origin`	remote details	yes	low

Command	What it does	Net	Risk
`git remote update`	fetch all remotes	yes	low
`git remote prune origin`	delete stale tracking refs	yes	low
`git bundle create f.bundle --all`	history in one file	no	low
`git archive HEAD -o src.tar`	export files, no history	no	low
`git submodule update --init`	fetch nested repos	yes	low

1. Only the rows marked yes need the network. Everything in 52.7.3, 52.7.4, 52.7.5 and 52.7.7 works offline, because the whole history is already in `.git`.
2. That is why the reader could keep committing to branches during a total outage and push them later. Commits are local writes to local files.
3. `git ls-remote origin` is the only way to learn the true current state of a remote. `git rev-parse origin/main` reads a cached value from your last fetch.
4. In the reader's session, `origin/main` showed `0a95cc8` during the outage. That was the remote-tracking ref, which is a note about the past, not a live fact.
5. Never use `--force` on a shared branch. Use `--force-with-lease`, which checks that the remote is still where you last saw it and refuses if somebody else pushed in the meantime.

52.7.7 Undoing

Command	What it does	Net	Risk
`git restore --staged file`	unstage, keep the change	no	low
`git restore file`	throw away the change	no	high
`git reset --soft HEAD~1`	undo commit, keep staged	no	med
`git reset --mixed HEAD~1`	undo commit, keep in tree	no	med
`git reset --hard HEAD~1`	undo commit, delete changes	no	high
`git reset --hard origin/main`	make local match remote	no	high
`git revert SHA`	new commit that undoes one	no	low
`git commit --amend`	fix the last commit	no	med
`git reflog` then `reset --hard`	recover a lost position	no	med
`git checkout SHA -- file`	one file from a past commit	no	high
`git clean -fdx`	delete untracked and ignored	no	high

1. The rule that keeps you safe: `revert` is for history that other people have. `reset` is for history only you have.
2. `--soft` keeps everything staged. `--mixed`, the default, keeps changes in the working tree but unstaged. `--hard` deletes them.

52.7.8 Plumbing, for looking inside

Command	What it does	Net	Risk
`git cat-file -t SHA`	the object's type	no	low
`git cat-file -p SHA`	the object's contents	no	low
`git cat-file -s SHA`	the object's size	no	low
`git hash-object -w file`	store a blob, print its SHA	no	low
`git rev-parse HEAD`	resolve a name to a SHA	no	low
`git rev-parse --short HEAD`	the short form	no	low
`git rev-parse --git-dir`	where .git actually is	no	low
`git ls-tree HEAD`	one tree object listed	no	low
`git ls-files -s`	the index, with SHAs	no	low
`git write-tree`	index becomes a tree object	no	low
`git commit-tree tree -p par`	build a commit by hand	no	low
`git update-ref refs/heads/x SHA`	point a ref somewhere	no	med
`git symbolic-ref HEAD`	what HEAD points at	no	low
`git show-ref`	every ref and its SHA	no	low
`git for-each-ref`	refs, formatted	no	low
`git fsck --full`	check every object	no	low
`git gc`	pack and prune	no	low
`git count-objects -v`	how many objects, how big	no	low
`git verify-pack -v file.idx`	inside a pack file	no	low
`git check-ignore -v file`	which rule ignores it	no	low

1. The four object types, seen with `git cat-file -t`: blob is file contents, tree is a directory listing, commit is a snapshot plus metadata, tag is an annotated tag.
2. Git stores content addressed by hash. Git 2.29, released October 2020, added experimental SHA-256 repositories; SHA-1 remains the default in 2026.

52.8 Decision tree: my git operation failed

52.8.1 The five families

1. Every git failure belongs to one of five families. Naming the family is 90 per cent of the fix.

Family	Where it happens	Retry safe
Transport	before or during the bytes	usually
Authentication	who are you	after fixing
Permission or scope	you, but not allowed	after fixing
Server-side rejection	server said no, on purpose	after fixing
Local	never left your machine	yes

```
git command failed. Read the FIRST error line.
              |
Does it mention a host, port, TLS,
timeout, disconnect, or reset?
   yes -> TRANSPORT.  See 52.8.3
   no  v
Does it say Authentication failed,
Permission denied (publickey),
or ask for a password repeatedly?
```

```
      yes -> AUTHENTICATION.  See 52.8.4
      no  v
   Does it name a scope, a permission,
   or say 403 / not authorized?
      yes -> SCOPE.  See 52.8.5
      no  v
   Does it say rejected, non-fast-forward,
   protected branch, hook declined, or
   pre-receive?
      yes -> SERVER REJECTION.  See 52.8.6
      no  -> LOCAL.  See 52.8.7
```

52.8.2 First, always: did it actually happen

1. Before diagnosing anything, find out the current true state.
2. Run `git ls-remote origin`. That is a live query. It is the only reliable answer.
3. Compare with `git rev-parse HEAD` locally.
4. Do not trust `git rev-parse origin/main` or `git status` after a failure. Those read remote-tracking refs, which are a cached memory of your last successful fetch.
5. This step is not optional. Half of all git panic is caused by acting on a stale local belief about the remote.

52.8.3 Transport failures

1. Symptoms: `Could not resolve host`, `Failed to connect`, `Connection timed out`, `Connection reset by peer`, `SSL_ERROR_SYSCALL`, `RPC failed`, `early EOF`, `unexpected disconnect while reading sideband packet`.
2. Diagnostics, in order:

```
git ls-remote origin              # does anything work at all
ssh -T git@github.com             # SSH transport, port 22
nc -vz github.com 22              # is port 22 even reachable
nc -vz github.com 443             # is port 443 reachable
curl -sS -o /dev/null -w '%{http_code}\n' \
  https://github.com
GIT_CURL_VERBOSE=1 git fetch      # verbose HTTPS
GIT_TRACE=1 GIT_TRACE_PACKET=1 git fetch
```

3. Retry: usually safe for fetch, clone and pull, because they only read.
4. Retry for push: safe only after you have re-queried with `git ls-remote` and know whether the previous attempt landed.
5. Common fixes. If port 22 is blocked but 443 is not, use SSH over 443 by putting this in `~/.ssh/config`:

```
Host github.com
  Hostname ssh.github.com
  Port 443
  User git
```

6. If large pushes fail but small ones work, raise the buffer: `git config --global http.postBuffer 524288000`. This is a workaround, not a fix, and it helps only with HTTP transport.
7. If HTTPS fails and SSH works, or the reverse, that is not a git problem at all. It is two different ports to two different endpoints, and one path is broken. The reader saw exactly this in the same session.

52.8.4 Worked branch: sideband packet disconnect

1. The exact error, seen twice in the reader's session: `send-pack: unexpected disconnect while reading sideband packet`.
2. What it means, literally. Git pushed the pack of objects, then waited on the sideband channel for the server's progress and status messages, and the connection died while it was reading them.

3. Now the important part. The disconnect happened while reading the **response**. It says nothing about whether the **write** was applied.
4. Three outcomes are all consistent with that one message. The server never received the pack. The server received it and rejected it. The server received it, applied it, and the acknowledgement was lost on the way back.
5. This is the same problem as a POST over HTTP that times out. The request is not idempotent, so the client cannot safely assume anything.
6. The only correct next step is to ask the server what it now believes:

```
git ls-remote origin
git ls-remote origin refs/heads/main
git rev-parse HEAD
```

7. If the remote SHA equals your local SHA, the push landed. Do nothing.
8. If it does not, push again. A repeated push of the same commits is harmless, because git will only send objects the server lacks.
9. In the reader's case, both times the operation had **not** been applied. That was established by re-querying, not by reading the error.
10. Retry safety: safe to retry, but only after checking. Never force-push as a response to this error. Force is how a lost acknowledgement turns into lost commits.

52.8.5 Authentication and scope failures

1. Authentication symptoms: `Permission denied (publickey)`, `Authentication failed`, repeated password prompts, `Invalid username or password`.

```
ssh -vT git@github.com            # verbose SSH handshake
ssh-add -l                        # keys the agent holds
ssh-add ~/.ssh/id_ed25519         # load a key
git config --get remote.origin.url
git credential-osxkeychain erase  # macOS, then type host lines
```

2. GitHub stopped accepting account passwords for git over HTTPS on 13 August 2021. If something is prompting you for a password, it wants a personal access token, not your password.
3. Scope symptoms are different and more specific. The push authenticates fine and is then refused with a named reason.
4. Worked branch, from the reader's session. A push was refused because the token lacked the `workflow` scope, and the push modified a file under `.github/workflows/`.
5. That is not a bug. It is a deliberate rule: a token that can push code should not be able to change what runs automatically in CI, because CI has its own secrets and its own privileges.
6. The diagnosis is the error text plus one question: does this push touch a workflow file? Check with `git diff --stat origin/main..HEAD` and look for `.github/workflows/`.
7. The fix is to add the `workflow` scope to the token, or to move the workflow change into a separate push made with a token that has it.
8. Retry safety: retrying with the same token will fail identically, every time. Nothing was applied, because the server refused before applying.
9. This is a clean example of the difference between 401 and 403 thinking. You were identified successfully. You were then found to lack a permission.
10. The general rule for tokens: check three things, in this order. Is it expired. Does it cover this repository. Does it have the scope this specific action needs.

52.8.6 Server-side rejections

1. Symptoms: `! [rejected]`, `non-fast-forward`, `Updates were rejected because the remote contains work that you do not have locally`, `protected branch`, `pre-receive hook declined`, `GH006`.
2. A non-fast-forward rejection means somebody pushed since your last fetch. The server is protecting their work.

```
git fetch origin
git log --oneline HEAD..origin/main   # what they added
git log --oneline origin/main..HEAD   # what you added
git rebase origin/main                # or: git merge origin/main
git push
```

3. A protected-branch rejection means the repository's rules forbid a direct push. Open a pull request instead. This is a setting, not an error.
4. A `pre-receive hook declined` means the server ran a script that said no. The reason is usually in the lines just after. Common causes are commit message format rules, file size limits, and secret scanning.
5. GitHub blocks any single file over 100 MiB, and warns above 50 MiB. That limit has been in place since 2015 and is a service policy, not a git limit. Use Git Large File Storage for big files.
6. Retry safety: retrying unchanged always fails again. Nothing was applied. Fix the cause first.

52.8.7 Local failures

1. Symptoms: `not a git repository`, `pathspec did not match any file`, `your local changes would be overwritten`, `You have unmerged paths`, `index.lock exists`, `detached HEAD`.

```
git rev-parse --git-dir           # am I in a repo at all
git status                        # the honest current state
git diff --name-only --diff-filter=U   # conflicted files
rm -f .git/index.lock             # only if no git is running
git fsck --full                   # object integrity
git switch -                      # leave detached HEAD safely
```

2. `index.lock exists` almost always means a previous git process died or an editor's git integration is running. Check with `ps aux | grep git` before deleting the lock file.
3. `detached HEAD` is not an error. It means HEAD points at a commit rather than a branch. Commits made there are reachable only from the reflog, so create a branch before you go further: `git switch -c rescue`.
4. Retry safety: always safe. Nothing left your machine.
5. Finally, one distinction that matters and that the reader's session showed clearly. Git the tool and GitHub the service are different things. During the outage plain git kept working locally, and the GitHub web API did not. An error from `gh` or from the API is not a git error.

52.8.8 Verifying CI properly

1. A green tick next to a branch name means "some run for something on this branch passed". It does not say which commit.
2. The strong check is by run identifier against the exact head SHA:

```
git rev-parse HEAD
gh run list --branch main --limit 5
gh run view <run-id> --json headSha,status,conclusion
```

3. Compare the `headSha` field with your local SHA character by character. If they differ, the tick is about a different commit.
4. Merge state, seen through the API, uses values such as `CLEAN`, `BLOCKED` and `UNKNOWN`. `UNKNOWN` usually means the server has not finished computing the merge yet; re-query rather than assuming a failure.

52.9 Shell and system cheat sheet

52.9.1 Files and directories

Command	What it does
`pwd`	print the current directory
`cd -`	go back to the previous one
`ls -lah`	long, all, human sizes
`ls -lt`	newest first
`ls -lS`	biggest first
`mkdir -p a/b/c`	make the whole path
`cp -r src dst`	copy a directory
`cp -a src dst`	copy, keep all attributes
`mv old new`	move or rename
`rm -i file`	delete, asking first
`rm -rf dir`	delete a tree, no questions
`ln -s target link`	make a symbolic link
`touch file`	create empty, or update time
`stat file`	size, times, permissions
`file thing`	guess the format
`du -sh dir`	total size of a directory
`du -sh * | sort -h`	biggest items first
`df -h`	free space per filesystem
`basename path`	last part of a path
`dirname path`	everything but the last part

52.9.2 Looking inside files

Command	What it does
`cat file`	print the whole file
`less file`	page through it, q to quit
`head -n 20 file`	first 20 lines
`tail -n 20 file`	last 20 lines
`tail -f log`	follow as it grows
`wc -l file`	count lines
`sort file`	sort lines
`sort -u`	sort and remove duplicates
`uniq -c`	count runs, needs sort first
`cut -d, -f2 file`	second comma field
`tr 'a-z' 'A-Z'`	change characters
`sed 's/old/new/g'`	replace text
`awk '{print $3}'`	print the third field
`grep -n pattern file`	matching lines, numbered
`grep -ri pattern dir`	recursive, ignore case
`grep -v pattern`	lines that do not match
`grep -c pattern`	just the count
`diff -u a b`	unified difference
`xxd file | head`	hex dump
`jq '.field' file.json`	read JSON properly

52.9.3 Finding things

```
find . -name '*.log'              # by name
find . -type f -size +100M        # files over 100 MB
find . -mtime -1                  # changed in the last day
find . -name '*.tmp' -delete      # find and delete
find . -type f -exec grep -l x {} +   # search inside
which python3                     # which one runs
type -a python3                   # every match, and aliases
command -v python3                # portable form
locate filename                   # from an index, if built
```

52.9.4 Processes and resources

Command	What it does
`ps aux`	every process
`ps aux | grep name`	find one
`top`	live view
`htop`	nicer live view, install it
`kill PID`	ask politely, signal 15
`kill -9 PID`	force, signal 9
`pkill -f pattern`	kill by command line
`jobs`	this shell's background jobs
`bg` and `fg`	move a job
`nohup cmd &`	keep running after logout
`time cmd`	how long it took
`uptime`	load averages
`free -h`	memory, Linux only
`vm_stat`	memory, macOS
`sysctl -n hw.ncpu`	core count, macOS
`nproc`	core count, Linux
`uname -a`	kernel and machine
`sw_vers`	macOS version
`lsb_release -a`	Linux distribution

1. Signal 15, SIGTERM, asks a program to shut down cleanly. Signal 9, SIGKILL, cannot be caught or ignored and gives the program no chance to save. Try 15 first, always.

52.9.5 Archives, permissions and transfer

```
tar -czf out.tar.gz dir/     # create, gzip
tar -xzf in.tar.gz           # extract
tar -tzf in.tar.gz           # list without extracting
zip -r out.zip dir/          # create a zip
unzip in.zip                 # extract a zip
gzip file                    # replaces file with file.gz
gunzip file.gz               # and back
chmod 644 file               # rw for owner, r for others
chmod 755 script             # add execute for everyone
chmod +x script              # just add execute
chown user:group file        # change ownership
umask                        # default permission mask
rsync -avz src/ host:dst/    # copy, resume, compress
scp file host:/path          # simple copy over SSH
ssh -i key user@host         # connect with a key
```

1. Permission digits are octal. Read is 4, write is 2, execute is 1. Add them. The three digits are owner, group, others. So 755 is read-write-execute for the owner and read-execute for everyone else.

52.9.6 Keyboard shortcuts in the shell

Keys	What it does
Ctrl-C	interrupt the running command
Ctrl-D	end of input, or log out
Ctrl-Z	suspend, resume with `fg`
Ctrl-L	clear the screen
Ctrl-A	jump to start of line
Ctrl-E	jump to end of line
Ctrl-U	delete to start of line
Ctrl-K	delete to end of line
Ctrl-W	delete the word before
Ctrl-Y	paste what you just deleted
Alt-B, Alt-F	back or forward one word
Ctrl-R	search command history
Ctrl-G	leave the history search
Up, Down	previous, next command
Tab	complete a name
Tab Tab	show all completions
`!!`	the previous command
`!$`	last argument of it
`sudo !!`	rerun the last one as root

1. On macOS Terminal, Alt is the Option key, and you may need to enable "Use Option as Meta key" in the settings before Alt-B and Alt-F work.

52.9.7 The shell expansion order

1. The shell rewrites your command line in a fixed order before running anything. Knowing the order explains almost every strange quoting bug.

```
1. brace expansion        {a,b}  ->  a b
2. tilde expansion        ~      ->  /Users/you
3. parameter expansion    $VAR   ->  its value
4. arithmetic expansion   $((2+2)) -> 4
5. command substitution   $(cmd) ->  its output
6. word splitting         on spaces, tabs, newlines
7. pathname expansion     *.txt  ->  matching files
8. quote removal          "x"    ->  x
```

2. The two consequences that bite people.
3. Word splitting happens **after** variable expansion. So an unquoted `$FILE` holding `my report.txt` becomes two arguments. Quoting it stops that.
4. Pathname expansion happens after variable expansion too, so a variable containing `*` will expand to filenames unless you quote it.

52.9.8 The safety rules

1. **Quote every variable.** Write `"$FILE"`, not `$FILE`. Write `"$@"`, not `$@`. This one habit prevents most shell bugs.
2. **Start scripts with strict mode.**

```
#!/usr/bin/env bash
set -euo pipefail
IFS=$'\n\t'
```

3. `-e` exits on the first failing command. `-u` treats an unset variable as an error. `-o pipefail` makes a pipeline fail if any stage fails, not just the last. Setting IFS stops word splitting on plain spaces.
4. **Never pipe an unknown URL into a shell.** `curl ... | sh` hands a remote server the ability to run anything as you. Download first, read it, then run it.
5. There is a further reason beyond trust: a server can serve different bytes to `curl` piped into a shell than to `curl` writing a file, because it can see the download stall as the shell executes. Reading the file you downloaded is the only real protection.
6. **Test destructive commands first.** Use `git clean -n`, `rm -i`, `rsync --dry-run`, and `find ... -print` before `find ... -delete`.
7. **Never run `rm -rf` with a variable that might be empty.** If `$DIR` is unset, `rm -rf "$DIR"/` becomes `rm -rf /`. `set -u` prevents this.
8. **Use sudo for one command, not for a session.** A root shell left open is a mistake waiting for a typing error.
9. **Prefer absolute paths in scripts and cron jobs.** The working directory and the `PATH` in an automated context are not the ones you have interactively.

52.10 File format identification by magic number

1. A magic number is a fixed byte sequence at a known offset that identifies a file's format. The extension is a hint; the magic number is the fact.
2. Check with `file thing` for a guess, or `xxd file | head -2` for the bytes.

Format	Hex bytes at start	Extension
PNG image	89 50 4E 47 0D 0A 1A 0A	.png
JPEG image	FF D8 FF	.jpg
GIF image	47 49 46 38	.gif
BMP image	42 4D	.bmp
TIFF, little-endian	49 49 2A 00	.tif
TIFF, big-endian	4D 4D 00 2A	.tif
WebP	52 49 46 46 then WEBP	.webp
ICO icon	00 00 01 00	.ico
PSD, Photoshop	38 42 50 53	.psd
PDF document	25 50 44 46 2D	.pdf
PostScript	25 21	.ps
RTF document	7B 5C 72 74 66	.rtf
ZIP, and docx, xlsx, jar	50 4B 03 04	.zip
GZIP	1F 8B	.gz
BZIP2	42 5A 68	.bz2
XZ	FD 37 7A 58 5A 00	.xz
Zstandard	28 B5 2F FD	.zst
7-Zip	37 7A BC AF 27 1C	.7z
RAR version 5	52 61 72 21 1A 07 01	.rar
TAR	75 73 74 61 72 at byte 257	.tar
Old Office, OLE2	D0 CF 11 E0 A1 B1 1A E1	.doc
MP3 with ID3 tag	49 44 33	.mp3
FLAC audio	66 4C 61 43	.flac

Format	Hex bytes at start	Extension
OGG container	4F 67 67 53	.ogg
WAV audio	52 49 46 46 then WAVE	.wav
MP4, HEIC, AVIF	66 74 79 70 at byte 4	.mp4
Matroska, WebM	1A 45 DF A3	.mkv
ELF executable	7F 45 4C 46	none
Mach-O 64-bit	CF FA ED FE	none
Windows PE, EXE	4D 5A	.exe
Java class file	CA FE BA BE	.class
WebAssembly	00 61 73 6D	.wasm
SQLite 3 database	53 51 4C 69 74 65 20 66	.sqlite
Git pack file	50 41 43 4B	.pack
Git index file	44 49 52 43	none
Debian package	21 3C 61 72 63 68 3E	.deb
RPM package	ED AB EE DB	.rpm
Shell script	23 21	.sh
PEM certificate	2D 2D 2D 2D 2D 42 45 47	.pem
ISO 9660 image	43 44 30 30 31 at 0x8001	.iso
WOFF web font	77 4F 46 46	.woff
OpenType font	4F 54 54 4F	.otf

3. Two entries in that table are the same bytes, and this is a real trap. `CA FE BA BE` is both a Java class file and an old Mach-O universal binary header. The disambiguation is what follows.
4. `50 4B 03 04` covers a very large family. Modern Office files, Java jar files, EPUB books, Android APK files and OpenDocument files are all zip archives with different contents inside.
5. Two other useful facts. `MZ` at the start of a Windows executable stands for Mark Zbikowski, one of the original MS-DOS developers, and has been there since 1983. `7F 45 4C 46` is a non-printable byte followed by the letters `ELF`, deliberately chosen so text tools do not mangle it.

52.11 Complexity and performance reference

52.11.1 Big-O and what fits in one second

1. Big-O describes how the work grows as the input grows. It hides constant factors on purpose.
2. The table assumes a rough budget of 100 million simple operations per second, which is a fair figure for interpreted-to-compiled code across the range. Compiled C can do ten times more; Python often ten times less.

Complexity	Name	Largest n in 1 s
O(1)	constant	any size
O(log n)	logarithmic	effectively unlimited
O(n)	linear	100,000,000
O(n log n)	linearithmic	about 4,500,000
O(n^2)	quadratic	about 10,000
O(n^3)	cubic	about 464
O(2^n)	exponential	about 26
O(n!)	factorial	about 11

3. The practical reading. If n is a million, you need O(n) or O(n log n). If n is ten thousand, O(n^2) is acceptable. If n is twenty, anything works.

4. Constants matter in real code. A cache-friendly O(n^2) can beat a pointer-chasing O(n log n) for small n. Measure before optimizing.

52.11.2 Data structure operation costs

Structure	Lookup	Insert or delete
Array, by index	O(1)	O(n)
Array, by value	O(n)	O(n)
Dynamic array, append	O(1)	O(1) amortized
Linked list	O(n)	O(1) at a known node
Hash table	O(1) average	O(1) average
Hash table, worst case	O(n)	O(n)
Balanced tree	O(log n)	O(log n)
Binary heap, minimum	O(1)	O(log n)
Trie	O(k), k is key length	O(k)
B-tree	O(log n)	O(log n)
Skip list	O(log n) expected	O(log n) expected
Bloom filter	O(k) hashes	insert only

1. A hash table is O(1) on average and O(n) in the worst case, when every key lands in the same bucket. Modern languages randomize the hash seed at startup to stop attackers forcing that deliberately.
2. A B-tree is the shape used by databases and filesystems because its nodes are sized to a disk block, so one comparison level costs one read.
3. A Bloom filter can say "definitely not present" or "probably present". It never gives a false negative and cannot delete.

52.11.3 Sorting algorithms

Algorithm	Average	Worst	Note
Quicksort	O(n log n)	O(n^2)	in place, unstable
Mergesort	O(n log n)	O(n log n)	needs O(n) space, stable
Heapsort	O(n log n)	O(n log n)	in place, unstable
Timsort	O(n log n)	O(n log n)	stable, fast on runs
Introsort	O(n log n)	O(n log n)	quicksort with fallback
Insertion sort	O(n^2)	O(n^2)	O(n) if nearly sorted
Selection sort	O(n^2)	O(n^2)	fewest swaps
Bubble sort	O(n^2)	O(n^2)	teaching only
Counting sort	O(n + k)	O(n + k)	small integer keys
Radix sort	O(n k)	O(n k)	not comparison based

1. **Stable** means two records with equal keys keep their original order. That matters when you sort by one column and then another.
2. Timsort was written by Tim Peters in 2002 for Python. Java adopted it for objects in Java 7, released 2011. It finds runs that are already sorted and merges them, which is why it is O(n) on nearly sorted data.
3. Introsort, by David Musser in 1997, is what most C++ standard libraries use for `std::sort`. It runs quicksort and switches to heapsort if the recursion gets too deep, which removes the O(n^2) worst case.
4. No comparison sort can beat O(n log n) in the worst case. That is a proved lower bound, not an engineering limit. Counting and radix sort beat it only because they do not compare.

52.11.4 The cost of common operations

Operation	Rough cost
Integer add or subtract	under 1 ns
Integer multiply	1 to 3 ns
Integer divide	6 to 13 ns
Float add or multiply	1 to 2 ns
Float divide	4 to 5 ns
Square root	about 5 ns
Function call	1 to 2 ns
Virtual or indirect call	2 to 5 ns
Branch, predicted correctly	under 1 ns
Branch, mispredicted	3 to 5 ns
Uncontended mutex	about 17 ns
Contended atomic increment	50 to 100 ns
Hash a short string	20 to 50 ns
Small malloc and free	50 to 100 ns
A simple system call	100 to 500 ns
Thread context switch	1 to 5 us
Create a thread	10 to 30 us
Fork and exec a process	0.5 to 1 ms
One Python bytecode step	30 to 50 ns
One Python function call	about 100 ns

1. System calls got more expensive in 2018, when the Meltdown and Spectre mitigations forced page table isolation. Depending on the machine and the mitigation set, a simple system call can cost several times what it did before.
2. The gap between a Python operation and a C operation is roughly fifty to a hundred times. That is why numerical Python work is done inside NumPy, where the loop lives in C.

52.12 Cloud and infrastructure reference

52.12.1 Service equivalence

Category	AWS	Azure	Google
Virtual machines	EC2	Virtual Machines	Compute Engine
Object storage	S3	Blob Storage	Cloud Storage
Block storage	EBS	Managed Disks	Persistent Disk
Shared file storage	EFS	Azure Files	Filestore
Archive storage	S3 Glacier	Archive Storage	Archive
Managed relational	RDS	Azure SQL Database	Cloud SQL
Scale-out relational	Aurora	SQL Hyperscale	Spanner
Managed NoSQL	DynamoDB	Cosmos DB	Firestore
Wide-column store	Keyspaces	Cosmos DB	Bigtable
Data warehouse	Redshift	Synapse, Fabric	BigQuery
In-memory cache	ElastiCache	Azure Cache	Memorystore
Serverless functions	Lambda	Functions	Cloud Run functions
Containers, managed	ECS, Fargate	Container Apps	Cloud Run
Kubernetes	EKS	AKS	GKE
Container registry	ECR	Container Registry	Artifact Registry

Category	AWS	Azure	Google
Load balancing	ELB, ALB, NLB	Load Balancer	Cloud Load Balancing
Content delivery	CloudFront	Front Door	Cloud CDN
DNS	Route 53	Azure DNS	Cloud DNS
Private network	VPC	Virtual Network	VPC
Identity	IAM	Entra ID	IAM
Secrets	Secrets Manager	Key Vault	Secret Manager
Message queue	SQS	Service Bus	Pub/Sub
Publish and subscribe	SNS	Event Grid	Pub/Sub
Event streaming	Kinesis	Event Hubs	Pub/Sub
Managed Kafka	MSK	Event Hubs	Managed Kafka
Workflow orchestration	Step Functions	Logic Apps	Workflows
Monitoring and logs	CloudWatch	Azure Monitor	Cloud Monitoring
Infrastructure as code	CloudFormation	Bicep, ARM	Deployment Manager
Machine learning	SageMaker	Azure ML	Vertex AI
Hosted large models	Bedrock	Azure AI Foundry	Vertex AI

1. Names change. Microsoft renamed Azure Active Directory to Microsoft Entra ID in 2023, and has been folding Synapse into Microsoft Fabric since 2023. Google renamed Cloud Functions to Cloud Run functions in 2024.
2. The equivalences are approximate. Spanner and Aurora are not the same thing; both solve "a relational database bigger than one machine" and solve it differently.

52.12.2 The service model responsibility split

Model	You manage	Provider manages
On-premises	everything	nothing
IaaS	OS upward	hardware, network, power
PaaS	code and data	OS, runtime, scaling
Serverless	function and data	everything else
SaaS	your data and users	the whole application

1. One rule survives every model: **you are always responsible for your own data and your own access control**. No provider takes that on.
2. The shared responsibility model is the standard phrase. AWS states it as "security of the cloud" being theirs and "security in the cloud" being yours.
3. The most common real breach in cloud is not a provider failure. It is a storage bucket left open to the public, which is entirely on the customer side of the line.

52.12.3 Instance families and what they are for

Family letter	Optimized for	Use it for
t	burstable, cheap	dev, small sites
m	balanced	general servers
c	CPU per unit cost	encoding, simulation
r	memory	caches, in-memory DBs
x, z	very large memory	SAP, huge datasets
i, d	local NVMe storage	databases, search

Family letter	Optimized for	Use it for
p	GPU for training	model training
g	GPU for inference	inference, graphics
inf, trn	custom AI silicon	inference, training

1. The letters above are the AWS convention. Azure uses series letters such as D for general purpose, F for compute, E and M for memory, and N for GPU. Google uses names such as N2, C3, M3 and A3.
2. A modern name such as `m7g.2xlarge` reads as: family m, generation 7, Graviton Arm processor, size 2xlarge. The suffix `g` for Arm is an AWS convention.
3. Size doubling is a convention too. Each step up in size usually doubles vCPU and memory and doubles the price, so cost per unit of work is roughly flat. That is why splitting across smaller instances rarely saves money by itself.

52.12.4 Cost dimensions to check on any bill

1. Compute time, per hour or per second, per instance.
2. Storage, per gigabyte-month, separately for each class.
3. Data transfer out to the internet, per gigabyte. This is the item that surprises people. Ingress is usually free; egress is not.
4. Data transfer between availability zones and between regions, per gigabyte, in both directions.
5. Request counts. Object storage charges per PUT and per GET, and a workload with many tiny objects can cost more in requests than in storage.
6. Provisioned throughput and IOPS on disks and databases, whether or not you use them.
7. Load balancer hours plus a per-capacity-unit charge.
8. NAT gateway hours plus a per-gigabyte processing charge. This is a common source of unexplained cost in private networks.
9. Public IPv4 addresses. AWS began charging for every public IPv4 address on 1 February 2024, at 0.005 US dollars per hour, which is about 3.60 dollars a month each.
10. Snapshots and backups, which keep growing and which nobody deletes.
11. Log ingestion and log retention, charged separately.
12. Idle waste: unattached volumes, stopped instances still holding disks, old snapshots, forgotten test environments.
13. Support plan, usually a percentage of the total spend.
14. Commitment discounts. Reserved instances and savings plans trade flexibility for a lower rate. Spot instances are much cheaper and can be taken away with two minutes of notice.

52.13 Security reference

52.13.1 Hash functions, as of 2026

Algorithm	Verdict	Note
MD5	broken	collisions since 2004
SHA-1	broken	collision proved 2017
SHA-256	use	the default choice
SHA-384, SHA-512	use	wider, often faster on 64-bit
SHA-3 family	use	different internal design
BLAKE2, BLAKE3	use	very fast, not FIPS
CRC32	not a hash	error detection only

1. MD5 was broken by Xiaoyun Wang and colleagues in 2004. SHA-1 was broken in practice on 23 February 2017, when Google and CWI Amsterdam published two different PDF files with the same SHA-1 digest, an attack they named SHAttered. A chosen-prefix collision followed in January 2020.
2. Neither is safe for signatures or integrity against an attacker. Both are still acceptable as non-security checksums, and git still uses SHA-1 with an added collision-detection check.

52.13.2 Ciphers and key exchange, as of 2026

Purpose	Use	Avoid
Symmetric encryption	AES-256-GCM	DES, 3DES, RC4
Symmetric, no AES hardware	ChaCha20-Poly1305	AES in ECB mode
Mode of operation	GCM, or GCM-SIV	CBC without a MAC
Key exchange	X25519, ECDHE	static RSA, small DH
Signature	Ed25519, ECDSA P-256	DSA, RSA-1024
RSA, if you must	3072 bits or more	1024, 2048 for new keys
SSH host and user keys	Ed25519	DSA, RSA-1024

1. NIST disallowed three-key Triple DES for new encryption after 31 December 2023. RC4 was prohibited in TLS by RFC 7465 in February 2015.
2. Post-quantum standards arrived on 13 August 2024, when NIST published FIPS 203 for ML-KEM, derived from Kyber, FIPS 204 for ML-DSA, derived from Dilithium, and FIPS 205 for SLH-DSA, derived from SPHINCS+.
3. Deployment is hybrid: browsers and servers have been negotiating X25519 combined with ML-KEM-768 since 2024, so a future quantum computer cannot decrypt traffic captured today. This is established practice now, not a research claim.

52.13.3 Password storage parameters

1. Never store a password. Store a slow, salted hash of it. These are the OWASP recommended minimums.

Algorithm	Parameters
Argon2id, preferred	m=19456 KiB, t=2, p=1
Argon2id, alternative	m=47104 KiB, t=1, p=1
Argon2id, alternative	m=12288 KiB, t=3, p=1
scrypt	N=2^17, r=8, p=1
scrypt, alternative	N=2^16, r=8, p=2
bcrypt	work factor 10 or more
PBKDF2-HMAC-SHA256	600,000 iterations
PBKDF2-HMAC-SHA512	220,000 iterations

2. `m` is memory in kibibytes, `t` is the number of passes, `p` is parallelism. Memory cost is the point: it is what makes custom hardware attacks expensive.
3. bcrypt silently ignores everything past 72 bytes of input. Enforce a maximum length or pre-hash. The work factor is a power of two, so 12 is four times the work of 10.
4. Always use a random salt per password, at least 16 bytes. A pepper, a secret stored outside the database, is optional and adds real value.
5. Plain SHA-256 of a password is not acceptable, with or without a salt. It is far too fast: modern hardware computes billions per second.

52.13.4 TLS configuration essentials

1. Enable TLS 1.3 and TLS 1.2. Disable everything older. TLS 1.0 and 1.1 were formally deprecated by RFC 8996 in March 2021.
2. TLS 1.3, standardized in RFC 8446 in August 2018, has only five cipher suites and all of them are safe. The three you will see are `TLS_AES_128_GCM_SHA256`, `TLS_AES_256_GCM_SHA384` and `TLS_CHACHA20_POLY1305_SHA256`.
3. For TLS 1.2, allow only suites beginning `ECDHE_` with `GCM` or `CHACHA20`. That gives forward secrecy and authenticated encryption.
4. Turn on OCSP stapling so clients do not have to contact the certificate authority themselves.
5. Send the HTTP Strict Transport Security header with a long lifetime: `Strict-Transport-Security: max-age=31536000; includeSubDomains`.
6. Disable TLS compression and client-initiated renegotiation. The CRIME attack of 2012 used compression to recover session cookies.
7. Publish a CAA DNS record naming which certificate authorities may issue for your domain.
8. Watch expiry. The CA/Browser Forum agreed in April 2025 to reduce the maximum certificate lifetime in stages, reaching 47 days by March 2029. Automated renewal stops being optional.

52.13.5 The OWASP Top 10, 2025 release

Code	Category
A01	Broken Access Control
A02	Security Misconfiguration
A03	Software Supply Chain Failures
A04	Cryptographic Failures
A05	Injection
A06	Insecure Design
A07	Authentication Failures
A08	Software or Data Integrity Failures
A09	Security Logging and Alerting Failures
A10	Mishandling of Exceptional Conditions

1. This list replaced the 2021 edition. Two changes are worth knowing. Supply chain failures moved up to third place, reflecting attacks on dependencies and build systems rather than on the application itself. Mishandling of exceptional conditions is new at tenth: error handling that leaks information or fails open.
2. The list is an awareness document, ordered by a mix of measured incidence and survey. It is not a checklist you can pass.

52.13.6 A personal security checklist

1. Use a password manager and a different password for every site. This one change removes the largest category of real-world account loss.
2. Turn on two-factor authentication everywhere it is offered. Prefer a passkey or a hardware security key, then a time-based code app, and use SMS only if nothing else is available. SMS codes can be stolen by taking over the phone number.
3. Turn on full disk encryption. On macOS that is FileVault, in System Settings under Privacy and Security.
4. Let the operating system and browser update themselves.
5. Use a non-administrator account for daily work where you can.
6. Back up on the 3-2-1 rule: three copies, on two kinds of media, one of them off-site. Test a restore at least once; an untested backup is a guess.
7. Lock the screen when you walk away.

8. Review which applications have access to your accounts. On GitHub, check authorized OAuth applications and personal access tokens, and delete the ones you do not recognize.
9. Give every token the smallest scope that works, and an expiry date. The reader's `workflow` scope refusal is the system working correctly.
10. Check your email addresses against public breach databases once a year.
11. Do not install browser extensions you do not need. An extension can read everything you see on every page.
12. Be suspicious of urgency. Almost every successful phishing message manufactures a deadline.

52.14 AI and machine learning reference

52.14.1 The memory formula

1. For inference, the base memory is simple: **bytes = number of parameters times bytes per parameter**.

Precision	Bytes per parameter	Note
FP32	4	full precision
FP16 or BF16	2	standard for inference
FP8	1	newer hardware only
INT8	1	common quantization
INT4 or NF4	0.5	aggressive quantization

2. Add roughly 20 per cent on top for the runtime, activations and fragmentation.
3. Then add the key-value cache, which grows with the conversation: **KV bytes = 2 times layers times kv-heads times head-dim times sequence length times batch size times bytes per element.**
4. The leading 2 is because there is one key tensor and one value tensor.
5. For training with the Adam optimizer in mixed precision, budget roughly 16 to 20 bytes per parameter: 4 for the master weights, 4 for momentum, 4 for variance, 2 for the working copy, 2 for gradients, plus activations.
6. So a 7 billion parameter model needs about 14 GB to run at 16-bit and about 112 GB or more to train the same way. That gap is why fine-tuning methods such as LoRA exist.

52.14.2 Model size against hardware

Parameters	4-bit	8-bit	16-bit
1 B	0.5 GB	1 GB	2 GB
3 B	1.5 GB	3 GB	6 GB
7 B	3.5 GB	7 GB	14 GB
8 B	4 GB	8 GB	16 GB
13 B	6.5 GB	13 GB	26 GB
32 B	16 GB	32 GB	64 GB
70 B	35 GB	70 GB	140 GB
405 B	203 GB	405 GB	810 GB

1. Practical readings for 2026 hardware. A laptop with 16 GB of unified memory runs a 7B or 8B model comfortably at 4-bit. A 24 GB consumer GPU runs 13B at 8-bit or 32B at 4-bit. A 70B model at 16-bit needs two 80 GB accelerators. A 405B model needs a multi-node cluster.
2. Quantization costs quality, and the cost is not linear. Going from 16-bit to 8-bit is usually almost free. Going to 4-bit is usually acceptable. Going below 4-bit degrades noticeably, though the frontier moves.

52.14.3 Token cost rules of thumb

Rule	Figure
1 token, English	about 4 characters
1 token, English	about 0.75 words
1,000 words English	about 1,300 tokens
1 page of prose	about 500 tokens
Source code	about 3 chars per token
Hindi in Devanagari	3 to 6 times English
A 300-page book	about 170,000 tokens

1. Non-Latin scripts cost more because the tokenizer was trained mostly on English text, so it has fewer merged pieces for them. This is a measured property of specific tokenizers, not a law.
2. Cost scales with tokens in plus tokens out, and output tokens are usually priced several times higher than input tokens.
3. Attention cost grows with the square of the sequence length in the classical transformer, so doubling the context roughly quadruples that part of the compute.

52.14.4 Common hyperparameters

Name	What it controls
Learning rate	step size when updating weights
Batch size	examples per update
Epochs	passes over the whole dataset
Warmup steps	ramp-up before full learning rate
Weight decay	penalty that keeps weights small
Dropout	fraction of units randomly ignored
Gradient accumulation	fake a bigger batch on small memory
LoRA rank	size of the small trained adapter
Temperature	randomness when sampling output
Top-p, nucleus	keep tokens summing to probability p
Top-k	keep only the k most likely tokens
Repetition penalty	discourage repeating tokens
Context length	how many tokens the model can see
Max new tokens	how long the answer may be

1. Temperature 0 means always take the most likely token, which makes output nearly deterministic. Temperature 1 samples from the model's own distribution. Above about 1.2 the output usually becomes incoherent.
2. Learning rate is the hyperparameter that matters most. Too high and training diverges; too low and it never gets anywhere.

52.14.5 How to read a model announcement

1. Separate the three kinds of claim before you read anything else.
2. **Established fact**: parameter count, context length, licence, price per million tokens, whether weights can be downloaded, release date.
3. **Active research**: benchmark scores, reasoning ability, agentic behaviour, long-context quality. These are measured, but the measurements are contested and move quickly.
4. **Marketing claim**: "state of the art", "PhD-level", "reasoning", "understands". These words have no agreed definition.

5. Then ask these ten questions.
6. Which exact benchmark, which version, and which prompting setup. A score with chain-of-thought and 64 samples is not comparable to a single-shot score.
7. Was the benchmark possibly in the training data. Contamination is common and usually undetectable from outside.
8. Is this a base model or an instruction-tuned model. They are not interchangeable.
9. What is the licence. "Open weights" means you can download the numbers. "Open source" properly means you also get the training data and code, which is rare.
10. What is the real context window, and is there an independent measurement of quality at the far end of it. Advertised context and usable context often differ.
11. What is the throughput in tokens per second, and at what batch size.
12. What is the price per million input tokens and per million output tokens.
13. Who ran the evaluation. First-party numbers should be treated as a claim until someone independent repeats them.
14. What is the cut-off date of the training data.
15. What is not mentioned. Announcements omit the benchmarks where the model did badly, and that omission is information.

52.15 Regular expressions

52.15.1 The syntax table

Pattern	Means	Example match
`.`	any one character	`a.c` matches `abc`
`^`	start of string or line	`^Hi`
`$`	end of string or line	`end$`
`*`	zero or more of the last	`ab*` matches `a`
`+`	one or more of the last	`ab+` matches `ab`
`?`	zero or one of the last	`colou?r`
`{3}`	exactly three	`a{3}` matches `aaa`
`{2,4}`	two to four	`a{2,4}`
`{2,}`	two or more	`a{2,}`
`*?`, `+?`	lazy, match as few as possible	`<.+?>`
`[abc]`	any one listed character	`[aeiou]`
`[^abc]`	any character not listed	`[^0-9]`
`[a-z]`	a range	`[A-Za-z]`
`\d`	a digit	same as `[0-9]`
`\D`	not a digit	
`\w`	word character	letters, digits, underscore
`\W`	not a word character	
`\s`	whitespace	space, tab, newline
`\S`	not whitespace	
`\b`	word boundary	`\bcat\b`
`\B`	not a word boundary	
`|`	either side	`cat|dog`
`( )`	group and capture	`(ab)+`
`(?: )`	group without capturing	`(?:ab)+`
`(?<name> )`	named capture group	
`\1`	what group 1 matched	`(\w)\1` finds doubles
`(?= )`	followed by, lookahead	`foo(?=bar)`

Pattern	Means	Example match
`(?! )`	not followed by	`foo(?!bar)`
`(?<= )`	preceded by, lookbehind	`(?<=Rs)\d+`
`(?<! )`	not preceded by	
`\.`	a literal dot	escapes the dot
`(?i)`	ignore case from here	

1. Flags are set outside the pattern in most languages: `i` ignore case, `m` multi-line so `^` and `$` match at each line, `s` so `.` also matches a newline, `g` find all matches.
2. Two warnings. Nested quantifiers such as `(a+)+` can take exponential time on a failing input, which is a real denial-of-service class called catastrophic backtracking. And you cannot parse HTML with a regular expression; use a parser.

52.15.2 Five patterns worth keeping

1. Match an IPv4 address, roughly:

```
\b(?:\d{1,3}\.){3}\d{1,3}\b
```

2. It finds four groups of one to three digits separated by dots. It will also match `999.999.999.999`, so validate the ranges afterwards in code. A regular expression that fully validates IPv4 is long and unreadable, and checking each number is under 256 in code is clearer.
3. Find the start of every log line with a timestamp:

```
^\d{4}-\d{2}-\d{2}[ T]\d{2}:\d{2}:\d{2}
```

4. This matches both `2026-08-13 14:05:09` and the ISO 8601 form with a `T`. Use it with `grep -E` to pick out real log lines and drop stack-trace continuation lines.
5. Match a quoted string, without running past the closing quote:

```
"(?:[^"\\]|\\.)*"
```

6. The trick is `[^"\\]`, which means any character that is not a quote and not a backslash, alternated with `\\.`, which means a backslash followed by anything. That handles escaped quotes inside the string.
7. Find a possible secret in a diff before you commit it:

```
(?i)(api[_-]?key|secret|token|password)\s*[:=]\s*\S+
```

8. Run it with `git diff --staged | grep -E` before every commit. It produces false positives, which is the correct trade-off for this job.
9. Capture the status code and byte count from a common log format line:

```
"\s(\d{3})\s(\d+|-)$
```

10. Group 1 is the HTTP status, group 2 is the response size or a dash. Pipe it through `awk` or `sed -E` to build a count by status code.

52.16 Keyboard shortcuts and editor basics

52.16.1 Terminal

Keys	What it does
Cmd-T	new tab, macOS Terminal
Cmd-N	new window, macOS

Keys	What it does
Cmd-K	clear scrollback, macOS
Ctrl-Shift-T	new tab, Linux terminals
Ctrl-Shift-C, Ctrl-Shift-V	copy, paste, Linux
Ctrl-A then D	detach from a tmux session
Ctrl-A then C	new tmux window
Ctrl-A then N	next tmux window

1. The tmux prefix is Ctrl-B by default; Ctrl-A is a very common reconfiguration and a convention rather than a default.

52.16.2 Shell line editing

1. The shortcuts in section 52.9.6 are Emacs-style bindings, which is the default in bash and zsh.
2. If you prefer vi bindings, run `set -o vi`. Then Escape puts you in command mode and the vim movement keys work on the command line.
3. `Ctrl-R` then typing searches history backwards. Press `Ctrl-R` again to go further back. `Ctrl-G` cancels.

52.16.3 Enough vim to edit a file and get out

1. vim has modes. That is the whole thing people find confusing. You start in normal mode, where letters are commands, not text.

Key	What it does
`i`	insert before the cursor
`a`	insert after the cursor
`o`	open a new line below
`Escape`	leave insert mode
`h j k l`	left, down, up, right
`0`	start of line
`$`	end of line
`gg`	top of file
`G`	bottom of file
`:42`	go to line 42
`/text` then Enter	search forward
`n`	next match
`x`	delete one character
`dd`	delete the whole line
`yy`	copy the whole line
`p`	paste after the cursor
`u`	undo
`Ctrl-R`	redo
`:w`	write the file
`:q`	quit
`:wq`	write and quit
`:q!`	quit and throw away changes

2. The five you actually need, in order: press `Escape` to be sure you are in normal mode. Press `i` to type. Press `Escape` again. Type `:wq` and press Enter to save and leave. Type `:q!` and press Enter to leave without saving.

3. If a git command drops you into an editor you did not expect, that is vim, because it is the default `core.editor` on many systems. Change it with `git config --global core.editor "nano"` if you would rather.
4. nano is the friendly alternative. `Ctrl-O` writes, `Ctrl-X` exits, and it prints the shortcuts along the bottom of the screen.

52.17 What to do when you are stuck

52.17.1 The debugging method

1. **Reproduce it.** A bug you cannot trigger on purpose cannot be fixed, only guessed at. Write down the exact steps.
2. **Read the actual error.** Not the summary. The first line and the deepest line of the stack.
3. **Establish what is true.** Write down three things you have proved and how you proved them. Write down three things you are assuming.
4. **Check the assumptions first.** Most bugs live there. The version you think you are running, the file you think you are editing, the environment variable you think is set.
5. **Bisect.** Cut the problem in half. Does the failure happen with half the input, half the config, half the code. `git bisect` does this over history automatically.
6. **Change one thing at a time.** Two changes at once means you learn nothing from the result.
7. **Look at the boundary.** Most faults sit where two systems meet: your code and a library, your machine and the network, your process and the filesystem.
8. **Ask what changed.** If it worked yesterday, something moved. A deployment, a certificate, a DNS record, a dependency update, a disk filling up.
9. **Explain it out loud** to a person or an object. Saying it in order forces the gap into view. This is called rubber-duck debugging, and it works because it makes you state your assumptions.
10. **Take a break.** After ninety minutes with no progress, stop. Returning with fresh attention beats another hour of the same.
11. **Write down the fix** and why it worked. The same problem will come back to you or to somebody else.

52.17.2 How to read an error message

1. Read the **first** line and the **last** line before anything in between. The first says what went wrong. The last is often where it went wrong.
2. Find the **first line that mentions your own code**. Everything above it is usually library internals.
3. Separate the **error type** from the **message** from the **location**. `FileNotFoundError` is the type. The path is the message. The line number is the location.
4. Look for a number: an exit code, an errno, a status code, a signal number. Those are searchable and precise.
5. Common exit codes: 0 success, 1 general error, 2 misuse of a shell builtin, 126 found but not executable, 127 command not found, 130 interrupted by Ctrl-C, 137 killed by signal 9, often the out-of-memory killer.
6. Common errno values: 2 ENOENT no such file, 13 EACCES permission denied, 32 EPIPE broken pipe, 98 EADDRINUSE address already in use, 110 ETIMEDOUT connection timed out, 111 ECONNREFUSED connection refused.
7. Note what the error does **not** say. An error that says "connection refused" tells you something answered. An error that says "timed out" tells you nothing answered. Those are different problems.

52.17.3 The minimal reproduction

1. Copy the failing thing into a new, empty directory or file.
2. Delete half of it. Does it still fail. If yes, delete half of what is left. If no, put it back and delete the other half.

3. Repeat until removing anything makes the failure go away.
4. What remains is the reproduction. It is usually five to twenty lines.
5. Remove your credentials, your company's names and your real data. Replace them with `example.com` and the documentation addresses from section 52.4.3.
6. Half the time the bug becomes obvious during this process, and you never need to ask anyone. That is not wasted effort; that is the method working.

52.17.4 Searching effectively

1. Search the **exact error text**, in quotes, with your own paths, names and numbers removed.
2. Add the technology and the version: `postgres 16`, not `postgres`.
3. Add the year if the technology moves fast, or restrict the results to the last year. Answers about a 2019 version of a fast-moving tool are often wrong now.
4. Search the project's own issue tracker before the general web. The people who wrote the code discuss it there.
5. Read the official documentation for the exact version you are running. Not the latest, the one you have.
6. When an answer says "just disable the check", read why the check exists before doing it.
7. Prefer answers that explain the mechanism. An answer that gives you a command with no reason will not help you next time, and may not be safe.

52.17.5 Asking a question that gets answered

1. Give these seven things, in this order, and you will get an answer.
2. **What you are trying to do**, in one sentence, at the level of the goal and not the current attempt.
3. **What you did**, as exact commands or code that someone else could run.
4. **What you expected** to happen.
5. **What actually happened**, pasted verbatim, including the full error, as text and not as a picture.
6. **Your environment**: operating system and version, tool versions, and how it was installed.
7. **What you have already tried** and what each attempt ruled out. This is the single line that most changes the quality of the answers, because it stops people suggesting things you have done.
8. **A minimal reproduction**, from section 52.17.3.
9. Two things not to do. Do not ask "does anyone know about X"; just ask the question. And do not describe the problem you invented while trying to solve the real one, without saying what the real one is; that is the most common reason a thread goes nowhere.
10. If you solve it yourself before anyone replies, post the answer. Somebody with the same problem will find it.

52.98 Common wrong ideas

1. Wrong: a failed ping means the host is down. Right: it means ICMP did not come back. Very many hosts drop ICMP deliberately and serve HTTPS perfectly.
2. Wrong: `* * *` at the end of a traceroute proves the fault is at that hop. Right: it proves those routers did not send ICMP time-exceeded messages, which is common and often deliberate.
3. Wrong: a timeout and a connection refused are both just failures. Right: refused means something answered and said no; a timeout means nothing answered at all. They point at different causes.
4. Wrong: private addresses in a traceroute mean something is misconfigured. Right: ISPs number their own cores privately as normal practice, which is why the reader saw 172.31, 172.26 and 172.16 addresses.
5. Wrong: `git rev-parse origin/main` tells you where the remote is. Right: it tells you where the remote was at your last fetch. Only `git ls-remote` asks the server.
6. Wrong: a disconnect during a push means the push did not happen. Right: the disconnect happened while reading the reply, so it says nothing about whether the write landed. Re-query and find out.

7. Wrong: 1 GB and 1 GiB are the same. Right: they differ by 7.4 per cent, and the gap grows with every step up the scale.
8. Wrong: MB/s and Mb/s are the same. Right: they differ by a factor of eight. A 100 Mbit/s line delivers about 12 MB/s at best.
9. Wrong: a green tick on a branch means your commit passed CI. Right: it means some run passed for something on that branch. Check the run's head SHA against your own.
10. Wrong: hashing a password with SHA-256 is secure storage. Right: SHA-256 is far too fast. Use Argon2id, scrypt or bcrypt with the parameters in section 52.13.3.

52.99 Chapter summary in 20 lines

1. This chapter is for looking things up, not for reading through.
2. Learn the latency ladder as shape, not as digits: cache in nanoseconds, memory in hundreds of nanoseconds, storage in microseconds, disks and networks in milliseconds.
3. Scaled to human time at one nanosecond per second, a cross-continent round trip is several years and a fifteen-second timeout is several centuries.
4. About one millisecond of round trip per hundred kilometres of fibre is a rule you can do in your head, and it is physics.
5. Learn 2^10, 2^20 and 2^30, and the three that cause bugs: 2^31, 2^32 and 2^53.
6. Decimal and binary prefixes differ by a percentage that grows with scale: 2.4 per cent at kilo, 7.4 at giga, 20.9 at yotta.
7. One hex digit is four bits and two are one byte. That is the only reason hexadecimal exists in computing.
8. UTF-8 costs one byte for English, two for most European scripts, three for Devanagari and East Asian scripts, four for emoji.
9. The seven OSI layers are the vocabulary; the four TCP/IP layers are what actually runs. Both statements are true together.
10. The private ranges are 10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16, from RFC 1918, and the middle one reaches to 172.31.
11. For every command, ask two questions: what does a pass prove, and what does it not prove. That second question is the whole method.
12. For "I cannot reach a site", work upward: link, DNS, TCP, TLS, HTTP, content. Each pass eliminates everything below it.
13. Refused, unreachable and silent are three different failures with three different causes, and only the third costs you a full timeout.
14. Report a fault to an ISP in protocol terms with a control test, and never accuse anyone; the evidence supports a silent drop and does not name a party.
15. Almost every git command works offline, because the whole history is in the `.git` directory on your machine.
16. Git failures fall into five families: transport, authentication, scope, server-side rejection and local. Naming the family is most of the fix.
17. After any failed push, ask the server what it now believes with `git ls-remote` before doing anything else.
18. Quote your variables, start scripts with `set -euo pipefail`, and never pipe an unknown URL into a shell.
19. A magic number identifies a file; the extension only suggests one.
20. When stuck: reproduce, read the real error, separate what is proved from what is assumed, change one thing at a time, and write down the fix.

53

CHAPTER

KB

Master Timeline and Glossary

53.0 How to use this chapter

1. This is the last chapter, and it is the only one built for looking things up rather than reading straight through.
2. Every other chapter in this book teaches one topic twice: once in plain words, once in the engineer's words. This chapter holds the index to all of them.
3. Because it is a reference, it does not use the six-block shape that the rest of the book uses. It is a timeline, three smaller timelines, a long glossary, and two closing pages.
4. Section 53.1 is the master timeline of computing, from clay tokens around 3000 BCE to the state of things in 2026. Three columns: the year, what happened, and why it mattered.
5. Section 53.2 is a timeline of programming languages, with who made each one and what problem it was built for.
6. Section 53.3 is a timeline of standards and protocols: the written-down agreements that let machines from different makers work together.
7. Section 53.4 is the glossary. It is the main body of this chapter. It is alphabetical, one letter per heading, and it is meant to cover every term used anywhere in the 53 chapters.
8. Every glossary line has exactly three parts, separated by dashes: the term, then the plain meaning in one short sentence, then the technical meaning with the precise definition.
9. Read the plain meaning if you just want to follow a conversation. Read the technical meaning if you have to write it down, argue about it, or build it.
10. Section 53.5 is honest advice on what to do next. Section 53.6 compresses the whole book into exactly 100 numbered lines.
11. A word about dates. Where a thing has several defensible dates, we pick one and say which event it is: invented, first working, published, standardized or shipped. Those can be years apart.
12. Where experts genuinely disagree about a date, we say "about" and give the range. The Antikythera mechanism is the clearest case: published estimates cluster between 150 and 100 BCE.
13. Anything in this chapter dated 2025 or 2026 is recent enough that you should check it again before quoting it. Everything before about 2020 is settled history.
14. Two small conventions used throughout. BCE means before the common era, so the numbers count down. "c." is short for the Latin circa and means "about".

53.1 The master timeline of computing

The table below has one row per event. It is one continuous list in year order. Ancient dates are approximate by nature; modern ones are exact.

Year	Event	Why it mattered
c. 3000 BCE	Sumerian clay tokens	First stored count outside a head
c. 2700 BCE	Sumerian counting board	Ancestor of every abacus
c. 2000 BCE	Babylonian base-60 tables	Reusable answers, written down
c. 1900 BCE	Babylonian place value	Position of a digit carries value
c. 1550 BCE	Rhind papyrus copied	Oldest surviving arithmetic manual
c. 500 BCE	Chinese counting rods	Positional arithmetic with sticks
c. 300 BCE	Euclid writes the Elements	A procedure you can follow blindly
c. 300 BCE	Salamis counting tablet	Oldest surviving counting board
c. 100 BCE	Antikythera mechanism	Geared model of the sky, 30+ gears
c. 100 CE	Roman hand abacus	Calculation becomes portable
c. 499 CE	Aryabhata's Aryabhatiya	Indian decimal place value written
628 CE	Brahmagupta's rules for zero	Zero becomes a number, not a gap
c. 820 CE	Al-Khwarizmi on calculation	Gives us "algorithm" and "algebra"
c. 1200 CE	Chinese suanpan in wide use	Fast decimal arithmetic for trade
1202	Fibonacci's Liber Abaci	Arabic numerals reach Europe
1614	Napier publishes logarithms	Multiplying becomes adding
1617	Napier's bones described	Portable multiplication aid
1622	Oughtred's slide rule	Analogue calculation for 350 years
1623	Schickard's calculating clock	First known geared calculator
1642	Pascal builds the Pascaline	Geared adder with automatic carry
1673	Leibniz's stepped reckoner	First machine to multiply directly
1703	Leibniz publishes on binary	Base 2 arithmetic written down
1725	Bouchon's punched-tape loom	Holes in paper steer a machine
1728	Falcon's punched-card loom	Cards replace fragile tape
1801	Jacquard shows loom in Paris	Pattern separated from mechanism
1804	Jacquard head patented	The program becomes a thing you own
1822	Babbage's difference engine	Machine-made mathematical tables
1837	Cooke-Wheatstone telegraph	Messages faster than horses
1837	Analytical Engine designed	Mill, store, cards: a real computer
1838	Morse and Vail show their code	Letters become dots and dashes
1843	Lovelace publishes Note G	First published algorithm for a machine
1844	Washington to Baltimore line	Telegraph proven over distance

Year	Event	Why it mattered
1847	Boole's Mathematical Analysis	Logic written as algebra
1854	Boole's Laws of Thought	AND, OR and NOT get their rules
1858	First transatlantic cable works	Continents linked, then it failed
1866	Permanent Atlantic cable laid	The first reliable global network
1876	Bell patents the telephone	Sound itself travels on wires
1877	Edison's phonograph	Sound recorded and played back
1878	First telephone exchange	Switching, not point-to-point wires
1887	Hertz produces radio waves	Signals without any wire
1889	Hollerith patent 395,782	Counting by electricity and holes
1890	US census tabulated by machine	First mass data processing job
1896	Tabulating Machine Company	The firm that becomes IBM
1897	Thomson identifies the electron	The carrier of charge is named
1900	Hilbert lists 23 problems	Sets the century's maths agenda
1901	Marconi's transatlantic signal	Radio crosses an ocean
1904	Fleming's thermionic valve	First electronic one-way device
1906	De Forest's audion triode	Electronic amplification is possible
1911	CTR formed by merger	Punched cards become an industry
1924	CTR renamed IBM	Sixty years of dominance begin
1925	Bell Telephone Laboratories	Research lab behind much of this book
1928	The Entscheidungsproblem posed	Can proof itself be mechanized
1931	Godel's incompleteness theorems	Some true things cannot be proved
1936	Turing's On Computable Numbers	Defines what a computer can ever do
1936	Church's lambda calculus	A second, equivalent model of computing
1937	Shannon's relay thesis at MIT	Boolean algebra meets real wires
1937	Stibitz builds the Model K	Binary adder from relays, on a table
1938	Zuse's Z1 in a Berlin flat	Mechanical binary programmable machine
1939	Hewlett-Packard founded	The garage startup pattern begins
1940	Stibitz demonstrates remotely	First computing over a telegraph line
1941	Zuse's Z3 runs	First working program-driven computer
1942	Atanasoff-Berry Computer works	Electronic binary arithmetic, no program
1943	Colossus Mark 1 at Bletchley	First large-scale electronic computer

Year	Event	Why it mattered
1944	Harvard Mark I dedicated	US sequence-controlled calculator
1945	First Draft on the EDVAC	Stored-program design written down
1945	Bush's As We May Think	Imagines linked documents, the memex
1946	ENIAC shown to the press	Electronic speed becomes public news
1947	Transistor at Bell Labs	The end of the vacuum tube begins
1948	Manchester Baby runs a program	First stored program actually executed
1948	Shannon's theory of communication	Information gets a unit and a limit
1948	Wiener publishes Cybernetics	Feedback and control as one subject
1949	EDSAC enters regular service	Subroutines and a real user service
1950	Turing's imitation game paper	The question of machine thinking framed
1951	Ferranti Mark 1 delivered	First commercially sold computer
1951	UNIVAC I accepted by the census	Computing enters US government work
1952	Hopper's A-0 translator	The idea of a compiler appears
1952	UNIVAC predicts the election	Computers enter everyday speech
1953	IBM 701 shipped	IBM enters electronic computing
1953	Core memory in Whirlwind	Fast, reliable random-access memory
1954	Silicon transistor at TI	Silicon beats germanium on heat
1956	IBM 350 RAMAC disk drive	Random access to five megabytes
1956	Dartmouth workshop names AI	The field gets its name
1957	Fortran compiler delivered	Programming above the machine
1957	Sputnik launched	Panic that funds ARPA and the space race
1957	Eight leave Shockley	Fairchild, and then Silicon Valley
1958	Kilby demonstrates a circuit on one chip	Components without separate wires
1958	ARPA founded	The agency that funds the internet
1959	Noyce's planar integrated circuit	Chips can be made in quantity
1959	COBOL specified by CODASYL	Business data processing standardized
1959	DEC PDP-1 announced	Interactive computing begins
1960	ALGOL 60 report published	Block structure and formal grammar

Year	Event	Why it mattered
1960	Baran begins work at RAND	Networks that survive damage
1961	CTSS time-sharing demonstrated	Many users share one machine
1962	Licklider heads ARPA's IPTO	Funding aimed at networked computing
1962	Spacewar on the PDP-1	The first widely copied video game
1963	ASCII published	One agreed number per letter
1963	Sutherland's Sketchpad	Interactive graphics and constraints
1964	IBM System/360 announced	One family, one instruction set
1964	BASIC written at Dartmouth	Beginners can write programs
1964	Engelbart's first mouse	Pointing becomes an input method
1965	Moore's law article published	The industry gets a roadmap
1965	Davies coins the word packet	Data is chopped into labelled pieces
1965	DEC PDP-8 at $18,500	The minicomputer era opens
1966	Taylor funds the ARPANET	The project is actually paid for
1968	Intel founded	The chip giant begins
1968	Engelbart's demo in San Francisco	Mouse, windows, hypertext, video call
1969	RFC 1 published	The open standards habit begins
1969	ARPANET's first message, 29 Oct	Two computers talk across a country
1969	Unix begun at Bell Labs	The operating system family that won
1969	AMD founded	A second source for chips
1970	Xerox PARC founded	Where the modern desktop was invented
1970	Codd's relational model paper	Data as tables you can query
1970	Intel 1103 DRAM shipped	Semiconductor memory beats magnetic core
1971	Intel 4004 announced	A whole processor on one chip
1971	Tomlinson sends network email	The at sign is chosen as a separator
1971	IBM ships the floppy disk	Removable storage for everyone
1971	Project Gutenberg begins	Text made free and machine-readable
1972	C language at Bell Labs	Portable systems programming
1972	Atari's Pong released	Video games become a business
1972	ARPANET demonstrated publicly	Doubters are convinced
1973	Xerox Alto working	Screen, mouse, windows, network, all at once
1973	Metcalfe's Ethernet memo	Local networks over one shared cable

Year	Event	Why it mattered
1973	Norway and UK join ARPANET	The network becomes international
1974	Cerf and Kahn's protocol paper	Networks of networks: the internet idea
1974	Intel 8080 released	The chip the hobbyists could buy
1975	Altair 8800 on sale	Personal computing starts in kit form
1975	Microsoft founded	Software becomes a separate industry
1975	MOS 6502 sold at $25	Cheap CPU for home computers
1976	Apple founded	The other half of the home computer story
1976	Diffie and Hellman's paper	Secrets agreed over an open line
1976	Cray-1 delivered	Vector supercomputing arrives
1977	Apple II, PET and TRS-80	Computers you buy ready to use
1977	RSA public-key algorithm	Signatures and key exchange with maths
1977	Three-network TCP test	The internet idea proven end to end
1978	Intel 8086 released	The x86 line that still runs today
1978	TCP is split into TCP and IP	The thin waist of the internet appears
1978	Kernighan and Ritchie's C book	One small book teaches a generation
1979	VisiCalc released	The first application worth buying a computer for
1979	Usenet created	Global discussion with no owner
1979	Motorola 68000 released	The CPU behind the first Macs and Amigas
1980	Ethernet Blue Book published	Three firms agree one local network
1980	Sinclair ZX80 sold in the UK	Computers under 100 pounds
1981	IBM PC model 5150 announced	The business standard, and clones
1981	RFC 791 and RFC 793 published	IPv4 and TCP written down properly
1981	Xerox Star launched	The desktop metaphor goes on sale
1982	Sun Microsystems founded	Workstations and networked Unix
1982	Commodore 64 released	Best-selling single computer model ever
1982	SMTP defined in RFC 821	Email gets one agreed delivery protocol
1983	TCP/IP flag day, 1 January	The old protocol switched off overnight
1983	DNS proposed in RFC 882	Names stop living in one shared file

Year	Event	Why it mattered
1983	IEEE 802.3 Ethernet standard	Ethernet becomes vendor-neutral
1983	GNU project announced	Free software as an organized movement
1983	Apple Lisa launched	A graphical interface reaches the market
1984	Macintosh launched, 24 January	Graphical computing becomes affordable
1984	Cisco founded	Routing becomes a product
1984	ISO 7498 publishes the OSI model	Seven layers, the teaching vocabulary
1985	Windows 1.0 released	Microsoft enters graphical computing
1985	IEEE 754 floating point published	Every machine rounds the same way
1985	symbolics.com registered, 15 March	The first .com domain name
1985	ARM1 first works at Acorn	The CPU design now in every phone
1985	NSFNET begins	The research backbone that becomes the internet
1986	First IETF meeting	The internet's standards body forms
1987	Perl 1.0 released	Text processing and system glue
1987	RFC 1034 and 1035 published	DNS as it still works today
1987	Toshiba presents NAND flash	The storage in every phone and SSD
1987	IBM VGA introduced	640x480 becomes the baseline display
1988	The Morris worm, 2 November	First internet-wide security incident
1988	NSFNET upgraded to T-1	A 1.544 Mbit/s national backbone
1988	IRC written in Finland	Real-time text chat, still running
1989	Berners-Lee's web proposal	Linked documents over the internet
1989	RFC 1122 host requirements	What every internet host must do
1989	GNU General Public Licence v1	Copyleft in a legal document
1990	ARPANET decommissioned	The original network retires quietly
1990	First web server runs at CERN	info.cern.ch answers requests
1990	Windows 3.0 released	Windows finally sells in volume
1991	Linux kernel announced, 25 August	A free Unix-like kernel appears
1991	Web software released publicly	Anyone can run a server or a browser
1991	Python 0.9.0 released	Readability as a design goal
1991	Unicode 1.0 published	One number per character, all scripts

Year	Event	Why it mattered
1991	PGP released	Strong encryption in ordinary hands
1991	First GSM call made	Digital mobile telephony begins
1992	UTF-8 designed on a placemat	Unicode that does not break old software
1992	JPEG published as a standard	Photographs become small enough to send
1993	Mosaic browser released	Images inline, and the web grows fast
1993	CERN puts the web in public domain	No fee, no permission, no owner
1993	Intel Pentium released	The superscalar x86 era
1993	Doom released, 10 December	3D games and shareware distribution
1993	MP3 standardized in MPEG-1	Music files small enough to share
1994	Netscape founded	The web becomes an industry
1994	W3C founded	The web gets a standards body
1994	Linux 1.0 released	Free Unix reaches production quality
1994	Amazon founded	Retail moves online
1995	Netscape IPO, 9 August	The dot-com boom psychology begins
1995	NSFNET switched off, 30 April	The internet becomes commercial
1995	Windows 95 released, 24 August	The graphical PC becomes the default
1995	Java, JavaScript, PHP and Ruby	Four lasting languages in one year
1995	India's public internet, 15 August	VSNL opens dial-up to the public
1996	USB 1.0 published	One connector to replace many
1996	RFC 1918 private addresses	10.x, 172.16.x and 192.168.x reserved
1996	HTTP/1.0 written down, RFC 1945	The web's protocol described at last
1997	IEEE 802.11 published	Wireless local networking standardized
1997	HTTP/1.1 published, RFC 2068	Reused connections and virtual hosts
1997	Deep Blue beats Kasparov	Search plus hardware beats a champion
1998	Google founded, 4 September	Search becomes the front door
1998	ICANN incorporated	Names and numbers get a caretaker
1998	XML 1.0 published	Machine-readable structured text
1998	IPv6 specified in RFC 2460	128-bit addresses defined
1998	Bluetooth SIG founded	Short-range wireless gets an owner

Year	Event	Why it mattered
1999	Napster launched	Peer-to-peer file sharing at scale
1999	802.11b and the Wi-Fi name	Wireless becomes a consumer product
1999	TLS 1.0 published, RFC 2246	SSL becomes an open standard
1999	Nvidia GeForce 256	The word GPU enters common use
2000	NASDAQ peaks, 10 March	5,048.62, then a long fall
2000	USB 2.0 published	480 Mbit/s makes USB storage practical
2000	GPS accuracy degradation ended	Civil GPS becomes ten times better
2000	DDR SDRAM standardized	Two transfers per clock in main memory
2001	Wikipedia launched, 15 January	Reference work written by everyone
2001	Mac OS X 10.0 released	A Unix core under a consumer desktop
2001	Windows XP released	The PC desktop for a decade
2001	iPod released	Portable digital music goes mainstream
2001	Agile Manifesto published	A named alternative to big up-front plans
2001	IBM POWER4 has two cores	Multicore processors begin
2002	The Bezos API mandate	Teams must talk only through interfaces
2003	AMD Opteron ships x86-64	64-bit on ordinary hardware
2003	PCI Express 1.0 published	Serial lanes replace parallel buses
2003	Skype released	Voice calls over the public internet
2003	Steam launched	Games delivered without discs
2004	Facebook launched, 4 February	Social networking at national scale
2004	Firefox 1.0 released	Browser competition returns
2004	Gmail beta, 1 April	Applications move into the browser
2004	Ubuntu 4.10 released	Linux packaged for ordinary users
2005	BitKeeper licence withdrawn	The trigger for writing git
2005	Git's first commit, 7 April	Distributed version control done right
2005	YouTube founded	Video becomes a normal web object
2005	Dual-core desktop chips ship	The clock speed race ends
2006	Amazon S3 launched, 14 March	Storage rented by the gigabyte-month
2006	Amazon EC2 beta, 25 August	Servers rented by the hour
2006	Twitter launched	Short public messages at scale
2006	Hadoop released	Batch processing across cheap machines

Year	Event	Why it mattered
2007	iPhone announced, 9 January	The pocket computer with a real browser
2007	CUDA released	GPUs become general-purpose compute
2007	Android announced, 5 November	An open phone platform for everyone else
2007	Netflix begins streaming	Video moves from post to packets
2008	Bitcoin white paper, 31 October	Money without a central ledger keeper
2008	Chrome released, 2 September	Fast JavaScript and per-tab isolation
2008	GitHub launched	Code hosting becomes social
2008	First Android phone ships	The two-platform phone world begins
2008	App Store opens, 10 July	Software distribution through one gate
2008	USB 3.0 published	5 Gbit/s over a compatible connector
2009	Bitcoin genesis block, 3 January	The chain actually starts
2009	Node.js released	JavaScript on the server
2009	Go announced, 10 November	Simple concurrency for server software
2009	IEEE 802.11n published	MIMO takes Wi-Fi past 100 Mbit/s
2010	iPad released	Tablets become a product category
2010	Bluetooth 4.0 adds Low Energy	Coin-cell devices join the network
2010	Stuxnet discovered	Software as a weapon against machinery
2010	OpenStack launched	Open-source infrastructure clouds
2011	IANA runs out of IPv4, 3 February	The address shortage becomes real
2011	IBM Watson wins Jeopardy	Statistical language systems go public
2011	Siri ships on the iPhone 4S	Voice assistants reach the mainstream
2011	Smartphones outsell PCs	The centre of computing moves
2012	AlexNet wins ImageNet	Deep learning on GPUs beats everything
2012	World IPv6 Launch, 6 June	IPv6 turned on permanently by big sites
2012	Raspberry Pi released	A 35-dollar computer for learning
2012	DDR4 standard published	Higher density at lower voltage
2013	Docker released, March	Containers become easy to use
2013	Snowden disclosures begin, June	Encryption everywhere becomes the norm

Year	Event	Why it mattered
2013	IEEE 802.11ac published	Gigabit Wi-Fi in the 5 GHz band
2014	Kubernetes announced, June	Cluster orchestration gets a standard
2014	Heartbleed disclosed, 7 April	One buffer bug affects half the web
2014	Swift announced	Apple replaces Objective-C
2014	USB Type-C specified	A reversible connector for everything
2015	HTTP/2 published, RFC 7540	Binary framing and multiplexed streams
2015	Kubernetes 1.0 and the CNCF	Container orchestration becomes neutral
2015	Let's Encrypt begins issuing	Free automated certificates for all
2015	Rust 1.0 released, 15 May	Memory safety without a garbage collector
2015	TensorFlow released	Machine learning frameworks go public
2016	AlphaGo beats Lee Sedol, March	Search plus learning solves Go
2016	Mirai botnet DDoS, 21 October	Insecure home devices break big services
2016	Bluetooth 5.0 published	Four times the range for Low Energy
2016	PyTorch released	Define-by-run makes research faster
2017	Attention Is All You Need, June	The transformer architecture published
2017	WannaCry ransomware, 12 May	Unpatched machines at national scale
2017	PCI Express 4.0 published	16 GT/s per lane
2018	Meltdown and Spectre, 3 January	Speculation leaks data across boundaries
2018	GDPR takes effect, 25 May	Data protection with real penalties
2018	TLS 1.3 published, RFC 8446	Fewer round trips, forward secrecy only
2018	GPT-1 and then BERT	Pretraining plus fine-tuning wins in NLP
2018	Microsoft agrees to buy GitHub	Open source and big vendors converge
2019	GPT-2 released in stages	Staged release becomes a debate
2019	Wi-Fi 6 certification opens	OFDMA brings order to crowded air
2019	PCI Express 5.0 published	32 GT/s per lane
2019	USB4 specification published	Thunderbolt's tunnelling becomes open
2020	GPT-3 shows in-context learning	Scale alone changes what a model can do
2020	Apple M1 ships, November	ARM reaches the desktop with real speed

Year	Event	Why it mattered
2020	DDR5 standard published, July	4800 MT/s and on-module regulation
2020	Remote work at global scale	Video and cloud tools become essential
2021	GitHub Copilot preview, June	Code completion from a language model
2021	Log4Shell disclosed, 9 December	One logging library, global emergency
2021	IEEE 802.11ax approved	Wi-Fi 6 becomes a full standard
2022	PCI Express 6.0 published	PAM4 signalling and forward error correction
2022	HTTP/3 published, RFC 9114	HTTP over QUIC, on top of UDP
2022	The Chinchilla paper, March	Data, not just parameters, sets quality
2022	Stable Diffusion released, August	Image generation with open weights
2022	ChatGPT released, 30 November	Language models reach the general public
2023	GPT-4 released, 14 March	Multimodal input and much stronger results
2023	Llama and Llama 2 released	Open-weight models become serious
2023	Claude released, 14 March	A second major assistant family
2023	EU AI Act agreed, December	The first broad legal framework for AI
2024	Wi-Fi 7 certification, 8 January	320 MHz channels and multi-link operation
2024	Nvidia Blackwell announced	208 billion transistors in two dies
2024	Bluetooth 6.0 published	Channel sounding measures true distance
2024	El Capitan tops the TOP500	About 1.74 exaflops at Livermore
2024	Unicode 16.0 published	154,998 characters defined
2024	EU AI Act in force, 1 August	Obligations start phasing in
2025	PCI Express 7.0 specification	128 GT/s per lane, no products yet
2025	Reasoning models become normal	Models spend compute at answer time
2025	2 nm class production starts	Gate-all-around transistors in volume
2026	Agentic tools in daily use	Models call tools and run multi-step work
2026	Long contexts are routine	Whole codebases fit in one request
2026	On-device inference is common	Small models run on phones and laptops

53.2 Timeline of programming languages

The year given is the year the language first worked or was first released, not the year it was designed or the year it became popular. Those can differ by a decade.

Year	Language	Creator	What it was for
1949	Short Code	Mauchly and Schmitt	Readable arithmetic on UNIVAC
1952	Autocode	Alick Glennie	First compiler, Manchester Mark 1
1952	A-0	Grace Hopper	Assembling reusable subroutines
1956	IPL	Newell, Shaw, Simon	Lists and symbols for early AI
1957	FORTRAN	John Backus, IBM	Scientific and numerical work
1958	ALGOL 58	International committee	Publishing algorithms clearly
1958	LISP	John McCarthy	Symbolic computing and AI
1959	COBOL	CODASYL committee	Business records and reports
1960	ALGOL 60	Naur, Backus and others	Block structure, formal grammar
1962	APL	Kenneth Iverson	Whole-array mathematics
1962	Simula I	Dahl and Nygaard	Simulating real-world systems
1964	BASIC	Kemeny and Kurtz	Teaching students to program
1964	PL/I	IBM	One language for all workloads
1967	Simula 67	Dahl and Nygaard	Classes and objects, the first time
1967	Logo	Papert and colleagues	Teaching children with a turtle
1970	Pascal	Niklaus Wirth	Teaching structured programming
1970	Forth	Charles Moore	Small stack-based control systems
1972	C	Dennis Ritchie	Writing Unix portably
1972	Prolog	Colmerauer and Roussel	Rules and logical deduction
1972	Smalltalk	Alan Kay's team, PARC	Objects, messages, live interfaces
1973	ML	Robin Milner	Proofs, with inferred types
1975	Scheme	Sussman and Steele	A tiny, clean Lisp
1977	awk	Aho, Weinberger, Kernighan	One-line text processing
1979	Bourne shell	Stephen Bourne	Gluing Unix commands together
1979	SQL, first product	Chamberlin and Boyce	Asking questions of tables

Year	Language	Creator	What it was for
1980	Ada	Ichbiah for the US DoD	Long-lived safety-critical systems
1983	C++	Bjarne Stroustrup	Objects with no speed penalty
1984	Verilog	Moorby and Gateway	Describing hardware, not software
1984	MATLAB, commercial	Cleve Moler	Matrix and signal mathematics
1984	Objective-C	Cox and Love	Smalltalk-style objects over C
1985	PostScript	Warnock and Geschke	Telling a printer what a page is
1986	Erlang	Armstrong at Ericsson	Phone switches that never stop
1987	Perl	Larry Wall	Reports, text and system glue
1987	VHDL	US DoD project	Hardware description, IEEE 1076
1989	Bash	Brian Fox for GNU	A free, extended Bourne shell
1990	Haskell	Academic committee	Pure, lazy functional research
1991	Python	Guido van Rossum	Readable general-purpose code
1991	Visual Basic	Microsoft	Building Windows apps quickly
1993	Lua	Ierusalimschy and team	Small language embedded in apps
1993	R	Ihaka and Gentleman	Statistics and data graphics
1995	Java	James Gosling, Sun	One binary on many machines
1995	JavaScript	Brendan Eich, Netscape	Small scripts inside web pages
1995	PHP	Rasmus Lerdorf	Pages built on the server
1995	Ruby	Yukihiro Matsumoto	Programming that feels pleasant
1996	OCaml	INRIA team	ML with objects and speed
2000	C#	Anders Hejlsberg, Microsoft	Applications on .NET
2003	Scala	Martin Odersky	Objects and functions on the JVM
2005	F#	Don Syme, Microsoft	Functional programming on .NET
2007	Clojure	Rich Hickey	A Lisp with immutable data
2009	Go	Griesemer, Pike, Thompson	Simple, fast server programs
2011	Dart	Google	Web apps, later Flutter
2012	Elixir	Jose Valim	Erlang's reliability, nicer syntax

Year	Language	Creator	What it was for
2012	TypeScript	Anders Hejlsberg, Microsoft	Types added to JavaScript
2012	Julia	Bezanson and colleagues	Fast numerical computing
2014	Swift	Chris Lattner, Apple	Safer apps for Apple platforms
2014	Solidity	Gavin Wood and others	Contracts running on Ethereum
2015	Rust 1.0	Graydon Hoare, Mozilla	Safety without a garbage collector
2016	Kotlin 1.0	JetBrains	Concise, safer code for Android
2016	Zig	Andrew Kelley	C's job with fewer surprises
2023	Mojo	Chris Lattner, Modular	Python-like code for AI hardware

53.3 Timeline of standards and protocols

A standard is a document. It says what a thing must do so that two products made by strangers will work together. The number after the name is the document you can look up.

Year	Standard	What it defined
1963	ASCII, ASA X3.4	7-bit codes for 128 characters
1967	ASCII revision	Lower case and control codes settled
1969	RFC 1	The RFC document series itself
1974	Cerf-Kahn TCP paper	Internetworking between networks
1980	RFC 768, UDP	Send a datagram, no promises
1980	Ethernet Blue Book, DIX	10 Mbit/s over one shared cable
1981	RFC 791, IPv4	32-bit addresses and the IP header
1981	RFC 792, ICMP	Error and diagnostic messages
1981	RFC 793, TCP	Ordered, reliable byte streams
1982	RFC 821, SMTP	How mail moves between servers
1982	RFC 826, ARP	IP address to MAC address on a LAN
1983	IEEE 802.3	Ethernet as a vendor-neutral standard
1983	RFC 882 and 883	The first DNS design
1984	ISO 7498, OSI model	Seven named layers for teaching
1985	IEEE 754	Binary floating point, all machines alike
1985	RFC 959, FTP	File transfer with a control channel
1986	ISO 9660	The CD-ROM filesystem
1987	RFC 1034 and 1035	DNS as still used today

Year	Standard	What it defined
1988	IEEE 802.1D	Spanning tree, loop-free bridging
1989	RFC 1122 and 1123	What every internet host must do
1990	IEEE 1076, VHDL	Hardware description as a standard
1991	Unicode 1.0	One code point per character
1992	ITU T.81, JPEG	Lossy image compression
1993	RFC 1459, IRC	Real-time chat rooms
1993	ISO/IEC 11172-3, MP3	Perceptual audio compression
1993	RFC 1519, CIDR	Prefix lengths replace address classes
1993	UTF-8 in the X/Open standard	Unicode as bytes, ASCII-compatible
1994	RFC 1738, URLs	The shape of a web address
1995	SSL 3.0	Encrypted web sessions, now obsolete
1996	RFC 1918	Private address ranges for local use
1996	RFC 1945, HTTP/1.0	Request line, headers, body
1996	USB 1.0	12 Mbit/s, one connector, hot plug
1997	RFC 2068, HTTP/1.1	Keep-alive, host header, chunking
1997	IEEE 802.11	Wireless LAN at 1 and 2 Mbit/s
1997	RFC 2131, DHCP	Automatic address assignment
1998	RFC 2460, IPv6	128-bit addresses, simpler header
1998	RFC 2373, IPv6 addressing	Address types and text notation
1999	RFC 2246, TLS 1.0	SSL becomes an IETF standard
1999	RFC 2616, HTTP/1.1	The version everyone implemented
1999	IEEE 802.11b	11 Mbit/s at 2.4 GHz
1999	IEEE 802.11a	54 Mbit/s at 5 GHz, OFDM
1999	Bluetooth 1.0	Short-range personal wireless links
1999	IEEE 802.1Q	VLAN tags inside Ethernet frames
2000	USB 2.0	480 Mbit/s high speed
2000	JEDEC JESD79, DDR	Data on both clock edges
2001	RFC 3168, ECN	Congestion signalled, not dropped
2003	IEEE 802.11g	54 Mbit/s at 2.4 GHz
2003	PCI Express 1.0	2.5 GT/s serial lanes
2003	JEDEC DDR2	4n prefetch, lower voltage
2003	WPA, then WPA2 in 2004	Wi-Fi encryption that actually works
2004	Bluetooth 2.0 + EDR	About 3 Mbit/s over the same radio

Year	Standard	What it defined
2005	RFC 4271, BGP-4	How networks tell each other routes
2006	RFC 4291, IPv6 addressing	The current IPv6 address rules
2006	RFC 4443, ICMPv6	Errors and neighbour discovery for v6
2006	RFC 4346, TLS 1.1	Fixes for CBC attacks
2007	PCI Express 2.0	5 GT/s per lane
2007	JEDEC DDR3	8n prefetch, 1.5 V
2008	RFC 5246, TLS 1.2	SHA-256 and AEAD cipher suites
2008	RFC 5321, SMTP	The current mail transfer standard
2008	IEEE 754-2008	Decimal formats and fused multiply-add
2008	USB 3.0	5 Gbit/s SuperSpeed, extra wires
2009	IEEE 802.11n	MIMO, 40 MHz, frame aggregation
2009	Bluetooth 3.0 + HS	Wi-Fi used for the bulk transfer
2010	Bluetooth 4.0	Low Energy, GATT, coin-cell devices
2010	PCI Express 3.0	8 GT/s, 128b/130b encoding
2011	RFC 6265, cookies	How browsers store and send cookies
2011	RFC 6455, WebSocket	A two-way channel over one HTTP port
2012	RFC 6598, shared space	100.64.0.0/10 for carrier NAT
2012	RFC 6749, OAuth 2.0	Delegated access without passwords
2012	JEDEC DDR4	1.2 V, bank groups, higher density
2013	IEEE 802.11ac	80 and 160 MHz, 256-QAM, MU-MIMO
2014	RFC 7230 to 7235	HTTP/1.1 rewritten as six documents
2014	USB Type-C 1.0	A reversible 24-pin connector
2014	Bluetooth 4.2	Privacy, longer packets, IPv6 support
2015	RFC 7540, HTTP/2	Binary frames, streams, header compression
2015	RFC 7519, JWT	A signed token you can read
2016	Bluetooth 5.0	Four times the range, twice the speed
2017	IEEE 754-2019 work begins	Refinement of the 2008 revision
2017	PCI Express 4.0	16 GT/s per lane
2017	USB 3.2	20 Gbit/s using two lanes
2018	RFC 8446, TLS 1.3	One round trip, forward secrecy only
2018	RFC 8200, IPv6	IPv6 promoted to full internet standard
2019	PCI Express 5.0	32 GT/s per lane

Year	Standard	What it defined
2019	USB4	Tunnelling, 40 Gbit/s, Type-C only
2019	WPA3 required for new Wi-Fi	Individual encryption, better handshake
2020	JEDEC DDR5	4800 MT/s, on-module power regulation
2021	RFC 9000, QUIC	Reliable streams on top of UDP
2021	IEEE 802.11ax, Wi-Fi 6	OFDMA, 1024-QAM, target wake time
2022	RFC 9110 to 9114	HTTP semantics, and HTTP/3 over QUIC
2022	PCI Express 6.0	64 GT/s, PAM4, forward error correction
2022	USB4 Version 2.0	80 Gbit/s, and 120 Gbit/s one way
2024	Wi-Fi 7 from IEEE 802.11be	320 MHz, 4096-QAM, multi-link operation
2024	Bluetooth 6.0	Channel sounding for true distance
2024	Unicode 16.0	154,998 characters across 168 scripts
2025	PCI Express 7.0	128 GT/s per lane, on paper first

53.4 The glossary

Every line has the same three parts: the term, the plain meaning, then the technical meaning. Where a word means two different things in two fields, both senses are given as separate lines.

53.4.1 A

1. **A record** — the DNS entry holding an IPv4 address — a resource record of type A mapping a name to a 32-bit address; AAAA does the same for IPv6.
2. **ABI** — the binary rulebook two compiled parts must share — application binary interface: calling convention, register use, struct layout, symbol naming.
3. **Abstraction** — hiding detail behind a simpler idea — a layer offering a stable interface while the implementation beneath is free to change.
4. **Activation** — a number flowing through a network as it runs — a layer's output for one input, alive only during the pass and not part of the model.
5. **Activation function** — the bend that stops a network being one straight line — an elementwise nonlinearity such as ReLU, GELU or SwiGLU.
6. **ADC** — turns a real signal into numbers — analogue-to-digital converter, sampling a voltage at a fixed rate and quantizing to a fixed bit depth.
7. **Address space** — every address a program is allowed to name — the range of addresses a process can generate, commonly 2^48 bytes of user space on 64-bit systems.
8. **AES** — the normal way to scramble bulk data — Advanced Encryption Standard, 128-bit block cipher with 128, 192 or 256-bit keys, FIPS 197, 2001.
9. **Agent** — a model that acts instead of only answering — a loop where a model picks a tool call, the tool runs, the result returns, until a stop condition.
10. **Aliasing** — false patterns from sampling too slowly — signal components above half the sample rate folding back as lower, wrong frequencies.

11. **Alignment** (memory) — putting data at tidy addresses — placing an object at an address that is a multiple of its size, required or faster on most architectures.
12. **Alignment** (AI) — making a model behave as intended — the work of getting model behaviour to match stated human intent, values and constraints.
13. **Allocator** — the code that hands out memory — the runtime or kernel component that finds, records and returns memory blocks to callers.
14. **Alpha channel** — how see-through a pixel is — a per-pixel opacity value, usually 8 bits, used when compositing layers.
15. **ALU** — the part that does the sums — arithmetic logic unit, a combinational block performing add, subtract, bitwise and shift operations and setting flags.
16. **Amdahl's law** — the slow part limits your speedup — speedup is bounded by $1/(s + p/N)$ for serial fraction s over N processors.
17. **Ampere** — how much electricity flows per second — the SI unit of current, one coulomb of charge per second.
18. **Amplitude** — how big a wave is — the peak deviation of a signal from its resting value; in audio it maps to loudness.
19. **Analogue** — smoothly varying, not stepped — a signal that may take any value in a continuous range, unlike a digital signal with discrete levels.
20. **Anti-aliasing** — smoothing jagged edges — MSAA, FXAA, TAA and similar methods that add or reuse samples to reduce sampling artefacts.
21. **Anti-cheat** — software that catches players who cheat — combined client integrity checks, server-authoritative rules and statistical detection.
22. **Anycast** — one address, many machines — announcing the same prefix from several sites so each client reaches the topologically nearest one.
23. **Apex domain** — the bare name with no www — the zone's own name, such as example.com, which cannot hold a CNAME under standard DNS rules.
24. **API** — an agreed way for programs to ask each other for things — the endpoints, functions, arguments and guarantees one piece of software offers another.
25. **Argon2** — a password hash built to be slow and greedy for memory — winner of the 2015 Password Hashing Competition, with time, memory and parallelism costs.
26. **ARM** — the processor family in nearly every phone — a RISC architecture licensed as designs by Arm Ltd, first silicon working in 1985.
27. **ARP** — finding a neighbour's hardware address — Address Resolution Protocol, RFC 826, broadcasting for the MAC address owning an IPv4 address on the link.
28. **ARPANET** — the network the internet grew out of — the ARPA-funded packet network, first message 29 October 1969, retired 1990.
29. **Artifact** — a file a build makes and keeps — a stored pipeline output such as a binary, image or test report, retained for later jobs or download.
30. **ASCII** — numbers for English letters — a 7-bit character set from 1963 with 128 code points, 95 printable and 33 control codes.
31. **ASLR** — moving things so attackers cannot guess addresses — address space layout randomization of stack, heap and library bases at load time.
32. **Assembler** — turns readable instructions into machine code — a translator producing one machine instruction per mnemonic and resolving labels and directives.
33. **Assembly language** — machine code you can read — a notation with one mnemonic per instruction, plus labels, macros and assembler directives.
34. **AST** — a program drawn as a tree — abstract syntax tree, the parser's output, with nodes for expressions and statements and no punctuation.
35. **Asynchronous** — start now, collect the answer later — an operation returning before completion, signalling later by callback, event, future or poll.
36. **Atomic** — all at once or not at all — an operation no observer can see half-finished, enforced by hard-

ware instructions, locks or transactions.

37. **Attention** — letting each word look at the others — a weighted sum of value vectors, weighted by the scaled dot product of one query with every key.
38. **Attenuation** — a signal weakening with distance — loss of power along a path, measured in decibels, from resistance, absorption or spreading.
39. **Authoritative server** — the machine holding the real answer for a name — a DNS server answering from its own zone data rather than from cache.
40. **Autonegotiation** — two ports agreeing a speed by themselves — Ethernet link partners exchanging capability pulses and choosing the best shared speed and duplex.
41. **Autonomous system** — one network under one routing policy — an AS with a 16 or 32-bit number that announces its prefixes to neighbours using BGP.
42. **Availability** — how much of the time it works — uptime as a fraction; 99.9 percent permits about 43 minutes of downtime per month.
43. **Availability zone** — one isolated site inside a cloud region — a data centre with independent power and cooling, linked to its region at low latency.
44. **AVX** — wide maths instructions on x86 chips — Advanced Vector Extensions, SIMD sets using 256-bit registers, or 512-bit in AVX-512.

53.4.2 B

1. **Backpressure** — telling the sender to slow down — a flow-control signal propagated upstream when a consumer cannot keep up.
2. **Backpropagation** — working out which weight caused which mistake — reverse-mode automatic differentiation applying the chain rule from the loss back to every parameter.
3. **Bandwidth** — how much can flow per second — the data rate of a link in bits per second, distinct from latency, which is delay.
4. **Bandwidth throttling** — deliberately slowing traffic — rate limiting applied by policy at a network device, service or account level.
5. **Bandwidth-delay product** — how much data fits inside the wire — bandwidth multiplied by round-trip time, the amount that must be in flight to keep a link full.
6. **Bank** (memory) — one independently addressed section of a chip — a DRAM subdivision with its own row buffer, allowing one row open per bank.
7. **Bare repository** — a git repository with no working files — a repository containing only the object database and refs, normally used as a push target.
8. **Base model** — the raw model before it is taught manners — a network trained only on next-token prediction, not yet instruction-tuned or preference-tuned.
9. **Base station** — the mast your phone talks to — the mobile network's radio equipment serving one or more cells, called an eNodeB in LTE and gNodeB in 5G.
10. **Batch** — a handful of examples processed together — the group of samples whose gradients are averaged before one parameter update.
11. **bcrypt** — an old but sound password hash — a 1999 adaptive hash based on Blowfish, with a cost factor that raises work as hardware improves.
12. **Beamforming** — aiming a radio signal instead of shouting everywhere — controlling phase across several antennas so signals add in one direction and cancel elsewhere.
13. **Benchmark** — a test everyone runs so results can be compared — a fixed task and scoring rule; it measures the benchmark first and the world second.
14. **BGP** — how networks tell each other which routes they have — Border Gateway Protocol, RFC 4271, a path-vector protocol carrying prefixes and AS paths between autonomous systems.
15. **Bias** (AI) — the number added after the multiply — a per-unit learned constant added to the weighted sum before the activation function.
16. **Big-endian** — the biggest byte first — a byte order storing the most significant byte at the lowest address; network byte order is big-endian.

17. **Binary** — counting with only 0 and 1 — base-2 positional notation, where each place is worth twice the one to its right.
18. **Binary search** — halving the haystack each time — locating a value in sorted data in O(log n) comparisons by discarding half the range each step.
19. **Binning** — sorting finished chips by how good they turned out — testing dies and selling them at the highest speed and core count they pass.
20. **BIOS** — the old start-up firmware on PCs — the 16-bit firmware that initializes hardware and loads a 512-byte boot sector, superseded by UEFI.
21. **Bisect** — a binary search through history for the bad commit — git bisect, halving the commit range each test to find the first bad commit in log2(n) steps.
22. **Bit** — one on-or-off — a binary digit, the smallest unit of information, value 0 or 1.
23. **Bit depth** — how many steps a sample can take — the number of bits per sample; 16-bit audio gives 65,536 levels and about 96 dB of dynamic range.
24. **Bit rot** — data quietly going bad — undetected corruption of stored bits from charge leakage, media decay or cosmic rays, caught only by checksums.
25. **BLE** — the low-power half of Bluetooth — Bluetooth Low Energy, introduced in version 4.0 in 2010, using 40 channels and short connection events.
26. **BLE GATT** — the rules for reading a small value over Bluetooth — Generic Attribute Profile, a table of services and characteristics addressed by handle and UUID.
27. **Blob** — git's word for a file's contents — a git object holding raw bytes, named by the SHA-1 or SHA-256 of a header plus the content.
28. **Block** (storage) — the smallest lump a drive reads or writes — a fixed-size unit, 512 bytes or 4096 bytes on disks, and typically 128 KiB to 4 MiB in NAND erase blocks.
29. **Blocking** — waiting instead of doing something else — a call that does not return until its work completes, holding the calling thread.
30. **Boolean algebra** — the maths of true and false — an algebra over {0,1} with AND, OR and NOT, shown by Shannon in 1937 to describe switching circuits.
31. **Boot sector** — the very first block a machine reads from a disk — the first 512-byte sector, holding MBR partition entries and a small loader on legacy systems.
32. **Bootloader** — the small program that starts the real one — code that loads and hands control to a kernel, such as GRUB, systemd-boot or U-Boot.
33. **Bootstrapping** — a compiler built with itself — writing a compiler for a language in that same language, after first compiling it with another one.
34. **Boule** — the big silicon crystal a wafer is sliced from — a single-crystal ingot, typically 300 mm across, pulled from a melt by the Czochralski process.
35. **BPE** — cutting text into common chunks — byte-pair encoding, repeatedly merging the most frequent adjacent pair to build a fixed-size subword vocabulary.
36. **Branch** (CPU) — an instruction that may jump — a control-flow instruction whose target or direction may depend on a condition.
37. **Branch** (git) — a moving name for a line of work — a ref under refs/heads that points at one commit and moves forward as you commit.
38. **Branch prediction** — guessing which way a jump goes before knowing — hardware predicting direction and target so the pipeline keeps fetching; a wrong guess costs the pipeline depth.
39. **Bridge** — a device joining two network segments — a layer 2 device forwarding frames by learned MAC address; a switch is a multi-port bridge.
40. **BSSID** — the identity of one specific access point — the 48-bit MAC address of an access point's radio, unlike an SSID, which is a network name.
41. **Buffer** — a waiting area for data — memory holding data between a producer and a consumer running at different speeds.
42. **Buffer overflow** — writing past the end of a box — writing beyond an allocation's bounds, corrupting adjacent memory and often control data such as return addresses.

43. **Bufferbloat** — too much queueing makes everything feel slow — excessive buffering in network devices that inflates latency under load without preventing loss.
44. **Bus** — a shared set of wires — a set of conductors and rules carrying address, data and control signals between components.
45. **Byte** — eight bits, the usual unit of size — an octet, 8 bits, able to hold 256 distinct values.
46. **Bytecode** — halfway between source and machine code — a compact instruction set for a virtual machine, such as JVM bytecode or Python's .pyc format.

53.4.3 C

1. **Cache** — a small fast copy of things you keep needing — memory holding recently or likely used data closer to the consumer than the original.
2. **Cache coherence** — keeping every core's copy honest — a protocol such as MESI ensuring all cores see one consistent value for each cache line.
3. **Cache hit** — the copy was already there — a request satisfied from the cache without going to the slower backing store.
4. **Cache line** — the block a cache moves as one unit — the unit of transfer between cache levels, 64 bytes on most current x86 and ARM designs.
5. **Cache miss** — the copy was not there — a request that must be served from the next level, costing that level's latency.
6. **Callback** — code you hand over to be run later — a function passed to another routine to be invoked when an event or completion occurs.
7. **Canary release** — trying a change on a few users first — routing a small share of traffic to a new version and watching error rates before wider rollout.
8. **Capability** — permission attached to a token, not a person — a transferable right to do a specific operation, held as an unforgeable reference.
9. **Capacitor** — a component that stores a little charge — two conductors separated by a dielectric, storing charge Q equal to capacitance C times voltage V.
10. **CDN** — copies of a site kept near users — content delivery network: caches in many locations, steered by DNS or anycast to reduce distance and origin load.
11. **Certificate** — a signed statement that a key belongs to a name — an X.509 document binding a public key to identifiers, signed by an issuer and valid between two dates.
12. **Certificate authority** — an organization browsers already trust — a CA whose root certificate ships in trust stores and which signs certificates after validating control of a name.
13. **Certificate pinning** — accepting only one specific certificate — a client refusing certificates other than pinned keys, ignoring the wider CA set.
14. **CGNAT** — your provider sharing one public address between many homes — carrier-grade NAT, usually using 100.64.0.0/10 from RFC 6598.
15. **cgroup** — a fence around how much a process may use — a Linux control group limiting and accounting CPU, memory, I/O and process counts for a set of tasks.
16. **Chain of trust** — a signed line from a root to your site — the verification path from a trusted root through intermediates to the leaf certificate.
17. **Check run** — one automated test result on a commit — a status object on a forge, attached to a specific commit SHA, with a conclusion such as success or failure.
18. **Checkout** — putting a commit's files into your folder — updating the working tree and HEAD to match a specified commit, branch or path.
19. **Checksum** — a small number that proves nothing got mangled — a value computed over data and compared after transfer or storage to detect accidental change.
20. **Cherry-pick** — copying one commit onto another branch — applying a single commit's diff elsewhere, creating a new commit with a new hash.
21. **Chinchilla** — the finding that models were trained on too little data — the 2022 DeepMind result giving compute-optimal token and parameter ratios.

22. **Chiplet** — a big chip built from several smaller ones — separate dies packaged together and joined by a high-speed interconnect, improving yield and mixing process nodes.
23. **Chunk** — a piece of a bigger thing sent on its own — a bounded portion of data transferred or processed separately, as in chunked transfer encoding.
24. **CIDR** — writing an address range as a prefix — Classless Inter-Domain Routing, RFC 1519, notation such as 192.168.0.0/24 giving address and prefix length.
25. **Cipher suite** — the exact set of algorithms a connection uses — the negotiated combination of key exchange, authentication, bulk cipher and MAC or AEAD mode.
26. **Circuit switching** — a dedicated path reserved for one call — reserving capacity end to end for the call's duration, unlike packet switching, which shares links.
27. **Client prediction** — the game moving you before the server replies — applying local input immediately and reconciling later against authoritative server state.
28. **Clock** — the machine's heartbeat — a periodic square wave defining the instants at which state elements sample their inputs.
29. **Clock domain** — a region sharing one clock — part of a chip driven by one clock, with synchronizers needed for signals crossing to another domain.
30. **Closure** — a function that remembers where it came from — a function value together with captured bindings from its defining scope.
31. **Cloud region** — a geographic cluster of data centres — a named location such as ap-south-1, containing several availability zones with independent failure domains.
32. **CMOS** — the way nearly all logic is built — complementary metal-oxide-semiconductor, pairing n-channel and p-channel FETs so steady states draw almost no current.
33. **CNAME** — a name that points to another name — a DNS record aliasing one name to another, after which resolution restarts at the target.
34. **Coalescing** — joining small pieces into one bigger one — combining adjacent free blocks, memory accesses or writes to reduce overhead.
35. **Codec** — the thing that squeezes and unsqueezes media — coder-decoder: the algorithm pair compressing and decompressing audio or video, such as AAC, H.264 or AV1.
36. **Codepoint** — the number that identifies a character — a Unicode scalar value from U+0000 to U+10FFFF, excluding the surrogate range.
37. **Cold start** — the first call being slow — the delay while a serverless platform allocates and initializes an execution environment before running your code.
38. **Cold storage** — cheap storage you rarely read — archival tiers with low cost per gigabyte and retrieval delays measured in minutes to hours.
39. **Colour space** — the agreed meaning of a colour number — a defined set of primaries, white point and transfer function, such as sRGB, Display P3 or Rec. 2020.
40. **Commit** — a saved snapshot with a message — a git object naming a tree, zero or more parents, author, committer, timestamps and a message, identified by its hash.
41. **Compaction** — tidying storage by rewriting it — merging and rewriting data to reclaim space, as in log-structured stores and SSD garbage collection.
42. **Compiler** — turns source code into machine code — a program translating a whole source program into another language, usually machine code, before it runs.
43. **Compression** — making data smaller — encoding that reduces size, lossless if the original returns exactly, lossy if some information is discarded.
44. **Concurrency** — several things in progress at once — structuring work so tasks overlap in time; parallelism means they literally run at the same instant.
45. **Conflict** (git) — two edits to the same lines — a merge where the same region changed on both sides, leaving markers for a human to resolve.
46. **Constant folding** — doing the maths at compile time — a compiler optimization evaluating constant expressions during compilation.
47. **Container** — an isolated box for one program and its files — a process group isolated by kernel names-

paces and limited by cgroups, sharing the host kernel.

48. **Container** (media) — the wrapper holding audio and video together — a file format such as MP4, MKV or WebM carrying encoded streams, timing and metadata.
49. **Contamination** — a model already saw the exam — benchmark test data appearing in training data, inflating scores without matching ability.
50. **Context switch** — the CPU swapping one task for another — saving one thread's registers and state, then loading another's, costing roughly one to a few microseconds.
51. **Context window** — how much a model can read at once — the maximum number of tokens a model may attend over in one request, including its own output.
52. **Continuous delivery** — every good build is ready to ship — an automated pipeline that keeps the main branch in a deployable state, with release as a decision.
53. **Continuous integration** — merging often, tested every time — the practice of integrating work frequently, with an automated build and test on every change.
54. **Control plane** — the part that decides — the systems computing routes, policy and configuration, as opposed to the data plane that moves packets.
55. **Cookie** — a small note a site asks your browser to keep — a name-value pair stored by the browser and returned on matching requests, defined by RFC 6265.
56. **Copy-on-write** — share until someone writes — mapping the same pages into several users and duplicating a page only when one of them modifies it.
57. **Core** — one complete processor inside a chip — an independent set of execution units, registers and L1 caches able to run its own instruction stream.
58. **Coroutine** — a function that can pause and resume — a routine with suspendable execution state, allowing cooperative multitasking without OS threads.
59. **CORS** — a site giving permission for another site to read it — Cross-Origin Resource Sharing: response headers letting a browser relax the same-origin policy.
60. **CPU** — the part that follows the instructions — central processing unit, the general-purpose processor fetching, decoding and executing instructions.
61. **CPU-bound** — limited by thinking, not waiting — a workload whose completion time is set by processor throughput rather than I/O.
62. **CRC** — a strong checksum for detecting damage — cyclic redundancy check, polynomial division over GF(2); CRC-32 protects Ethernet frames.
63. **Cross-compilation** — building on one machine for another — producing binaries for a target architecture or OS different from the build host.
64. **Cryptographic hash** — a fingerprint of any data — a one-way function producing a fixed-size digest, where finding collisions is computationally infeasible.
65. **CSMA/CA** — listen before you transmit, and back off — Carrier Sense Multiple Access with Collision Avoidance, the Wi-Fi access method with random backoff.
66. **CSRF** — a site tricking your browser into using your login — cross-site request forgery, defeated by anti-forgery tokens and SameSite cookies.
67. **CSS** — the rules that decide how a page looks — Cascading Style Sheets: selectors and declarations applied by cascade, specificity and inheritance.
68. **Culling** — not drawing what nobody can see — discarding geometry outside the view frustum, facing away, or hidden behind other objects.
69. **Current** — the flow of electricity — the rate of charge flow, measured in amperes; one ampere is one coulomb per second.
70. **Czochralski** — the way a silicon crystal is grown — dipping a seed crystal into molten silicon and slowly pulling and rotating it to form a single-crystal boule.

53.4.4 D

1. **DAC** — turns numbers back into a real signal — digital-to-analogue converter, reconstructing a voltage from samples at a fixed rate.
2. **Daemon** — a program that runs quietly in the background — a long-lived process with no controlling terminal, usually started at boot.
3. **Datagram** — a message sent with no promises — a self-contained packet delivered without connection setup, ordering or retransmission.
4. **dBm** — signal strength on a logarithmic scale — power relative to one milliwatt; -50 dBm is strong Wi-Fi and -85 dBm is marginal.
5. **DDR** — memory that moves data on both clock edges — double data rate SDRAM; DDR5, standardized 2020, starts at 4800 MT/s.
6. **Dead code** — code that can never run — instructions unreachable on any path, normally removed by the compiler.
7. **Deadlock** — two tasks each waiting for the other — a cycle of hold-and-wait dependencies where no participant can proceed.
8. **Debouncing** — ignoring a switch's noisy chatter — filtering multiple contact transitions into one clean event, typically over 5 to 50 ms.
9. **Decibel** — a way to write huge ratios in small numbers — ten times the base-10 logarithm of a power ratio; 3 dB is roughly double.
10. **Decode** (AI) — writing the answer one token at a time — the autoregressive generation phase, one token per forward pass, memory-bandwidth bound.
11. **Decoder-only** — the transformer shape used by chat models — a stack of masked self-attention blocks with no separate encoder.
12. **Deep learning** — networks with many stacked layers — machine learning using multi-layer neural networks trained by gradient descent.
13. **Default route** — where to send anything you have no rule for — the 0.0.0.0/0 entry, or ::/0 for IPv6, in a routing table.
14. **Defragmentation** — putting a file's pieces back together — rearranging blocks so files occupy contiguous space, useful on disks and pointless on SSDs.
15. **Delta compression** — storing only what changed — encoding an object as a difference against a similar one, as git does inside packfiles.
16. **Demand paging** — loading pages only when touched — bringing a page into memory on its first access, signalled by a page fault.
17. **Dependency** — something your code needs to work — an external library, service or tool required to build or run software.
18. **Dependency injection** — passing in what a thing needs — supplying collaborators from outside rather than constructing them internally.
19. **Detached HEAD** — you are on a commit, not on a branch — HEAD holding a commit hash directly, so new commits belong to no branch.
20. **Deterministic lockstep** — every machine simulates the same thing — sending only inputs and relying on identical simulation on all clients.
21. **DHCP** — the network hands you an address automatically — Dynamic Host Configuration Protocol, RFC 2131, using the DORA exchange to lease settings.
22. **Die** — one chip cut from a wafer — a single rectangle of processed silicon containing a complete circuit.
23. **Dielectric** — the insulating filling in a capacitor — a non-conducting material that polarizes in a field and raises capacitance.
24. **Diffie-Hellman** — agreeing a secret over an open line — key exchange where each side combines its private value with the other's public value.
25. **Diode** — a one-way valve for electricity — a PN junction conducting when forward biased above about 0.7 V in silicon.

26. **Directory** — a folder holding names — a filesystem object mapping names to inode numbers or file records.
27. **Dirty bit** — a mark saying this copy changed — a flag showing a page or cache line differs from the backing store and must be written back.
28. **Disk image** — a file holding a whole drive's contents — a byte-for-byte copy of a block device, including partition table and filesystems.
29. **Distributed system** — several machines pretending to be one — a system whose components run on networked machines and coordinate by message passing.
30. **Distro** — one packaged version of Linux — a distribution bundling a kernel, libraries, package manager and defaults, such as Debian or Fedora.
31. **DMA** — devices moving data without bothering the CPU — direct memory access, where a controller transfers to or from memory and interrupts on completion.
32. **DNS** — the phone book turning names into addresses — a distributed hierarchical database of resource records, RFC 1034 and 1035.
33. **DNS resolver** — the helper that does the lookups for you — a recursive server that queries root, TLD and authoritative servers and caches results.
34. **Docker** — the tool that made containers easy — a container engine with an image format, build file and registry workflow, released 2013.
35. **Docker image** — a frozen filesystem plus start instructions — a layered, content-addressed bundle of filesystem layers and configuration metadata.
36. **DOM** — the page as a tree the code can change — Document Object Model, the browser's live object tree built from parsed HTML.
37. **Doping** — adding impurities to make silicon useful — introducing donors such as phosphorus or acceptors such as boron to make n-type or p-type material.
38. **DPI** — dots per inch on paper — printer resolution; PPI is the screen equivalent, pixels per inch.
39. **DPO** — training from preferences without a reward model — direct preference optimization, fitting the policy directly to pairs of preferred and rejected answers.
40. **DRAM** — the main memory that needs constant refreshing — dynamic RAM storing a bit as charge on a capacitor, refreshed roughly every 32 or 64 ms.
41. **Draw call** — one order sent to the graphics card — a command to render a batch of geometry with one state and shader set.
42. **Driver** — the code that speaks a device's language — kernel or user-space software translating generic requests into device-specific commands.
43. **DROP** — silently binning a packet — a firewall action discarding a packet with no reply, so the sender only sees a timeout.
44. **Duplex** — whether both ends can talk at once — full duplex allows simultaneous send and receive; half duplex allows only one direction at a time.
45. **Dynamic linking** — libraries joined at start-up, not build time — resolving symbols at load or first use, sharing one library copy across processes.

53.4.5 E

1. **eBPF** — running safe little programs inside the kernel — an in-kernel virtual machine with a verifier, used for tracing, networking and security.
2. **ECC** (memory) — memory that corrects its own small errors — error-correcting code memory, typically SECDED, fixing single-bit and detecting double-bit errors.
3. **ECC** (crypto) — strong keys that are short — elliptic curve cryptography; a 256-bit curve key is roughly as strong as a 3072-bit RSA key.
4. **ECMP** — spreading traffic over several equally good paths — equal-cost multipath, hashing a flow's five-tuple to pick one of several next hops.
5. **Edge case** — the odd input that breaks things — a condition at the boundary of valid input, often untested.

6. **Edge computing** — doing work near the user instead of far away — running compute at points of presence close to clients to cut latency.
7. **Elasticity** — capacity that grows and shrinks with demand — automatic scaling of resources in response to load, with matching cost changes.
8. **ELF** — the executable format on Linux — Executable and Linkable Format, with a header, program headers for loading and section headers for linking.
9. **Embedding** — a meaning turned into a list of numbers — a learned dense vector whose geometry places related items close together.
10. **Emulation** — one machine pretending to be another — software reproducing another system's instruction set and devices closely enough to run its software.
11. **Encapsulation** (networks) — wrapping one message inside another — placing a higher-layer unit into a lower-layer payload with its own header.
12. **Encoder** — the half that reads rather than writes — a transformer stack producing representations of an input without generating tokens.
13. **Endianness** — which end of a number comes first — byte order in memory; little-endian stores the least significant byte at the lowest address.
14. **Endpoint** — one address a service answers on — a specific URL and method combination, or a network address plus port.
15. **Entropy** — how surprising the data is — the average information per symbol in bits, the floor for lossless compression.
16. **Environment variable** — a setting passed to a program at start — a name-value pair inherited through the process environment, such as PATH or HOME.
17. **Ephemeral port** — the temporary port your side uses — a short-lived source port, usually from 32768 to 60999 on Linux, chosen per connection.
18. **Epoch** (training) — one full pass through the data — one complete presentation of the training set to the learner.
19. **Epoch** (time) — the moment counting starts — the Unix epoch, 00:00:00 UTC on 1 January 1970, the zero point for time_t.
20. **Error budget** — the amount of failure you are allowed — the difference between 100 percent and an SLO, spendable on risky changes.
21. **Etching** — cutting the pattern into the wafer — removing material chemically or with plasma where photoresist does not protect it.
22. **Ethernet** — the wired local network everyone uses — IEEE 802.3 framing and access rules, from 10 Mbit/s in 1983 to 800 Gbit/s today.
23. **EUV** — the light used to print the smallest features — extreme ultraviolet lithography at 13.5 nm wavelength, in production since 2019.
24. **Eventual consistency** — everyone agrees, given a moment — a model where replicas converge if updates stop, without guaranteeing immediate agreement.
25. **Exception** — an error that jumps out of normal flow — a signalled condition transferring control to a handler and unwinding the stack.
26. **Exit code** — the number a program leaves behind — a process's status, 0 for success by convention and 1 to 255 for failure.
27. **Exponent** — the part of a float that sets the scale — the biased power-of-two field in IEEE 754, 8 bits in binary32 and 11 in binary64.
28. **Extension** — the bit after the dot in a filename — a naming convention, not a guarantee; the real type comes from content or magic numbers.

53.4.6 F

1. **Fan-out** — how many things one thing drives — the number of inputs a single output feeds, limiting speed in logic and load in messaging.
2. **Fast-forward** — the branch just slides forward — a merge where the target is an ancestor of the source, so the ref simply moves, creating no commit.
3. **Fault tolerance** — carrying on despite broken parts — designing so component failure degrades service instead of stopping it.
4. **Feature flag** — a switch that turns a feature on or off — configuration allowing behaviour to change at runtime without redeploying.
5. **Fetch** (git) — download new objects but change nothing — updating remote-tracking refs and objects without touching your branches or working tree.
6. **FIFO** — first in, first out — a queue discipline, or a named pipe file created with mkfifo.
7. **File descriptor** — the small number a program uses to name an open file — a per-process index into the kernel's open file table; 0, 1 and 2 are stdin, stdout, stderr.
8. **File handle** — the object behind an open file — the kernel structure holding the file's position, mode and reference count.
9. **Filesystem** — the way files are laid out on a device — the on-disk structures and rules holding names, metadata and data blocks, such as ext4, APFS or NTFS.
10. **Fine-tuning** — extra training on your own examples — continued training of a pretrained model on a smaller task-specific dataset.
11. **FinFET** — a transistor with a fin sticking up — a 3D transistor whose gate wraps a thin silicon fin, in production from 2011 to about the 3 nm class.
12. **Firewall** — a guard deciding which packets may pass — a device or kernel subsystem filtering traffic by rules over addresses, ports, state and content.
13. **Firmware** — software living in the device itself — code stored in non-volatile memory on a device, run before or beneath the operating system.
14. **Firmware update** — replacing the software inside a device — writing new code to a device's flash, usually with A/B slots and a signature check.
15. **Fixed timestep** — the game's physics ticks at one steady rate — advancing simulation in constant increments regardless of frame time, with interpolation for display.
16. **Flash memory** — storage that keeps data with no power — floating-gate or charge-trap cells holding charge, erased in blocks and written in pages.
17. **Flip-flop** — a circuit that remembers one bit — an edge-triggered storage element capturing its input at a clock edge.
18. **Floating point** — a number with a moving decimal point — a sign, exponent and fraction encoding under IEEE 754, trading precision for range.
19. **Flow control** — stopping a fast sender flooding a slow receiver — TCP's advertised receive window, or the equivalent at other layers.
20. **Flush** — pushing buffered data out now — forcing pending writes from a buffer to the next level, such as fsync to storage.
21. **Fork** (process) — making a copy of the running program — the Unix call creating a child process with a copied address space, usually copy-on-write.
22. **Fork** (repository) — your own copy of someone's project — a server-side clone under your account, with its own refs and pull-request relationship.
23. **Forward proxy** — a middleman your client chooses — a server making requests on a client's behalf, often for policy or caching.
24. **Forward secrecy** — yesterday's traffic stays safe if today's key leaks — using ephemeral key exchange so long-term key compromise cannot decrypt past sessions.
25. **Fragment** (IP) — a packet cut up to fit — an IP packet split to fit a link's MTU, reassembled at the destination.

26. **Frame** (network) — the layer 2 envelope — the Ethernet or Wi-Fi unit with MAC addresses, payload and a frame check sequence.
27. **Frame** (graphics) — one complete picture — one full rendered image; at 60 Hz the frame budget is 16.67 ms.
28. **Frame budget** — the time you have for one picture — the per-frame time allowance, 16.67 ms at 60 Hz and 6.94 ms at 144 Hz.
29. **Frame rate** — pictures per second — displayed frames per second; smoothness also depends on frame time consistency, not the average alone.
30. **Framebuffer** — the memory holding the picture to be shown — a region of memory holding pixel values scanned out to a display.
31. **Free software** — software you may run, study, change and share — software distributed with the four freedoms, usually under a copyleft licence such as the GPL.
32. **Frequency** — how many cycles per second — cycles per second in hertz; a 4 GHz clock has a period of 250 picoseconds.
33. **FTL** — the SSD's private map of where data really is — flash translation layer, mapping logical block addresses to physical pages and handling wear and garbage collection.
34. **Full-duplex** — talking and listening at the same time — simultaneous bidirectional transmission on separate paths or with echo cancellation.
35. **Function call** — jumping into a named block and coming back — transferring control with arguments and a return address, following the ABI's calling convention.
36. **Fuzzing** — throwing random rubbish at a program to break it — automated testing with mutated or generated inputs, watching for crashes and assertion failures.

53.4.7 G

1. **Game loop** — read input, update, draw, repeat — the cycle sampling input, advancing simulation and rendering, once per frame.
2. **Gamma** — the curve between number and brightness — a nonlinear transfer function matching encoding to perception, roughly 2.2 for sRGB.
3. **Garbage collection** — memory tidied up automatically — a runtime reclaiming unreachable objects, by tracing or reference counting.
4. **Gate** (logic) — a tiny circuit applying one logical rule — a circuit implementing AND, OR, NOT, NAND, NOR or XOR over binary inputs.
5. **Gate** (transistor) — the leg that switches the flow — the control terminal whose voltage forms or removes the channel between source and drain.
6. **Gateway** — the way out of your local network — the router a host sends to when the destination is not on its own subnet.
7. **GATT** — how Bluetooth devices expose small values — Generic Attribute Profile, a hierarchy of services, characteristics and descriptors addressed by handle.
8. **GDDR** — memory built for graphics cards — graphics DDR SDRAM tuned for very high bandwidth with wide buses and high clock rates.
9. **Gigabit** — a thousand million bits — 10^9 bits; note that 1 Gbit/s is 125 MB/s before protocol overhead.
10. **GIL** — only one thread runs Python bytecode at a time — CPython's global interpreter lock, serializing bytecode execution across threads.
11. **Git** — the version control system almost everyone uses — a content-addressed distributed VCS created by Linus Torvalds in April 2005.
12. **Golden image** — the trusted base copy you build from — a validated machine or container image used as the starting point for deployments.
13. **Goodput** — the useful data rate, not the raw one — application-level throughput after headers, retransmissions and protocol overhead.
14. **Goroutine** — a very cheap thread in Go — a lightweight task multiplexed onto OS threads by the Go runtime scheduler.

15. **GPIO** — a pin you can set high or low yourself — general-purpose input/output, a configurable digital pin on a microcontroller or board.
16. **GPT** (partition) — the modern partition table — GUID Partition Table, part of UEFI, supporting large disks and 128 partitions by default.
17. **GPT** (model) — a generative pretrained transformer — a decoder-only transformer trained to predict the next token.
18. **GPU** — the chip built for many small sums at once — a processor with thousands of simple lanes, built for throughput on parallel work.
19. **Graceful degradation** — losing features instead of falling over — designing so partial failure reduces function rather than causing an outage.
20. **Gradient clipping** — stopping an update from being too big — rescaling gradients whose norm exceeds a threshold, to keep training stable.
21. **Gradient descent** — walking downhill on the error — updating parameters against the gradient of the loss, scaled by the learning rate.
22. **GraphQL** — the client asks for exactly the fields it wants — a query language and runtime with one endpoint, a typed schema and client-shaped responses.
23. **Ground** — the reference point voltages are measured from — the common return path defined as zero volts in a circuit.
24. **gRPC** — fast structured calls between services — a remote procedure call framework using Protocol Buffers over HTTP/2, with streaming support.
25. **Guard clause** — checking the bad case first and leaving — an early return that removes nesting from the main path.
26. **GUI** — pointing at pictures instead of typing commands — a graphical user interface with windows, icons, menus and a pointer.

53.4.8 H

1. **Hallucination** — a model stating something untrue with confidence — fluent output not grounded in training data or provided context.
2. **Handover** — your phone moving to the next mast without dropping — transferring an active connection between cells while keeping the session alive.
3. **Handshake** — the greeting that sets up a connection — the exchange establishing parameters, such as TCP's SYN, SYN-ACK, ACK.
4. **Hard link** — two names for the same file — a second directory entry pointing at the same inode, with a shared link count.
5. **Hash table** — a lookup that is fast because it computes where to look — a map using a hash of the key to index buckets, average O(1) lookup.
6. **HBM** — memory stacked right next to the processor — high bandwidth memory, stacked DRAM dies on an interposer, giving terabytes per second.
7. **HDD** — a drive with spinning magnetic platters — a hard disk with rotating platters and moving heads, giving milliseconds of seek latency.
8. **HDR** — brighter brights and darker darks — high dynamic range imaging with wider luminance range, usually with a PQ or HLG transfer function.
9. **HEAD** — where you are in the repository right now — a ref pointing to the current branch, or directly to a commit when detached.
10. **Header** — the label on the front of a message — the fixed or structured prefix carrying addressing and control information before the payload.
11. **Heap** — memory you ask for and give back yourself — the region for dynamic allocation, managed by the allocator rather than by scope.
12. **Heat sink** — metal that carries heat away — a finned metal block spreading and radiating heat from a package to moving air or liquid.
13. **Hertz** — cycles per second — the SI unit of frequency, one cycle per second.

14. **Hexadecimal** — counting in sixteens — base-16 notation using 0 to 9 and A to F, where one digit encodes exactly four bits.
15. **Hit rate** — how often the cache had it — the fraction of requests served from cache, the main determinant of average latency.
16. **Hop** — one router along the way — a single forwarding step; traceroute reveals hops by increasing the TTL.
17. **Hotfix** — an urgent small change — a minimal patch shipped outside the normal release cycle to fix a live problem.
18. **HTML** — the markup that says what parts a page has — HyperText Markup Language, a tree of elements the browser parses into the DOM.
19. **HTTP** — the rules browsers and servers use to talk — a request-response protocol with methods, status codes and headers, now RFC 9110.
20. **HTTPS** — HTTP inside an encrypted tunnel — HTTP carried over TLS, normally on port 443, giving confidentiality, integrity and server authentication.
21. **Huffman coding** — short codes for common symbols — an optimal prefix-free code built from symbol frequencies, used inside DEFLATE and JPEG.
22. **Hyperparameter** — a setting you choose before training — a value not learned by gradient descent, such as learning rate, batch size or layer count.
23. **Hyper-threading** — one core pretending to be two — simultaneous multithreading, running two instruction streams over one core's execution resources.
24. **Hypervisor** — software that runs whole machines inside one machine — a layer creating and scheduling virtual machines, type 1 on bare metal or type 2 on a host OS.

53.4.9 I

1. **IaaS** — renting raw machines and networks — infrastructure as a service, where you manage the OS and everything above it.
2. **IAM** — who is allowed to do what — identity and access management: principals, policies, roles and permission evaluation.
3. **ICMP** — the network's error and echo messages — the control protocol carrying unreachable, time exceeded and echo, RFC 792.
4. **Idempotent** — doing it twice changes nothing more — an operation whose repeated application has the same effect as one application.
5. **Idle** — doing nothing on purpose — a scheduler state with no runnable task, usually entering a low-power CPU state.
6. **IEEE 754** — the rule that makes floating point the same everywhere — the 1985 standard defining binary32, binary64, rounding modes, infinities and NaN.
7. **Image** (container) — a frozen filesystem plus start-up settings — a layered, content-addressed bundle of layers and configuration.
8. **IMEI** — the serial number identifying a phone itself — a 15-digit equipment identity, distinct from the SIM's subscriber identity.
9. **Immutable** — it cannot be changed after it is made — a value or object whose state is fixed at creation, simplifying sharing and caching.
10. **Impedance** — how much a component resists an alternating signal — the complex ratio of voltage to current; speakers are commonly 4 or 8 ohms.
11. **Index** (git) — the staging area between edits and a commit — a binary file listing paths, modes and blob hashes for the next commit.
12. **Inductor** — a coil that resists sudden changes in current — a component storing energy in a magnetic field, with voltage proportional to di/dt.
13. **Inference** — using a trained model to get an answer — the forward pass only, with no gradients and no parameter updates.
14. **Infrastructure as code** — servers described in files, not clicked into existence — declarative resource

definitions kept in version control and applied by a tool.

15. **init** — the first process the kernel starts — PID 1, which starts services and adopts orphaned processes.
16. **Inode** — the record holding a file's facts — a filesystem structure with mode, owner, times, size and block pointers, but not the name.
17. **Instruction pipeline** — several instructions in progress at once — overlapping fetch, decode, execute and write-back stages to raise throughput.
18. **Instruction set** — the list of operations a CPU understands — the ISA: opcodes, registers, addressing modes and the binary encoding.
19. **Instruction tuning** — teaching a model to follow requests — supervised fine-tuning on instruction-and-response pairs.
20. **Integer overflow** — a number too big for its box — a result exceeding the type's range, wrapping in unsigned types and undefined for signed C.
21. **Integration test** — checking that the parts work together — a test exercising several components and their real interfaces.
22. **Interface** (network) — one connection point on a machine — a NIC or virtual device with its own MAC address, addresses and statistics.
23. **Intermediate representation** — the compiler's private middle language — an IR such as LLVM IR, easier to optimize than source or machine code.
24. **Interpreter** — runs source code directly, line by line — a program executing a source or bytecode representation without producing a native binary.
25. **Interrupt** — a device tapping the CPU on the shoulder — an asynchronous signal causing the CPU to jump to a handler and return.
26. **Interrupt latency** — how long before the handler starts — the delay from signal assertion to the first handler instruction.
27. **IOMMU** — an address translator for devices — hardware mapping and restricting device DMA addresses, essential for safe passthrough.
28. **IP address** — the number that identifies a machine on a network — a 32-bit IPv4 or 128-bit IPv6 address, assigned to an interface, not a device.
29. **IPC** (CPU) — instructions finished per clock tick — instructions per cycle, the other half of performance beside frequency.
30. **IPC** (software) — how two programs talk — inter-process communication by pipes, sockets, shared memory or message queues.
31. **IPv4** — the old 32-bit address scheme — RFC 791 addressing, about 4.3 billion addresses, exhausted at IANA in 2011.
32. **IPv6** — the 128-bit replacement — RFC 8200 addressing with 2^128 addresses, no header checksum and no router fragmentation.
33. **IXP** — a building where networks meet and swap traffic — an internet exchange point with a shared switch fabric for peering.

53.4.10 J

1. **JavaScript** — the language that runs inside web pages — a dynamically typed, event-driven language standardized as ECMAScript.
2. **JIT** — compiling while the program runs — just-in-time compilation of hot paths to native code, guided by runtime profiles.
3. **Jitter** — how much the delay wobbles — variation in packet delay, more damaging than average latency for voice and games.
4. **Job** (CI) — one unit of work in a pipeline — a set of steps run on one runner, with its own environment and result.
5. **Job control** — putting a command in the background — shell management of process groups with foreground, background and stopped states.
6. **Journaling** — writing down the plan before doing it — a filesystem log of intended metadata changes, replayed after a crash.
7. **JSON** — a simple text format for structured data — JavaScript Object Notation: objects, arrays, strings, numbers, booleans and null.
8. **JWT** — a signed token you can read yourself — JSON Web Token, RFC 7519, a base64url header, payload and signature joined by dots.

53.4.11 K

1. **Kernel** (OS) — the part of the system with full power over the machine — the code running in privileged mode, owning scheduling, memory and devices.
2. **Kernel** (maths) — the small window slid over data — a filter matrix applied by convolution across an image or signal.
3. **Kernel mode** — the privileged half of the processor — the state allowing all instructions and address ranges, entered by trap or interrupt.
4. **Kernel panic** — the system giving up — an unrecoverable kernel error that halts the machine rather than risk corruption.
5. **Key** (attention) — what each token offers to be matched against — a projected vector compared with queries to produce attention weights.
6. **Key derivation** — turning a password into a proper key — a KDF such as PBKDF2, scrypt or Argon2 stretching a secret into key material.
7. **Key exchange** — agreeing a secret without posting it — a protocol such as Diffie-Hellman letting both sides derive a shared key.
8. **Keyframe** — a video frame that stands alone — an intra-coded frame decodable without reference to others, the seek point in a stream.
9. **Kill** — telling a process to stop — sending a signal such as SIGTERM to request exit or SIGKILL to force it.
10. **Kilobyte** — a thousand bytes — 1000 bytes in SI usage; 1024 bytes is a kibibyte, KiB.
11. **Knowledge cutoff** — the date a model's reading stops — the point beyond which pretraining data was not collected.
12. **Kubernetes** — the system that keeps containers running — a container orchestrator with a declarative API and reconciling controllers.
13. **KV cache** — remembering earlier keys and values so they are not redone — stored per-layer key and value tensors for tokens already processed.

53.4.12 L

1. **Lag compensation** — the server rewinding time to judge your shot — re-evaluating a hit against the world state the shooter actually saw.
2. **Latch** — a memory circuit that follows while enabled — a level-sensitive storage element, unlike an edge-triggered flip-flop.

3. **Latency** — the waiting time before something starts — the delay between cause and effect, measured end to end and reported as percentiles.
4. **LBA** — numbering blocks straight through — logical block addressing, a linear block number replacing cylinder, head and sector.
5. **LCD** — a screen where a backlight shines through gates — liquid crystal display, twisting crystals to control light per subpixel.
6. **Learning rate** — how big a step training takes — the scale factor on the gradient, the most sensitive hyperparameter.
7. **Least privilege** — give only the rights actually needed — restricting each principal to the minimum permissions for its task.
8. **Lexer** — the part that cuts source code into words — the tokenizer producing a stream of tokens from characters.
9. **LFS** — keeping big files out of git's history — Large File Storage, replacing large files with pointers and storing content separately.
10. **Library** — reusable code you call — a packaged collection of functions linked statically or loaded dynamically.
11. **Linked list** — items each pointing at the next — a sequence of nodes with pointers, cheap to insert and slow to index.
12. **Linker** — joins compiled pieces into one program — the tool resolving symbols, laying out sections and producing an executable or shared object.
13. **Little-endian** — the smallest byte first — byte order storing the least significant byte at the lowest address, as x86 and ARM do.
14. **Load balancer** — spreading requests across servers — a device or service distributing connections by round robin, least connections or hashing.
15. **Loader** — the code that puts a program into memory — the kernel or dynamic loader mapping segments, resolving symbols and jumping to the entry point.
16. **localhost** — the name a machine uses for itself — the hostname resolving to 127.0.0.1 and ::1, never leaving the machine.
17. **Locality** — recently or nearby data is likely used again — temporal and spatial locality, the reason caches work.
18. **Lock** — one at a time, please — a synchronization primitive giving exclusive access to shared state.
19. **LOD** — simpler models for distant objects — level of detail, swapping meshes or textures by screen size to save work.
20. **Log** — the running record of what happened — an append-only stream of events used for debugging, audit and recovery.
21. **Logic gate** — a tiny circuit applying one rule — a combinational circuit implementing a Boolean function of its inputs.
22. **Logit** — the raw score before it becomes a probability — the unnormalized output per vocabulary entry, fed to softmax.
23. **Loopback** — the machine talking to itself — the virtual interface for 127.0.0.0/8 and ::1, never leaving the host.
24. **LoRA** — training a small add-on instead of the whole model — low-rank adaptation, learning two small matrices whose product updates a frozen weight.
25. **Loss function** — the score that says how wrong the model is — the objective minimized in training, typically cross-entropy for language models.
26. **Lossless** — nothing is thrown away — compression from which the exact original bytes can be restored, as in FLAC, PNG and gzip.
27. **Lossy** — some detail is thrown away for size — compression discarding information judged imperceptible, as in JPEG, MP3 and H.264.
28. **LRU** — throw out what was used longest ago — least recently used, a common cache replacement policy.

29. **LTE** — the fourth generation mobile standard — 3GPP Long Term Evolution, an all-IP radio network using OFDMA downlink.

53.4.13 M

1. **MAC** (crypto) — a tag proving a message was not altered — message authentication code, such as HMAC, computed with a shared secret key.
2. **MAC address** — the hardware name of a network port — a 48-bit link-layer address, the first 24 bits identifying the manufacturer.
3. **Mach-O** — the executable format on macOS — Apple's binary format, with load commands and support for fat binaries holding several architectures.
4. **Magic number** — the few bytes at the start that say what a file is — a fixed signature such as 0x7F ELF or PK for zip, used by the file command.
5. **Malloc** — asking for memory while running — the C library call returning a pointer to an uninitialized block of the requested size.
6. **MBR** — the old partition table in the first sector — master boot record, four primary partitions and a 2 TiB limit.
7. **Memory leak** — memory taken and never given back — allocated memory that stays reachable but unused, growing the process footprint.
8. **Merge** — joining two lines of work — a commit with two or more parents, combining changes since the merge base.
9. **Merge base** — the last commit both branches share — the best common ancestor used as the reference point in a three-way merge.
10. **Merge queue** — testing changes in the order they will land — a forge feature building each pull request against the real future main branch.
11. **mergeStateStatus** — a forge's summary of whether a merge is possible — GitHub's field with values such as CLEAN, BLOCKED, BEHIND, DIRTY or UNKNOWN.
12. **MESI** — the four states a cache line can be in — modified, exclusive, shared, invalid: the coherence protocol keeping cores consistent.
13. **Metadata** — data about the data — attributes such as size, times, type and permissions, held separately from content.
14. **Microkernel** — a tiny kernel with drivers outside it — a design keeping only scheduling, memory and IPC privileged, with services in user space.
15. **Microservice** — a small service that does one job — an independently deployable service with its own data and interface.
16. **Middleware** — code sitting between request and handler — a chain of components processing requests before or after the application.
17. **Migration** (data) — changing the shape of stored data — a versioned script altering schema or contents, applied in order and recorded.
18. **MIMO** — several antennas sending several streams — multiple input multiple output, using spatial paths to multiply capacity.
19. **Mipmap** — smaller pre-made copies of a texture — a chain of half-size texture levels selected by screen footprint to avoid shimmering.
20. **Mixture of experts** — only part of the model runs per token — a layer routing each token to a few expert subnetworks, so active parameters are fewer than total.
21. **MMU** — the hardware that translates addresses — memory management unit, walking page tables to map virtual to physical addresses.
22. **Mode** (file) — the permission bits on a file — the 12-bit field holding owner, group and other permissions plus setuid, setgid and sticky.
23. **Model card** — the label describing what a model is and is not — a document stating intended use, training data summary, evaluations and limitations.
24. **Modulation** — changing a wave so it carries data — varying amplitude, frequency or phase to encode

symbols on a carrier.

25. **Monolithic kernel** — one big kernel with drivers inside — a design running services and drivers in kernel mode, as Linux does.
26. **Monorepo** — everything in one repository — a single repository holding many projects, with shared tooling and atomic cross-project changes.
27. **Moore's law** — transistors per chip keep doubling — Gordon Moore's 1965 observation, revised in 1975 to a doubling roughly every two years.
28. **MOSFET** — the switch nearly all chips are made from — metal-oxide-semiconductor field-effect transistor, whose gate voltage forms a channel.
29. **Mount** — attaching a filesystem into the tree — making a device or remote share visible at a directory path.
30. **Mount point** — the directory where a filesystem appears — the path at which a filesystem's root is grafted into the namespace.
31. **MTU** — the biggest packet a link will carry — maximum transmission unit, 1500 bytes on standard Ethernet.
32. **Multicast** — one send, many listeners — delivery to a group address, where the network duplicates packets only where paths diverge.
33. **Multi-head attention** — several attention views at once — running attention in parallel subspaces and concatenating the results.
34. **Multiplexing** — many conversations over one link — interleaving several logical streams on one physical or transport channel.
35. **Mutation testing** — checking that tests would notice a bug — deliberately altering code to see whether the test suite fails.
36. **Mutex** — a lock only one holder may take — mutual exclusion primitive with owner tracking, blocking other threads until released.

53.4.14 N

1. **Namespace** (Linux) — a private view of a system resource — a kernel feature isolating mounts, PIDs, network stacks, users or hostnames per process group.
2. **NaN** — the float that means "not a number" — an IEEE 754 value produced by invalid operations, unequal even to itself.
3. **NAND flash** — the storage in phones and SSDs — non-volatile memory of series-connected cells, written per page and erased per block.
4. **NAT** — many private machines behind one public address — network address translation, rewriting addresses and ports and tracking the mapping.
5. **Netcode** — the part of a game that handles the network — the synchronization design: tick rate, prediction, interpolation and reconciliation.
6. **Netmask** — which part of an address is the network — a bit mask marking the prefix, written as 255.255.255.0 or as /24.
7. **Network interface** — the port a machine uses to reach a network — a physical or virtual device with a MAC address and its own addresses.
8. **Neural network** — layers of weighted sums with bends between them — a differentiable function of stacked linear maps and nonlinearities, fitted by gradient descent.
9. **Next hop** — the router to hand this packet to — the neighbouring address a routing table entry sends matching traffic to.
10. **Nginx** — a common web server and reverse proxy — an event-driven server handling many connections with few processes.
11. **Nibble** — half a byte — four bits, exactly one hexadecimal digit.
12. **NIC** — the network card — network interface controller, handling framing, checksums and often offloads.
13. **Node** (compute) — one machine in a cluster — a host contributing CPU, memory and storage, managed

by the cluster's scheduler.
14. **Node** (process) — the transistor generation name — a marketing label such as 5 nm or 3 nm, no longer matching any physical dimension.
15. **Nonce** — a number used exactly once — a value ensuring freshness or uniqueness, never repeated under one key.
16. **Non-volatile** — it remembers with the power off — storage retaining state without power, unlike DRAM and SRAM.
17. **Normalization** (AI) — keeping numbers in a sensible range — layer or RMS normalization rescaling activations to stabilize training.
18. **Notification** (interrupt) — see interrupt — an asynchronous event delivered to software rather than polled for.
19. **NTP** — how machines agree what time it is — Network Time Protocol, disciplining clocks to within milliseconds over the internet.
20. **Null pointer** — a pointer deliberately pointing at nothing — the reserved address zero, whose dereference traps on most systems.
21. **NUMA** — memory that is closer to some cores than others — non-uniform memory access, where latency depends on which node owns the memory.
22. **NVMe** — the protocol built for flash, not for disks — a command set over PCIe with deep parallel queues and very low overhead.
23. **NVRAM** — small memory that survives a reboot — non-volatile RAM used for firmware settings and boot variables.
24. **Nyquist** — sample at least twice the highest frequency — the sampling theorem requiring a rate above twice the signal bandwidth to avoid aliasing.

53.4.15 O

1. **OAuth** — letting an app act for you without your password — a delegation framework issuing scoped access tokens, RFC 6749.
2. **Object file** — compiled code not yet joined up — a relocatable binary with sections, symbols and relocation entries.
3. **Observability** — being able to tell what a system is doing — the ability to answer new questions from existing logs, metrics and traces.
4. **OFDM** — splitting a channel into many narrow lanes — orthogonal frequency division multiplexing, used by Wi-Fi, LTE, 5G and DSL.
5. **Ohm's law** — push equals flow times resistance — V = IR, relating voltage, current and resistance in a linear element.
6. **OLED** — a screen where each dot makes its own light — organic light-emitting diode panels with no backlight, giving true black.
7. **Opcode** — the number that says which operation — the field of an instruction encoding selecting the operation to perform.
8. **Opcode fusion** — merging two instructions into one operation — a CPU front-end optimization combining common instruction pairs.
9. **Open weights** — the trained numbers are published — a model whose parameters are downloadable, which is not the same as open training data.
10. **Orchestration** — deciding what runs where — automated placement, scaling and recovery of workloads across a cluster.
11. **Origin** — the default name for where you cloned from — the conventional remote name git creates for the source repository.
12. **Origin server** — the machine holding the real content — the source a CDN fetches from when it has no valid cached copy.
13. **Orphan** (git) — an object nothing points to — an unreachable object, removable by garbage collection after its grace period.

14. **Oscillator** — the circuit that makes the ticking — a crystal or ring circuit producing a periodic reference signal.
15. **OSI model** — seven named layers for teaching — ISO 7498, from physical up to application, useful as vocabulary, not as an implementation.
16. **Out-of-order** — doing ready work first, then tidying up — executing instructions as operands become available, retiring them in program order.
17. **Overclocking** — running a chip faster than rated — raising frequency and often voltage beyond specification, at the cost of heat and stability.
18. **Overfitting** — memorizing the answers instead of learning — fitting noise in training data, shown by training loss falling while validation loss rises.
19. **Overlay network** — a network built on top of another — virtual links tunnelled over an existing network, as with VXLAN or WireGuard.
20. **Overprovisioning** — spare flash the drive keeps hidden — reserved capacity that improves garbage collection and endurance.

53.4.16 P

1. **PaaS** — you bring the code, they run everything else — platform as a service, managing runtime, scaling and patching for you.
2. **Packaging** (chips) — putting the die in something you can solder — attaching a die to a substrate with bumps or wires and encapsulating it.
3. **Packet** — one labelled chunk of data — a network-layer unit of header plus payload, routed independently.
4. **Packet loss** — a packet that never arrives — discard caused by congestion, errors or policy, inferred from missing acknowledgements.
5. **Packfile** — many git objects compressed into one file — a container storing objects with delta compression, plus an index for lookup.
6. **Page** — the unit memory is managed in — a fixed-size block of virtual memory, usually 4 KiB, mapped to a physical frame.
7. **Page fault** — the page was not in memory — a trap when a virtual page has no valid mapping, handled by loading or allocating.
8. **Page table** — the map from virtual to physical pages — a multi-level structure walked by the MMU, cached in the TLB.
9. **Parameter** (AI) — one learned number in the model — a single weight or bias updated by training and stored in the model file.
10. **Parity** — one extra bit to spot an odd error — a bit making the count of ones even or odd, detecting single-bit errors.
11. **Parser** — the part that works out the structure — the component turning tokens into a syntax tree by grammar rules.
12. **Partition** — one labelled region of a drive — a contiguous block range described by an MBR or GPT entry.
13. **Pass@k** — did any of k attempts pass — a benchmark metric giving credit if at least one of k samples is correct.
14. **Passphrase** — a long password made of words — a secret chosen for entropy through length rather than symbol variety.
15. **Patch** — a file describing a change — a unified diff with context lines, applicable to matching source.
16. **PATH** — the list of folders the shell searches — an environment variable of directories scanned in order for a command name.
17. **Payload** — the actual content, minus the wrapping — the data a protocol carries, excluding headers and trailers.
18. **PCIe** — the fast lanes connecting chips inside a computer — PCI Express, point-to-point serial lanes; gen 5 runs 32 GT/s per lane.

19. **PCM** — sound stored as plain numbers — pulse-code modulation, uncompressed samples at a fixed rate and bit depth.
20. **PE** — the executable format on Windows — Portable Executable, with a DOS stub, COFF header and sections such as .text and .rdata.
21. **Peering** — two networks exchanging traffic directly — a bilateral agreement to swap customer routes, often settlement-free.
22. **Permission** — what you are allowed to do to a file — access bits or ACL entries checked by the kernel on open.
23. **Persistent storage** — data that outlives the process — storage surviving restarts, unlike memory or a container's writable layer.
24. **Phase** — where a wave is in its cycle — angular offset in degrees or radians, the basis of phase-shift keying.
25. **Photolithography** — printing the circuit pattern with light — projecting a mask image onto photoresist to define features on a wafer.
26. **Photoresist** — the light-sensitive coating on a wafer — a polymer whose solubility changes on exposure, defining where etching happens.
27. **Piconet** — a small group of Bluetooth devices working together — one primary device and up to seven active secondaries sharing a hopping pattern.
28. **Ping** — asking a host whether it is there — sending ICMP echo requests and timing the replies.
29. **Pipe** — the output of one command becoming the input of the next — a kernel buffer joining one process's stdout to another's stdin.
30. **Pipeline** (CPU) — instructions in a production line — overlapping instruction stages so one finishes per cycle in the ideal case.
31. **Pixel** — one dot of the picture — the smallest addressable element of a raster image or display.
32. **Plist** — the settings file format on Apple systems — a property list, in XML or binary form, holding key-value configuration.
33. **PN junction** — where p-type meets n-type silicon — the boundary forming a depletion region and conducting in only one direction.
34. **Pod** — the smallest deployable unit in Kubernetes — one or more containers sharing a network namespace, address and storage volumes.
35. **Pointer** — a value that holds an address — a variable containing the memory address of another object.
36. **Polling** — asking repeatedly instead of being told — checking a status in a loop rather than waiting for an interrupt or event.
37. **Port** — the number identifying which program on a machine — a 16-bit endpoint identifier in TCP and UDP headers.
38. **Positional encoding** — telling the model where each word sits — added or rotated information giving order to otherwise position-blind attention.
39. **POSIX** — the portable Unix rulebook — IEEE 1003 standards defining system calls, shell and utilities.
40. **PPI** — how tightly packed the dots are — pixels per inch of a display, about 460 on current high-end phones.
41. **Precision** — how finely a number can be written — the count of significant bits or digits a format can represent.
42. **Prefetch** — fetching before it is asked for — hardware or software loading data into cache ahead of predicted use.
43. **Prefill** — reading the whole prompt in one go — the parallel forward pass over input tokens that fills the KV cache, compute bound.
44. **Pretraining** — the long first training on huge text — self-supervised next-token training that produces a base model.
45. **Primary key** — the column that identifies a row — a unique, non-null identifier for a table's rows.
46. **Privilege escalation** — getting more power than you were given — an exploit raising a principal's permissions beyond its intended level.

47. **Process** — one running program with its own memory — an address space, one or more threads, and a table of open file descriptors.
48. **Profiling** — measuring where the time actually goes — sampling or instrumenting a program to attribute cost to code paths.
49. **Prompt injection** — hidden text telling a model to misbehave — untrusted content in a model's context that overrides intended instructions.
50. **Propagation delay** — the time a gate takes to change its output — the interval from input transition to a valid output, tens of picoseconds today.
51. **Protocol** — the agreed rules for a conversation — the message formats, ordering and state machine two parties follow.
52. **Proxy** — a middleman making requests for you — a server that terminates one connection and opens another on the client's behalf.
53. **PTY** — a fake terminal for a program that expects one — pseudo-terminal, a master and slave pair providing terminal semantics without hardware.
54. **Pull request** — asking for your branch to be merged — a forge object linking a source branch, target branch, review and checks.
55. **Push** — sending your commits to a remote — uploading objects and asking the remote to update refs, refused if it is not a fast-forward.
56. **PWM** — a switch flicked fast to fake a level — pulse-width modulation, varying duty cycle to control average power.

53.4.17 Q

1. **QAM** — packing more bits into each radio symbol — quadrature amplitude modulation; 1024-QAM carries 10 bits per symbol.
2. **Quantization** — using coarser numbers to save space — reducing weight precision, for example FP16 to INT8 or 4-bit, with a small accuracy cost.
3. **Query** (attention) — what one token is looking for — a projected vector compared against every key to produce attention weights.
4. **Queue** — a line of waiting work — a FIFO structure decoupling producers from consumers.
5. **QUIC** — a fast transport built on UDP — RFC 9000, giving streams, encryption and connection migration without head-of-line blocking.
6. **Quorum** — enough members agreeing to decide — a majority of replicas required for a consistent read or write.
7. **Quota** — a cap on how much you may use — an enforced limit on storage, requests or resources per account or user.

53.4.18 R

1. **Race condition** — the answer depends on who gets there first — a bug where behaviour depends on the unsynchronized timing of concurrent access.
2. **RAG** — letting a model read documents before answering — retrieval-augmented generation, injecting retrieved passages into the prompt.
3. **RAID** — several drives acting as one, for speed or safety — striping, mirroring or parity across disks; RAID is not a backup.
4. **RAM** — the fast memory that forgets when power stops — volatile random-access memory holding running code and data.
5. **Rebase** — replaying your commits on a new base — creating new commits with the same changes but new parents and new hashes.
6. **Refactoring** — tidying code without changing behaviour — a behaviour-preserving transformation improving structure or clarity.
7. **Reflog** — git's private diary of where refs pointed — a local log of ref updates, letting you recover from a bad reset.

8. **Refresh rate** — how often the screen redraws — display updates per second in hertz, commonly 60, 120 or 144.
9. **Register** — the CPU's own handful of tiny stores — a named storage location inside the core, accessed in a fraction of a cycle.
10. **Registry** (containers) — the shop images are pulled from — a service storing and serving container images by name and digest.
11. **Registry** (Windows) — the central settings database — a hierarchical store of keys and values used by Windows and its applications.
12. **Regularization** — stopping the model memorizing — techniques such as weight decay, dropout or early stopping that limit overfitting.
13. **REJECT** — refusing a packet and saying so — a firewall action sending an ICMP unreachable or TCP RST, so the sender fails fast.
14. **Remote** — a named repository somewhere else — a stored URL plus fetch and push refspecs, such as origin.
15. **Remote-tracking ref** — your last known idea of a remote branch — a local ref under refs/remotes updated only by fetch or push.
16. **Replication** — keeping copies on more than one machine — maintaining synchronized data on several nodes for durability or read scaling.
17. **Repository** — the whole project with all its history — a directory containing the object database, refs and configuration.
18. **Resistance** — how much a material fights the flow — the ratio of voltage to current, in ohms, dissipating power as heat.
19. **Resolution** — how many dots the picture has — the pixel count of an image or display, such as 1920 by 1080.
20. **Resolver** — the helper that does DNS lookups for you — a recursive server querying the hierarchy and caching results by TTL.
21. **REST** — using plain HTTP methods on named resources — an architectural style with stateless requests and uniform verbs over URLs.
22. **Reverse proxy** — a middleman in front of the servers — a server accepting client requests and forwarding them to backends.
23. **RFC 1918** — the address ranges reserved for private use — 10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16, never routed on the internet.
24. **Ring buffer** — a fixed queue that wraps around — a circular array with head and tail indices, overwriting or blocking when full.
25. **RLHF** — training a model on what people preferred — reinforcement learning from human feedback, optimizing against a learned reward model.
26. **Root server** — the servers at the top of DNS — the 13 named root identities serving the root zone, reached by anycast.
27. **Rootkit** — malware that hides itself in the system — code subverting the kernel or firmware to conceal its own presence.
28. **RoPE** — encoding position by rotating the vectors — rotary position embedding, applying a position-dependent rotation to queries and keys.
29. **Round-trip time** — how long there and back takes — the delay from sending to receiving the acknowledgement, driving TCP behaviour.
30. **Router** — the box that forwards between networks — a device choosing an outgoing interface by longest-prefix match on the destination.
31. **Routing table** — the list of where to send what — the forwarding entries of prefix, next hop, interface and metric.
32. **RSA** — an old public-key algorithm based on big numbers — a scheme whose security rests on the difficulty of factoring; keys are 2048 bits or larger.
33. **RST** — a rude end to a TCP connection — a reset flag telling the peer to abandon the connection

immediately.

34. **Runner** — the machine that executes a CI job — an agent that claims jobs from a queue and reports results back.

53.4.19 S

1. **SaaS** — you just use the website — software as a service, run entirely by the provider with no infrastructure for you.
2. **Salt** — a random extra ingredient in a password hash — a unique per-password value stored alongside it, defeating precomputed tables.
3. **Same-origin policy** — one site cannot read another's data — the browser rule isolating documents by scheme, host and port.
4. **Sampling** (AI) — choosing the next word from the odds — drawing from the model's output distribution, shaped by temperature and top-p.
5. **Sampling rate** — how many measurements per second — samples per second; CD audio uses 44,100 Hz.
6. **Sandbox** — a restricted area a program cannot escape — an isolation mechanism limiting a process's syscalls and resources.
7. **SATA** — the older cable standard for drives — Serial ATA, 6 Gbit/s in revision 3, with one command queue.
8. **Scaling laws** — bigger models and more data give predictable gains — empirical power-law relations between compute, data, parameters and loss.
9. **Scheduler** — decides which task runs next — the kernel component choosing runnable threads by policy and priority.
10. **Scope** (OAuth) — the exact list of things a token may do — a string set attached to a token limiting the operations it authorizes.
11. **SDK** — the tools you need to build for a platform — a software development kit of libraries, headers, tools and samples.
12. **Sector** — the smallest addressable piece of a disk — 512 bytes traditionally, 4096 bytes on Advanced Format drives.
13. **Segment** — TCP's unit of data — the transport-layer unit with sequence numbers, flags and payload.
14. **Semantic versioning** — the version number tells you what changed — MAJOR.MINOR.PATCH, where a major bump means a breaking change.
15. **Semaphore** — a counter that limits how many may proceed — a synchronization primitive with wait and signal operations.
16. **Semiconductor** — a material between a conductor and an insulator — a solid such as silicon whose conductivity is controlled by doping, field or light.
17. **Serialization** — turning objects into bytes to send or store — encoding structures into a byte stream, reversed by deserialization.
18. **Serverless** — you write the function, not the server — a model where the platform runs code on demand and bills by invocation and duration.
19. **Session** — the memory of who you are between requests — server or token state associating requests with an authenticated user.
20. **SHA-256** — a common cryptographic fingerprint — a hash producing a 256-bit digest, used in TLS, git and signing.
21. **Shader** — a small program the GPU runs per vertex or pixel — GPU code for stages such as vertex, fragment or compute.
22. **Sharding** — splitting data across several databases — partitioning by key so each shard holds a subset of the rows.
23. **Shebang** — the first line saying which interpreter to use — the two bytes #! plus a path, read by the kernel when running a script.
24. **Shell** — the program that reads your commands — a command interpreter handling expansion, redi-

rection, pipelines and job control.

25. **Signal** (Unix) — a short message telling a process something happened — an asynchronous notification such as SIGINT, SIGTERM or SIGSEGV.
26. **Signal-to-noise ratio** — how loud the message is compared with the hiss — the power ratio of signal to noise, usually in decibels.
27. **SIM** — the card that identifies you to the mobile network — a smart card holding the IMSI and authentication key, or an eSIM profile.
28. **SIMD** — one instruction working on many numbers — single instruction multiple data, as in SSE, AVX and NEON.
29. **Single point of failure** — the one part whose loss stops everything — a component with no redundancy on the critical path.
30. **SLC to QLC** — how many bits are squeezed into one flash cell — single, multi, triple and quad-level cells, trading endurance and speed for density.
31. **Slice** — a view into part of an array — a reference with pointer, length and sometimes capacity, without copying.
32. **SLO** — the promise you hold yourself to — a service level objective, a target for a measured indicator over a window.
33. **SMTP** — how mail servers hand messages to each other — Simple Mail Transfer Protocol, RFC 5321, on ports 25, 465 or 587.
34. **Snapshot** — a frozen picture of data at one moment — a point-in-time copy, often made cheaply by copy-on-write.
35. **Socket** — the endpoint a program reads and writes a connection through — an OS object identified by address, port and protocol.
36. **Softmax** — turning scores into probabilities — exponentiating logits and dividing by their sum so the values total one.
37. **Spinlock** — waiting by looping instead of sleeping — a lock that busy-waits, correct only for very short critical sections.
38. **Spread spectrum** — deliberately smearing a signal over more bandwidth — modulating over a wide band to resist interference and interception.
39. **Sprite** — a small image drawn over the scene — a 2D bitmap composited into a frame, historically with dedicated hardware.
40. **SQL injection** — smuggling commands inside data — an attack where unescaped input becomes part of a database query.
41. **SRAM** — fast memory built from logic, used in caches — static RAM using six transistors per bit, needing no refresh.
42. **SSD** — storage with no moving parts — a solid-state drive of NAND flash plus a controller, with microsecond latency.
43. **SSH** — a secure way to log in to another machine — the Secure Shell protocol on port 22, with host keys and public-key user authentication.
44. **SSID** — the name of a wireless network — the human-readable network identifier broadcast in beacons.
45. **SSRF** — tricking a server into fetching an internal URL — server-side request forgery, reaching private services from a trusted position.
46. **Stack** — the memory used for calls and locals — a LIFO region holding frames with return addresses, arguments and locals.
47. **Staging area** — what will go into the next commit — git's index, holding the exact content to be committed.
48. **Stateful firewall** — it remembers connections, not just packets — a filter tracking flow state so replies are allowed automatically.
49. **Static linking** — the library is baked into the binary — copying library code into the executable at build time.
50. **Streaming** — playing while it is still arriving — delivering media progressively rather than download-

ing it whole.

51. **Subnet** — a slice of an address range — a network defined by a prefix, sharing one broadcast domain in IPv4.
52. **Sub-pixel** — the red, green or blue part of a dot — one colour element of a pixel, exploitable for text rendering.
53. **Superscalar** — issuing several instructions each cycle — a core with multiple parallel execution ports and wide issue.
54. **Swap** — using disk when memory runs out — backing anonymous pages with storage so more can be committed than exists.
55. **Symbolic link** — a file that points at another path — a special file holding a path string, resolved at open time.
56. **System call** — a program asking the kernel to do something — the trap instruction and ABI by which user code enters the kernel.
57. **systemd** — the thing that starts everything on modern Linux — an init system and service manager with units, sockets and journal.

53.4.20 T

1. **Tag** — a fixed name for one commit — a ref under refs/tags, lightweight or an annotated object with message and signature.
2. **TCP** — the protocol that makes sure everything arrives in order — a connection-oriented transport with sequence numbers, acknowledgements and retransmission.
3. **Tearing** — the picture split across two frames — a display artefact when the buffer swaps mid-scanout, fixed by vsync.
4. **Temperature** — how adventurous the model's word choice is — a divisor on logits before softmax; lower is more deterministic.
5. **Thread** — one line of execution inside a program — a schedulable flow with its own stack and registers, sharing the process address space.
6. **Threat model** — deciding who you are defending against — an explicit statement of assets, adversaries, capabilities and out-of-scope risks.
7. **Three-way handshake** — the greeting that opens a TCP connection — the SYN, SYN-ACK, ACK exchange establishing sequence numbers.
8. **Throughput** — how much work gets done per second — completed operations or bits per unit time, distinct from latency.
9. **Tick rate** — how often the game server updates the world — server simulation steps per second, often 20 to 128 in shooters.
10. **TLB** — a small cache of address translations — translation lookaside buffer, holding recent virtual-to-physical page mappings.
11. **TLD** — the last label of a domain name — the top-level domain such as com, org or in, delegated from the root zone.
12. **TLS** — the encryption under HTTPS — Transport Layer Security; version 1.3, RFC 8446, needs one round trip and requires forward secrecy.
13. **Token** — the small piece of text a model actually sees — a subword unit from the tokenizer's vocabulary, roughly 0.75 English words each.
14. **Tokenizer** — the tool that cuts text into tokens — the encoder mapping text to integer IDs, usually by byte-pair encoding.
15. **Top-p** — keep only the most likely words that add up to p — nucleus sampling, restricting choice to the smallest set whose probability exceeds

    p.
16. **Trace** (observability) — the story of one request across services — a tree of timed spans sharing one trace identifier.

17. **Traceroute** — showing the routers along the way — sending packets with increasing TTL and reading the ICMP time-exceeded replies.
18. **Trailing whitespace** — invisible spaces at the end of a line — characters that break diffs and comparisons without being visible.
19. **Transaction** — all of it or none of it — a group of operations with atomicity, consistency, isolation and durability.
20. **Transformer** — the design behind modern language models — a stack of self-attention and feed-forward blocks, from the 2017 paper Attention Is All You Need.
21. **Transistor** — an electrical tap — a semiconductor switch or amplifier; in CMOS logic, a field-effect transistor.
22. **Transpiler** — a compiler from one language to another readable one — a source-to-source translator, as TypeScript to JavaScript.
23. **Trap** — a deliberate jump into the kernel — a synchronous exception raised by an instruction, such as a system call.
24. **Trilateration** — finding a position from several distances — computing a location from ranges to known points, as GPS does.
25. **TRIM** — telling the drive which blocks are now rubbish — a command letting the filesystem mark blocks free so the SSD can erase them early.
26. **Truncation** — cutting data that does not fit — discarding bits, characters or tokens beyond a limit, often silently.
27. **Truth table** — every input combination and its output — an exhaustive listing of a Boolean function's behaviour.
28. **TTL** (IP) — a hop counter that stops loops — a header field decremented per router, triggering ICMP time exceeded at zero.
29. **TTL** (DNS) — how long an answer may be cached — the seconds a resolver may reuse a record before asking again.
30. **TTY** — the terminal a program is attached to — a kernel device providing line discipline, echo and job control.
31. **Tunnel** — one network carried inside another — encapsulating packets inside another protocol, as with VPNs and GRE.
32. **Two-factor authentication** — a password plus something else — requiring a second independent factor such as a code or hardware key.
33. **Two's complement** — how computers store negative numbers — inverting all bits and adding one, giving one zero and a range from -2^(n-1) to 2^(n-1)-1.
34. **Type system** — rules about what values may be combined — a static or dynamic discipline classifying values and rejecting invalid operations.

53.4.21 U

1. **UDP** — send it and hope — a connectionless transport with no ordering, no retransmission and very little overhead.
2. **UEFI** — the modern replacement for the BIOS — firmware with its own boot manager, GPT support, drivers and optional Secure Boot.
3. **Unicode** — one number for every character in every script — a character set of code points from U+0000 to U+10FFFF, with 154,998 assigned in version 16.0.
4. **Unit test** — checking one small piece on its own — a fast, isolated test of a single function or class.
5. **Unreachable** — nothing can get there from here — an object with no path from a root, or a network with no route.
6. **Upstream** — the branch or project yours follows — the configured remote branch a local branch compares and pushes to.
7. **Uptime** — how long since the last restart — elapsed time since boot, not the same as availability.
8. **USB** — the connector that replaced all the others — Universal Serial Bus, from 12 Mbit/s in 1996 to 80

Gbit/s in USB4 Version 2.0.

9. **User mode** — the restricted half of the processor — the unprivileged state where a program cannot touch devices or kernel memory directly.
10. **User space** — everything outside the kernel — processes running unprivileged, reaching the kernel only by system calls.
11. **UTF-8** — Unicode written as bytes, ASCII still working — a variable-length encoding using one to four bytes per code point.
12. **UUID** — a name unique without asking anyone — a 128-bit identifier, usually version 4 random or version 7 time-ordered.

53.4.22 V

1. **Value** (attention) — the content a token contributes when attended to — a projected vector summed with attention weights to form the output.
2. **Vector database** — storage that finds things by meaning — an index over embeddings supporting approximate nearest-neighbour search.
3. **Version control** — keeping every version and who changed what — a system recording snapshots, authorship and lineage of files.
4. **Vertex** — one corner of a shape — a point with position and attributes, processed by the vertex stage.
5. **Virtual machine** — a whole fake computer inside a real one — an isolated guest with its own kernel, scheduled by a hypervisor.
6. **Virtual memory** — every program thinks it owns the whole address space — per-process address translation through page tables, with protection and paging.
7. **Virtualization** — one machine pretending to be several — hardware-assisted isolation of guests, using extensions such as VT-x or AMD-V.
8. **VLAN** — several separate networks over one set of wires — IEEE 802.1Q tagging, giving separate broadcast domains on shared links.
9. **Voice coil** — the coil that moves a speaker cone — a wire coil in a magnetic gap, driven by current to move the diaphragm.
10. **Volatile** (memory) — it forgets when power stops — storage requiring power to retain state, as DRAM and SRAM do.
11. **Voltage** — electrical push — potential difference in volts, the work per unit charge between two points.
12. **Volume** — storage attached to a container or machine — a named block or filesystem mount whose lifetime is separate from the workload.
13. **VPN** — a private tunnel across a public network — an encrypted tunnel giving remote hosts an address inside a protected network.
14. **Vsync** — waiting for the screen before swapping the picture — synchronizing buffer swaps to the display refresh to avoid tearing.
15. **Vulnerability** — a weakness someone can exploit — a flaw allowing an attacker to violate a security property, often tracked by CVE number.

53.4.23 W

1. **Wafer** — the thin silicon disc chips are made on — a 300 mm polished slice cut from a boule, carrying hundreds of dies.
2. **Watchdog** — a timer that reboots a stuck system — hardware or software resetting the system if not periodically reset by software.
3. **Wear levelling** — spreading writes so no cell dies early — an SSD controller policy distributing program and erase cycles across blocks.
4. **Webhook** — the server calls you when something happens — an HTTP POST sent by a service to a URL you registered.
5. **WebSocket** — a two-way channel over one web connection — RFC 6455, upgrading an HTTP connection to a persistent frame-based channel.

6. **Weight** — one number that scales an input — a learned multiplier in a layer's matrix, the bulk of a model's parameters.
7. **Wi-Fi** — wireless local networking — IEEE 802.11 standards, branded by generation, currently up to Wi-Fi 7 from 802.11be.
8. **Word** (CPU) — the natural size the machine works in — the register and native operand width, 64 bits on current mainstream CPUs.
9. **Workflow** — the file describing an automated pipeline — a declarative definition of triggers, jobs and steps for a CI system.
10. **Working set** — the pages a program actually needs right now — the memory touched over a recent interval, the amount that must stay resident.
11. **Working tree** — the actual files you can see and edit — the checked-out files in your directory, separate from the index and object store.
12. **WPA2** — the Wi-Fi encryption most networks still use — 802.11i security with AES-CCMP and a four-way handshake.
13. **WPA3** — the newer Wi-Fi encryption — security using SAE key exchange, resisting offline dictionary attacks on the passphrase.
14. **Write amplification** — the drive writing more than you asked — the ratio of physical NAND writes to host writes, driven by garbage collection.

53.4.24 X

1. **x86** — the instruction set in most PCs — Intel's CISC architecture from the 1978 8086, extended to 64 bits as x86-64 in 2003.
2. **XDR** — the widest brightness range a display claims — extreme dynamic range, a marketing name for HDR panels with high peak luminance.
3. **XML** — structured text with tags — a markup language for machine-readable data, standardized in 1998.
4. **XOR** — true when the inputs differ — the exclusive-or Boolean function, the core of parity, adders and simple ciphers.
5. **XSS** — injecting script into someone else's page — cross-site scripting, where untrusted input is rendered as executable script.

53.4.25 Y

1. **Y2K** — the year 2000 date problem — the failure risk from two-digit year fields, largely averted by remediation before 2000.
2. **YAML** — an indentation-based configuration format — a superset of JSON where structure comes from indentation, common in CI and Kubernetes.
3. **Yield** — the share of chips that come out working — the fraction of dies on a wafer passing test, the main driver of chip cost.

53.4.26 Z

1. **Zero trust** — trust nothing just because it is inside the network — an architecture authenticating and authorizing every request regardless of network position.
2. **Zero-day** — an attack for which no fix exists yet — a vulnerability exploited before the vendor has released a patch.
3. **Zombie process** — a finished process nobody has collected — a terminated child whose exit status has not yet been reaped by its parent.
4. **Zone** (DNS) — one slice of the name tree under one authority — a portion of the namespace served by one set of authoritative servers.

53.5 Where to go next

1. Understanding comes from doing. You did not learn to ride a bicycle by reading about balance, and this is the same.
2. Pick the one layer in this book that annoyed you most because you did not quite get it. Go there first. Discomfort is a good compass.
3. Build the smallest possible version of something you used today. A shell that runs one command. A web server that returns one line. A tiny assembler.
4. Then break it on purpose. Unplug the network mid-transfer. Fill the disk. Send a malformed header. You learn more from the failure than the success.
5. Read real source code, not just tutorials. Start with something small and old: a coreutils command, a toy compiler, an early version of a database.
6. Read the actual specifications. RFCs are shorter and clearer than their reputation. So is the IEEE 754 summary. So is the git object format.
7. Learn one tool that shows you the truth: a packet capture tool, a debugger, a profiler, or `strace`. One of these changes how you think permanently.
8. When something breaks, write down what you observed before you write down what you think it means. Keep those two columns separate. That is diagnosis.
9. Do not chase every new framework. The layers under them barely change. The person who knows the layers can pick up any framework in a week.
10. Teach it to someone. If you cannot explain it plainly, you have not finished learning it. That is the whole reason this book has a plain half.
11. Be sceptical of benchmark numbers, including ones you like. Ask what was measured, on what, by whom, and what was left out.
12. It is normal to forget most of the details. Keep this chapter. Remember the shape of the thing, and look the numbers up when you need them.
13. Nobody understands all of this at once, including the people who built it. Every expert you admire has a large map and a few deep holes.
14. You started with charge in a wire and finished with a language model. That is one continuous story, and you now know it end to end. Go and build.

53.6 The whole book in 100 lines

1. Everything starts with charge, a property of matter that pushes and pulls.
2. Voltage is push, current is flow, resistance is how hard the path fights back.
3. Ohm's law ties them together: voltage equals current times resistance.
4. Silicon is neither a good conductor nor a good insulator, and that is why it is useful.
5. Add tiny amounts of other elements and you get n-type and p-type silicon.
6. Join them and you get a junction that passes current in only one direction.
7. Add a third terminal and a small voltage controls a large flow. That is a transistor.
8. A transistor used as a switch is the only real trick in the whole machine.
9. Sand becomes a pure crystal, the crystal becomes a wafer, the wafer becomes chips.
10. Light prints the pattern, chemicals cut it, and the layers are built up in turn.
11. Not every die works, so yield quietly decides the price of everything.
12. Wire a few transistors one way and you get a gate: AND, OR, NOT, NAND.
13. NAND alone can build every other gate, and therefore every digital circuit.
14. Gates wired in a pattern add binary numbers without knowing what addition is.
15. Feed a gate's output back into its input and it holds a value. That is memory.
16. A clock decides when things may change, so signals have time to settle first.
17. Six transistors hold one bit quickly. That is SRAM, and it becomes cache.
18. One transistor and one capacitor hold one bit cheaply. That is DRAM, and it forgets.

19. Charge trapped on an isolated gate holds a bit with no power. That is flash.
20. Nothing in any of this knows what a number means.
21. Meaning is a convention. We agree that a pattern of bits stands for something.
22. Binary counts in twos. Hexadecimal writes four bits per digit for human eyes.
23. Negative numbers use two's complement, so one adder handles both signs.
24. Fractions use floating point, standardized as IEEE 754 in 1985, and it rounds.
25. Text is numbers too. ASCII gave 128 characters. Unicode gives over a million.
26. UTF-8 stores Unicode as bytes and leaves old ASCII files working unchanged.
27. Pictures are grids of pixels, and each pixel is three numbers for brightness.
28. A framebuffer holds those numbers, and the screen scans them out many times a second.
29. Sound is air pressure measured thousands of times a second and stored as numbers.
30. A converter turns those numbers back into a voltage that moves a coil and a cone.
31. A CPU fetches an instruction, decodes it, executes it, and writes the result.
32. It does this in a pipeline, so several instructions are in progress at once.
33. It guesses which way each branch will go, and it is usually right.
34. It runs work out of order and finishes it in order, to hide the waiting.
35. Memory is far slower than the core, so caches keep recent data close by.
36. A cache line is 64 bytes, and code that respects that runs several times faster.
37. Programs use virtual addresses, and hardware translates them through page tables.
38. That gives every process its own world and stops one program reading another's.
39. Storage is not memory. Storage keeps things when the power stops.
40. A hard disk seeks in milliseconds. A solid-state drive answers in microseconds.
41. Flash erases in big blocks, so drives shuffle data and hide where it really is.
42. Filesystems turn a flat array of blocks into names, folders and permissions.
43. Buses carry data between chips, and inside a computer the fast one is PCIe.
44. Firmware runs before any operating system and knows how to wake the machine.
45. Software is a pattern of bits held in the same memory as data. That is the big idea.
46. A compiler reads source text, works out what it means, and writes machine code.
47. It tokenizes, parses into a tree, checks meaning, optimizes, then emits code.
48. An interpreter skips the output file and carries out the meaning directly.
49. A linker joins the compiled pieces and a loader maps them into memory to run.
50. Languages are tools with opinions, each built for the problem of its own time.
51. An operating system shares one machine between many programs, safely.
52. The kernel is privileged. Your code is restricted and asks by system call.
53. A scheduler decides who runs next, thousands of times every second.
54. Drivers translate general requests into the exact language of one device.
55. The shell is only a program that reads your words and starts other programs.
56. Pipes join those programs so one program's output becomes another's input.
57. A file extension is a hint, not a fact. The bytes at the start are the fact.
58. An executable is a file with a header telling the loader how to start it.
59. Pressing a key sends a code that becomes an event that becomes a character.
60. A font renderer turns that character into pixels, composited into one frame.
61. A GPU exists because pictures need the same small sum done millions of times.
62. Thousands of simple lanes beat a few clever cores when the work is parallel.
63. A network is machines agreeing how to wrap, address and forward data.
64. Data is cut into packets so one big transfer cannot block everyone else.
65. Each layer adds a header and treats the layer above it as opaque payload.
66. An IP address names an interface. A MAC address names a port on one link.
67. Private addresses are reused everywhere, and NAT hides many machines behind one.
68. Routers forward by longest prefix match and know nothing of your conversation.
69. BGP is how independent networks tell each other which prefixes they can reach.

70. DNS turns names into addresses through a hierarchy of caches and authorities.
71. TCP adds sequence numbers, acknowledgements and retransmission to make a stream.
72. UDP adds almost nothing, which is exactly what real-time traffic wants.
73. Traceroute shows the path by sending packets that expire one hop further each time.
74. Silence is evidence. A timeout with no reply tells you what a refusal does not.
75. Diagnosis is separating what an observation proves from what it merely suggests.
76. Radio carries the same bits on a wave, shaped by modulation and shared by turns.
77. TLS wraps a connection in encryption, and a certificate chain proves who answered.
78. A firewall drops or rejects, a VPN tunnels, and a CDN moves content closer to you.
79. The web is documents fetched by HTTP, marked up in HTML, styled by CSS, moved by JavaScript.
80. An API is a promise about requests and answers, so machines can call machines.
81. Git stores snapshots, not differences, and names every object by its content hash.
82. A commit points at a tree and at its parents, and cannot change without changing its name.
83. Branches are only moving labels, which is why making one costs nothing.
84. Because the whole history is local, you can work fully with no network at all.
85. Merge joins two histories. Rebase rewrites yours onto a new base, with new hashes.
86. A forge adds accounts, review, permissions and automation on top of plain git.
87. Continuous integration runs your tests on a machine you do not own, against one exact commit.
88. Cloud is renting someone else's computers, with the hard parts already solved.
89. Containers isolate processes. Virtual machines isolate whole kernels.
90. Security is a threat model, least privilege, and honesty about what you cannot stop.
91. A game is a loop: read input, advance the world by a fixed step, draw, repeat.
92. Everything in that loop must fit the frame budget, or the player feels it.
93. Across a network the client guesses ahead, and the server has the final say.
94. Machine learning replaces rules you write with numbers fitted to examples.
95. A network of weighted sums and bends is trained by gradient descent on a loss.
96. Backpropagation works out which weight caused how much of the error.
97. A transformer lets every token look at every other token, all at once.
98. A language model only predicts the next token. Everything else follows from that.
99. It has no memory beyond its context, no access to truth, and no intent of its own.
100. From charge in a wire to a conversation with a machine, it is one continuous story, and you have now read it.

ABOUT THE AUTHOR

Shikhar Singh

Founder & Chairman

KedByte Technologies Private Limited

Shikhar Singh is the Founder & Chairman of KedByte Technologies Private Limited.

He wrote this book to solve a problem he kept meeting: capable people concluding they were not technical, when in truth nobody had ever shown them the bottom of the stack. His conviction is that computing has no genuinely hard idea in it, only a very long chain of simple ones, and that anybody who is walked along that chain in order can hold the whole of it in their head.

That conviction shapes the method used here. Explain it plainly, with a comparison and a worked example, so that it lands. Then explain it again in the exact professional vocabulary, with the real units and the real figures, so that it is usable. Say clearly where the plain version stops being true. Never let a reader carry away a confident misunderstanding.

Under his direction, KedByte Technologies Private Limited treats technical education as infrastructure rather than as a product feature: knowledge that should be transferable, verifiable, and free of the vocabulary barriers that keep capable people out of the field.

ABOUT THE PUBLISHER

KedByte Technologies Private Limited

KedByte Technologies Private Limited publishes technical work under the KedByte Press imprint.

The editorial standard applied to this volume is simple to state and difficult to meet. Every explanation must work twice: once for a reader who has never opened a terminal, and once for an engineer who needs the exact figure. Every claim that can be checked must be checked. Every figure that will go stale must carry a date. Where the field genuinely disagrees, both positions are given rather than one being quietly chosen.

Where this book uses real evidence, it uses it unaltered. The network trace in Part F, the failed pushes in Part G and the error messages quoted throughout are reproduced exactly as they were observed. Diagnosis is taught against ambiguous real data, because that is the only kind there is.

KEDBYTE

TECHNOLOGIES PRIVATE LIMITED

COLOPHON

This first edition of *How Computers Work* runs to nine parts and fifty-three chapters, and was set entirely from plain text.

The body text is set in TeX Gyre Pagella, a digital revival of Hermann Zapf's Palatino, chosen for long-form reading. Headings, tables and the KedByte identity are set in Poppins. Code, command output and diagrams are set in DejaVu Sans Mono, and every diagram in this book is drawn in plain ASCII so that it survives being copied anywhere.

The book was composed with pandoc and typeset with XeLaTeX on A4 stock. The single accent colour used on rules, chapter numbers and section headings is KedByte navy.

Shikhar Singh

Founder & Chairman, KedByte Technologies Private Limited

KB

First Edition • KedByte Press